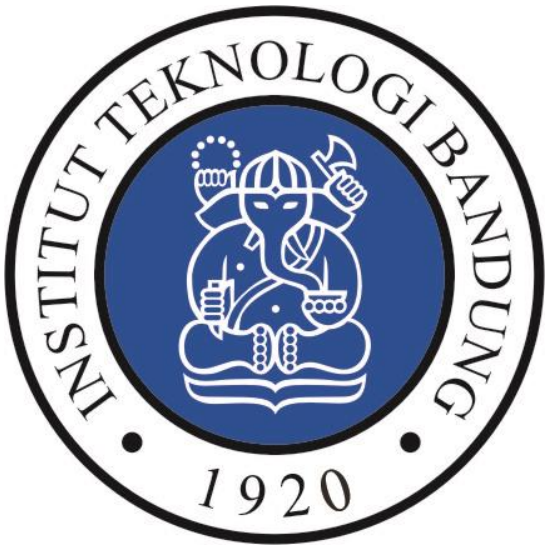
MODUL PRAKTIKUM EL3111

ARSITEKTUR SISTEM KOMPUTER

EDISI 2016

Semester I – Tahun Akademik 2016/2017



SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA
INSTITUT TEKNOLOGI BANDUNG
2016

Modul Praktikum EL3111
Arsitektur Sistem Komputer

Edisi 2016 untuk digunakan pada Semester I Tahun Akademik 2016/2017

Modul praktikum ini merupakan revisi dan pengembangan
dari modul praktikum yang telah digunakan pada tahun sebelumnya
dengan penyusun sebagai berikut.
Andri Haryono (Teknik Elektro 2006)
Frillazeus Goltha (Teknik Elektro 2006)
Niko Robbel (Teknik Elektro 2006)
Ardianto Satriawan (Teknik Elektro 2007)
Gilang Ilham (Teknik Elektro 2007)
Muhammad Johan A. (Teknik Elektro 2007)
Umar Abdul Aziz (Teknik Elektro 2007)
Tommy Gunawan (Teknik Elektro 2008)
Rizka Widyarini (Teknik Elektro 2009)
Silvia Anandita (Teknik Elektro 2010)
Yudi Isvara (Teknik Elektro 2010)
Bagus Hanindhito(Teknik Eletro 2011)
Baharuddin Aziz(Teknik Elektro 2011)
Muhammad Luqman (Teknik Elektro 2011)
Syaiful Andy (Teknik Elektro 2012)

Program Studi Teknik Elektro
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung

PRAKATA

Puji syukur kami panjatkan ke hadirat Allah swt. karena dengan petunjuk, rahmat, serta izin-Nya kami dapat menyelesaikan penyusunan Modul Praktikum EL3111 Arsitektur Sistem Komputer Edisi 2016 sebelum kegiatan praktikum dimulai. Modul Praktikum ini akan digunakan sebagai petunjuk pelaksanaan praktikum Arsitektur Sistem Komputer semester I tahun akademik 2016/2017.

Modul praktikum ini mengalami sedikit perbaikan dan penyesuaian terhadap peraturan-peraturan yang baru. Diantara perbaikan yang dilakukan adalah penyesuaian posisi gambar dengan tulisan, penambahan penjelasan mengenai konfigurasi *environment variables* pada *Windows 10,* instalasi *plugins HEX-Editor* pada Notepad++, dan beberapa perbaikan *minor* lainya.

Penulis mengucapkan terima kasih kepada Bapak Ir. Yudi Satria Gondokaryono, M.Sc., Ph.D. yang telah memberi penulis masukkan dan inspirasi dalam melakukan penyusunan modul praktikum ini. Penulis juga mengucapkan terima kasih kepada para koordinator asisten yang telah menjalankan dengan baik praktikum Arsitektur Sistem Komputer ini di tahun sebelumnya.

Penulis berharap modul praktikum ini dapat memberi penjalasan yang mudah dimengerti mengenai pelaksanaan praktikum Arsitektur Sistem Komputer. Lebih jauh lagi, penulis juga berharap modul praktikum ini dapat menumbuhkan ketertarikan praktikan dalam dunia arsitektur sistem komputer sehingga memicu timbulnya penelitian-penelitian baru di bidang ini.

Akhir kata, tidak ada gading yang tak retak, penulis menyadari bahwa modul praktikum ini masih jauh dari kata sempurna. Penulis menyambut dengan baik segala bentuk koreksi, saran, dan kritik terhadap modul praktikum ini.

Bandung, September 2016

Koordinator Praktikum Arsikom 2016

DAFTAR ISI

PERATURAN UMUM PRAKTIKUM

Berikut ini dijelaskan peraturan-peraturan umum yang berlaku selama pelaksanaan Praktikum EL3111 Arsitektur Sistem Komputer. Peraturan umum ini wajib dipatuhi oleh semua praktikan yang akan melaksanakan praktikum. **Pengabaian peraturan praktikum akan berakibat pada sanksi berupa pengurangan atau pengguguran nilai praktikum milik praktikan yang bersangkutan.**

Peraturan Sebelum Praktikum

Sebelum melakukan praktikum sesuai dengan jadwalnya, praktikan harus mempersiapkan diri dengan melakukan hal-hal sebagai berikut. Persiapan ini sangat berguna bagi praktikan untuk memahami materi-materi yang diperoleh saat praktikum dilaksanakan sekaligus melakukan analisis terhadap hasil yang diperoleh saat praktikum.

1. **Praktikan membaca dan memahami modul praktikum.**

   Praktikan diharapkan telah membaca dan memahami isi dari modul praktikum sehingga praktikan memperoleh gambaran besar (*overview*) terhadap praktikum yang akan dilaksanakan. Praktikan juga dapat mempelajari bahan-bahan serta materi yang berkaitan dengan praktikum yang akan dilaksanakan dari buku teks atau dari internet.

2. **Praktikan membawa Lembar Aktivitas untuk mendokumentasikan kegiatan sebelum, saat, dan setelah praktikum.**

   Praktikan diharuskan mencetak dan membawa lembar aktivitas (*activity log*) yang dicetak dalam kertas A4 secara bolak-balik. Format lembar aktivitas dapat dilihat pada bagian Petunjuk Teknis Pelaksanaan Praktikum. Lembar Aktivitas ini wajib dibawa karena merupakan salah satu syarat untuk mengikuti kegiatan praktikum. Praktikan yang tidak membawa lembar aktivitas tidak diperkenakan memasuki laboratorium untuk melaksanakan praktikum.

3. **Praktikan menulis ringkasan atau abstrak kegiatan praktikum yang akan dilaksanakan pada bagian Abstrak di lembar aktivitas.**

   Praktikan diwajibkan menulis ringkasan atau abstrak dari kegiatan praktikum yang akan dilaksanakan. Panduan menulis ringkasan atau abstrak ini dapat dilihat pada bagian Petunjuk Teknis Pelaksanaan Praktikum. Praktikan yang tidak menulis ringkasan atau abstrak tidak diperkenakan memasuki laboratorium untuk melaksanakan praktikum. Ringkasan atau abstrak ini dapat digunakan sebagai isi dari ringkasan atau abstrak pada Laporan Praktikum

4. **Praktikan mengerjakan Tugas Pendahuluan.**

   Praktikan wajib mengerjakan tugas pendahuluan. Panduan mengerjakan tugas pendahuluan dapat dilihat pada bagian Petunjuk Teknis Pelaksanaan Praktikum. Pada umumnya, tugas pendahuluan digunakan untuk membantu praktikan dalam melaksanakan praktikum.

Beberapa bagian dari modul praktikum dapat dikerjakan secara mandiri di luar laboratorium oleh praktikan. Hal ini sangat diperbolehkan untuk mengurangi beban praktikan saat pelaksanaan praktikum di laboratorium. **Jangan lupa tetap mengisi *activity log* ketika mengerjakan secara mandiri di luar laboratorium**. Beberapa perangkat lunak yang dibutuhkan untuk pelaksanaan

praktikum secara mandiri dapat diunduh dengan mudah (lihat Petunjuk Teknis Pelaksanaan Praktikum).

Peraturan saat Praktikum

1. **Praktikan hadir tepat waktu sesuai jadwal yang ditentukan.**

   Praktikum arsitektur sistem komputer dimulai pukul 13.30 WIB (GMT+7) dan diakhiri pukul 16.30 WIB (GMT+7). Praktikan harus hadir tepat waktu untuk melaksanakan praktikum. **Keterlambatan menghadiri praktikum menyebabkan pengurangan nilai akhir modul yang bersangkutan sesuai dengan lamanya keterlambatan**.

2. **Praktikan mengenakan kemeja, sepatu, sopan, dan membawa kelengkapan praktikum.**

   Praktikan mengenakan pakaian yang rapi dan sopan (kemeja, celana panjang/rok) dan mengenakan sepatu. Praktikan yang tidak mengenakan pakaian yang tidak sesuai dengan peraturan ini tidak diperbolehkan mengikuti praktikum. **Kelengkapan praktikum yang wajib dibawa antara lain modul praktikum (*softcopy* atau *hardcopy)*, lembar aktivitas dan tugas pendahuluan, alat tulis, dan kartu nama (*name tag*) Laboratorium Dasar Teknik Elektro. Praktikan diperbolehkan untuk menggunakan komputer atau laptop sendiri selama kegiatan praktikum berlangsung.**

3. **Praktikan mengisi daftar hadir, mengumpulkan tugas pendahuluan, menunjukkan lembar aktivitas kepada asisten, dan menulis nama pada Berita Acara Praktikum.**

   Praktikan datang tepat waktu dan mengisi daftar hadir, kemudian mengumpulkan tugas pendahuluan dan menunjukkan kepada asisten praktikum ringkasan atau abstrak mengenai praktikum yang akan dilaksanakan yang ditulis pada lembar aktivitas. Syarat ini mutlak untuk dapat mengikuti praktikum.

4. **Praktikan mengerjakan praktikum sesuai petunjuk pada modul praktikum.**

   Praktikan memanfaatkan seluruh waktu praktikum dengan baik untuk mengerjakan praktikum sesuai dengan petunjuk yang terdapat pada modul praktikum. Apabila terdapat pertanyaan, silakan diajukan dengan sopan pada asisten praktikum yang sedang bertugas. Praktikan wajib mencatat kegiatan-kegiatan selama di laboratorium pada lembar aktivitas.

5. **Praktikan menggunakan komputer dengan baik.**

   Praktikan menggunakan komputer yang tersedia di laboratorium sesuai dengan tujuan praktikum. Dilarang membuka program-program yang tidak ada hubungannya dengan praktikum. Praktikan juga harus dapat menghargai *privacy* orang lain dalam menggunakan komputer yang tersedia di laboratorium. Berhati-hatilah dengan ancaman keamanan (*virus*, *malware*, dsb.) yang dapat terjadi sewaktu-waktu. Apabila terjadi masalah dengan komputer yang digunakan, segera beritahu asisten praktikum yang sedang bertugas.

6. **Praktikan mengikuti tes akhir.**

   Tes akhir akan dilaksanakan 20 menit sebelum praktikum selesai. Praktikan harus mengikuti tes akhir ini sebagai bahan evaluasi pemahaman praktikan selama praktikum berlangsung.

Bobot dari tes akhir ini cukup besar sehingga praktikan diharapkan mengerjakan dengan baik. Praktikan mengerjakan tes akhir pada lembar aktivitas.

Praktikan diperbolehkan membawa dan menggunakan *laptop* masing-masing untuk melakukan aktivitas praktikum di laboratorium. Praktikan yang akan menggunakan *laptop* harus telah memasang perangkat lunak yang diperlukan saat praktikum. Peraturan saat praktikum dilaksanakan tetap berlaku meskipun praktikan menggunakan *laptop* sendiri saat kegiatan praktikum berlangsung.

Peraturan setelah Praktikum

1. **Praktikan mengumpulkan lembar aktivitas kepada asisten.**

   Lembar aktivitas yang telah terisi lengkap (ringkasan/abstrak praktikum, kegiatan di laboratorium, dan jawaban tes akhir) dikumpulkan kepada asisten praktikum.

2. **Praktikan merapikan kembali meja kerja yang digunakan saat praktikum.**

   Praktikan diwajibkan untuk merapikan kembali meja kerja yang digunakan saat praktikum. Praktikan harus tetap menjaga kebersihan ruang praktikum setelah digunakan.

3. **Praktikan menghapus semua data praktikum dari komputer di laboratorium setelah selesai praktikum.**

   Praktikan diwajibkan untuk memindahkan semua data praktikum dari komputer di laboratorium ke perangkat penyimpanan *portable* sebelum menghapus data praktikum dari komputer di laboratorium. Jangan lupa untuk mematikan komputer di laboratorium sebelum meninggalkan ruangan

4. **Praktikan menulis laporan hasil praktikum dan mengumpulkannya dalam bentuk *softcopy* dalam jangka waktu tiga hari biasa setelah hari praktikum dilaksanakan.**

   Praktikan diwajibkan untuk menulis laporan hasil praktikum sesuai dengan format yang berlaku (lihat Petunjuk Teknis Pelaksanaan Praktikum). Laporan hasil praktikum dikumpulkan paling lambat tiga hari biasa setelah praktikum dilaksanakan (lihat Petunjuk Teknis Pelaksanaan Praktikum). Pengumpulan laporan hasil praktikum dilakukan bersama dengan pengumpulan kode program yang dibuat selama praktikum.

Pertukaran Jadwal Praktikum

Pertukaran jadwal praktikum dapat dilakukan per orang dengan modul yang sama. Pada dasarnya pertukaran jadwal praktikum cukup dilakukan antarpraktikan yang jadwal praktikumnya akan ditukar. Apabila kedua praktikan telah setuju untuk menukar jadwal praktikum, praktikan cukup memberitahu asisten praktikum harian pada kedua jadwal yang ditukar.

Plagiarisme

Apabila ditemukan praktikan yang terbukti melakukan plagiarisme, yang bersangkutan akan diproses sesuai dengan norma akademik yang berlaku di Institut Teknologi Bandung.

PETUNJUK TEKNIS PELAKSANAAN PRAKTIKUM

Server Informasi Materi Praktikum

Untuk keperluan pelaksanaan Praktikum EL3111, disediakan sebuah server untuk menyimpan materi praktikum. Alamat server ini dapat diakses melalui URL **http://el3111.bagus.my.id**. Ikuti petunjuk dan menu yang tersedia pada server tersebut. Apabila server mengalami gangguan, silakan mengirimkan e-mail ke **syaifulandy@gmail.com** agar dapat segera ditindaklanjuti.

Komposisi Penilaian

Praktikum ini terdiri atas lima modul praktikum. Perhitungan indeks akhir akan dilakukan berdasarkan jumlah nilai dari seluruh modul praktikum yang kemudian dibandingkan dengan statistik perolehan nilai seluruh peserta praktikum. Setiap modul memiliki komposisi penilaian sebagai berikut.

Komponen	Bobot
Sebelum Praktikum	
Ringkasan / Abstrak Praktikum. Berisi ringkasan praktikum yang akan dilakukan. Ringkasan ini ditulis pada lembar aktivitas.	5 %
Tugas Pendahuluan Tugas pendahuluan untuk modul praktikum yang bersangkutan. Bila tidak ada tugas pendahuluan, maka ringkasan / abstrak praktikum berbobot 10%.	5 %
Saat Praktikum	
Aktivitas Praktikum Penilaian meliputi kelengkapan praktikum yang disiapkan oleh praktikan, sikap dan perilaku praktikan, serta kerjasama praktikan dalam satu kelompok praktikum.	15%
Kode (*Source Code*) Penilaian meliputi tugas praktikum yang dapat diselesaikan oleh praktikan selama waktu praktikum yang dibuktikan dengan adanya kode yang dapat dieksekusi atau disintesis. Tugas praktikum yang belum selesai wajib dikerjakan di luar praktikum untuk dianalisis pada laporan praktikum	15%
Tes Akhir Pemahaman praktikan terhadap materi praktikum yang dibuktikan dengan tes akhir	20%
Setelah Praktikum	
Laporan Praktikum Pemaparan hasil dan analisis praktikum oleh praktikan dalam suatu format laporan berstandar IEEE yang dikumpulkan tiga hari kerja setelah praktikum	40%
Total Nilai	**100%**

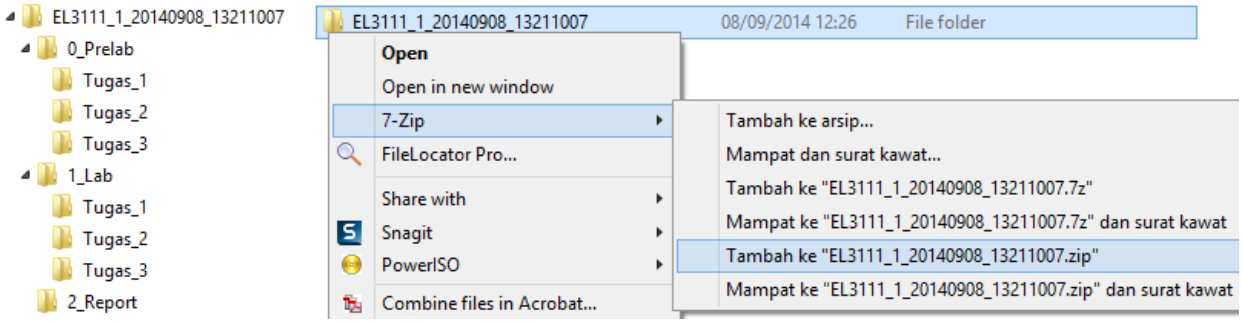
Struktur Folder Kerja

Praktikan diharuskan membuat folder kerja dengan struktur folder sebagai berikut. Struktur folder kerja ini digunakan agar file-file pekerjaan termasuk kode dan project terorganisasi dengan baik. Nama folder *root* memiliki format penamaan sebagai berikut.

Nama Folder Root : **EL3111_[No.Modul]_YYYYMMDD_[NIM]**
Nama File Zip : **EL3111_[No.Modul]_YYYYMMDD_[NIM].zip**

Nama File Laporan **: EL3111_[No.Modul]_YYYYMMDD_[NIM].pdf**

Folder **0_Prelab** digunakan untuk menyimpan kode (*source code*) tugas yang dikerjakan sebelum memasuki laboratorium (bila ada), contohnya tugas pendahuluan. Beberapa tugas pendahuluan mungkin tidak memerlukan kode (*source code*) sehingga tidak perlu memasukkannya ke dalam folder ini. Folder **1_Lab** digunakan untuk menyimpan tugas-tugas yang dikerjakan saat praktikum di laboratorium. Di dalam masing-masing folder **Lab** dan **Prelab** terdapat subfolder yang sesuai dengan nomor tugas yang diberikan. Folder **2_Report** digunakan untuk menyimpan **file PDF dari Laporan Praktikum**. Perhatikan bahwa folder kerja harus dilampirkan bersama dengan laporan praktikum dalam bentuk terkompresi **ZIP.** Nama file ZIP harus sama dengan nama folder *root*. Disarankan menggunakan 7-Zip untuk mempermudah proses kompresi. **File berekstensi .exe TIDAK perlu dikirimkan ke email asisten, HANYA *source code*-nya saja.**



File ZIP ini kemudian dikirim ke *email* **masing-masing asisten** dengan ketentuan sebagai berikut:

Subjek : NIM Asisten (tanpa tanda kurung)

Isi Email : Modul_NIM Praktikan_Nama Praktikan_NIM Asisten_Nama Asisten

Attachment : tidak lebih dari 5 MB

Contoh:

Kepada : el3111.arsikom@gmail.com

Subjek : 23216119

Isi : III_132120100_Fahmi Kurniawan_23216119_Syaiful Andy

Batas pengumpulan file ZIP (*source code,* laporan praktikum dalam bentuk PDF) adalah tiga hari biasa setelah praktikum dilaksanakan pukul 13.30 GMT+7.00. Apabila ada masalah saat pengiriman, maka praktikan dapat menguploadnya di *google drive* pribadi atau *cloud* lainya dan memberikan akses folder laporan kepada asisten, sehingga pada *email* tidak perlu mencantumkan *attachment* cukup link ke *drive* pribadinya saja. TIDAK ADA alasan dalam keterlambatan pengumpulan laporan praktikum.

Tugas Pendahuluan

Tugas pendahuluan harus dikerjakan sebelum praktikan memasuki ruang praktikum untuk menambah pemahaman praktikan mengenai materi praktikum yang akan dilakukan. Format lembar jawab tugas pendahuluan dapat diunduh pada **http://el3111.bagus.my.id**. Praktikan dapat

mencetak dokumen ini pada kertas HVS A4. Praktikan dapat langsung mengerjakan tugas pendahuluan dengan ditulis tangan atau ditik menggunakan komputer. Sangat disarankan untuk menggunakan komputer dalam mengerjakan tugas pendahuluan karena beberapa tugas pendahuluan akan berisi kode-kode yang cukup panjang apabila ditulis menggunakan tangan. Tugas pendahuluan harus dikumpulkan kepada asisten praktikum sebelum praktikum dimulai.

Lembar Aktivitas (*Activity Log*)

Lembar aktivitas merupakan lembar yang harus diisi dimulai dari sebelum melakukan praktikum, saat melakukan praktikum, hingga setelah melakukan praktikum. Format lembar aktivitas ini dapat diunduh pada **http://el3111.bagus.my.id**. Praktikan mencetak lembar aktivitas ini pada satu lembar HVS A4 secara bolak-balik. Lembar aktivitas dikumpulkan setelah tes akhir dilaksanakan. Lembar aktivitas terdiri atas tiga bagian.

1. **Ringkasan / Abstrak Praktikum**

   Bagian pertama ini harus dilengkapi oleh praktikan sebelum praktikum dimulai. Ringkasan / abstrak praktikum berisi penjelasan singkat mengenai praktikum yang akan dilakukan. Beberapa hal yang dapat ditulis dalam ringkasan atau abstrak praktikum sebagai berikut.

   - Apa saja yang akan dilakukan dalam praktikum?
   - Apa saja perangkat lunak atau bahasa pemrograman yang digunakan?
   - Bagaimana hasil yang diharapkan dari praktikum?

2. **Dokumentasi Kegiatan**

   Bagian kedua ini dilengkapi oleh praktikan saat praktikum dimulai. Praktikan menulis apa saja yang dilakukan selama praktikum beserta waktunya. Dokumentasi kegiatan **wajib** mencantumnya waktu serta deskripsi pekerjaan. Dokumentasi kegiatan diisi selama kegiatan praktikum berlangsung, bukan di akhir waktu praktikum. Contohnya sebagai berikut.

No.	Waktu	Deskripsi
1.	13:45	Membuat program hello_world.c
2.	13.55	Menguji program hello_world.c dan hasil pengujian berhasil
3.	13.58	Komputer mengalami *freeze* sehingga harus di-*restart*

3. **Tes Akhir**

   Tes akhir dikerjakan pada lembar aktivitas dengan menulis pertanyaan yang diberikan oleh asisten praktikum sebelum menjawab pertanyaan tersebut.

Tes Akhir

Tes akhir dilaksanakan dua puluh menit sebelum praktikum diakhiri (sekira pukul 16.10). Tes akhir ini dilakukan untuk menguji seberapa jauh praktikan memahami materi praktikum. Soal tes akhir diberikan oleh asisten praktikum yang bertugas dengan jumlah maksimal lima soal. Tes akhir ini dikerjakan pada lembar aktivitas.

Laporan Praktikum

Setelah melaksanakan praktikum, setiap praktikan wajib mengerjakan laporan praktikum. Laporan praktikum memberi sumbangan terbesar kepada komposisi nilai praktikum sehingga

diharapkan praktikan mengerjakan laporan praktikum dengan baik. Format laporan praktikum sesuai standar publikasi *paper* IEEE dan *template* dapat diunduh pada **http://el3111.bagus.my.id**. **Jangan lupa memberikan foto diri pada laporan**. Isi laporan praktikum sebagai berikut.

1. Abstrak. Abstrak dapat menyalin ringkasan / abstrak praktikum yang dikerjakan pada lembar aktivitas dengan beberapa pengubahan seperlunya. Jangan lupa memberikan kata kunci pada bagian abstrak.
2. Pendahuluan. Bagian ini berisi penjelasan singkat mengenai praktikum yang akan dilakukan, tujuan praktikum, metode praktikum, serta hasil-hasil yang diperoleh.
3. Landasan Teoretis. Bagian ini berisi pemaparan landasan teoretis berdasarkan rujukan pustaka yang sesuai dan dicantumkan pada Daftar Referensi.
4. Hasil dan Analisis. Bagian ini berisi hasil dan analisis setiap tugas praktikum. Hasil dapat berupa tangkapan layar (*screenshot*), kode (*source code*), maupun tabel dan grafik. Bila kode dan tangkapan layar terlalu besar, lebih baik melampirkannya pada bagian lampiran sehingga pemaparan hasil cukup merujuk lampiran. Analisis dilakukan terhadap hasil yang diperoleh selama praktikum dikaitkan dengan teori yang ada. Beberapa pertanyaan diberikan pada modul praktikum untuk membantu praktikan dalam melakukan analissis.
5. Simpulan. Bagian ini berisi simpulan dari praktikum yang dilakukan dalam bentuk poin-poin. Simpulan hendaknya menjawab tujuan praktikum yang telah didefinisikan pada pendahuluan.
6. Lampiran. Bagian ini berisi lampiran kode (*source code*) yang disusun sesuai urutan yang baik dan benar.

Laporan praktikum harus dikonversi ke dalam **Portable Document Format (PDF)** sebelum dimasukkan ke dalam folder **2_Report** pada folder kerja. Pada Microsoft® Office versi 2007 atau yang lebih baru, konversi file DOCX ke dalam PDF cukup dilakukan dengan melakukan *save-as* dengan tipe file PDF.

Penulisan Kode (*Source Code*)

Setiap kode program harus diberi *header* dengan menyesuaikan modul, percobaan, tanggal, kelompok, rombongan, nama praktikan, NIM praktikan, dan nama file. Untuk kode dalam bahasa C, *header* didefinisikan sebagai berikut.

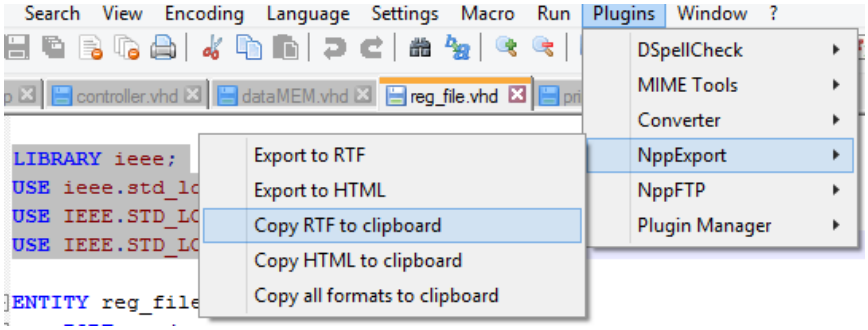
```
// Praktikum EL3111 Arsitektur Sistem Komputer
// Modul           :     2
// Percobaan       :     0
// Tanggal         :     7 November 2013
// Kelompok        :     VI
// Rombongan       :     A
// Nama (NIM) 1    :     Audra Fildza Masita (13211008)
// Nama (NIM) 2    :     Bagus Hanindhito (13211007)
// Nama File       :     printbitbyte.h
// Deskripsi       :     Menampilkan informasi bit dan byte dalam memory
```

Untuk kode dalam bahasa VHDL, *header* didefinisikan sebagai berikut.

```
-- Praktikum EL3111 Arsitektur Sistem Komputer
-- Modul           :     4
-- Percobaan       :     3
```

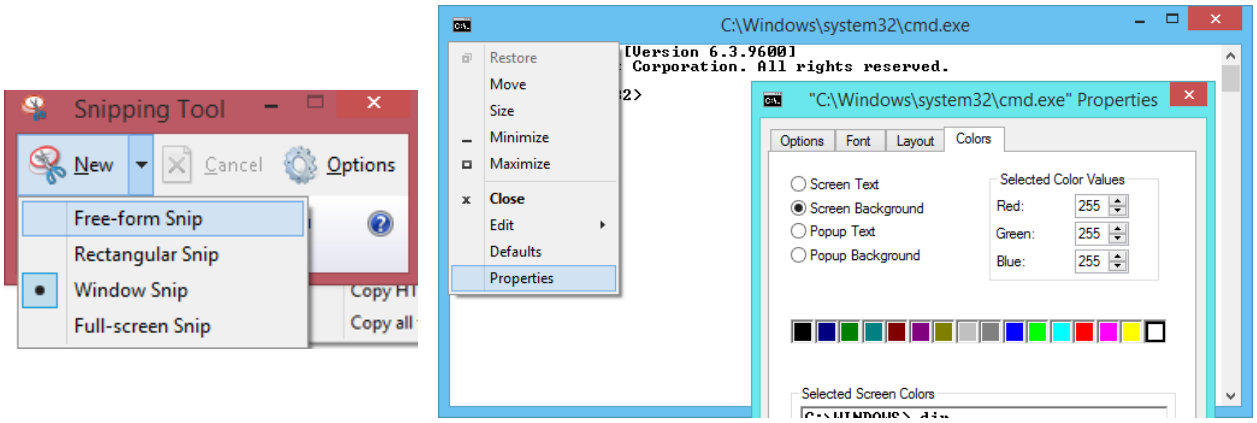
```
-- Tanggal          :      18 November 2013
-- Kelompok         :      VI
-- Rombongan        :      A
-- Nama (NIM) 1     :      Audra Fildza Masita (13211008)
-- Nama (NIM) 2     :      Bagus Hanindhito (13211007)
-- Nama File        :      reg_file.vhd
-- Deskripsi        :      Mengimplementasikan register pada MIPS
```

Untuk memasukkan kode (*source code*) pada laporan praktikum, gunakan program Notepad++ sebagai berikut. Pilih teks *source code* yang akan disalin ke laporan praktikum lalu pilih *Copy RTF to clipboard* dari NppExport di menu *Plugins* pada Notepad++. Selanjutnya, kita cukup menggunakan menu *paste* untuk menyalin kode tersebut ke dalam Microsoft® Office Word. Gunakan sebuah kontainer untuk meletakkan kode tersebut dalam Microsoft® Office Word yang dapat dibentuk menggunakan tabel 1x1 (1 baris, 1 kolom).



Tangkapan Layar (*Screenshot*)

Untuk melakukan tangkapan layar, kita dapat menggunakan Snipping Tool yang tersedia pada sistem operasi Microsoft® Windows™ 7/8/8.1. Gunakan pilihan *Free-form Snip* atau *Window-Snip* untuk melakukan *screenshot* jika tidak semua area layar perlu dimasukkan ke dalam gambar *screenshot*. Khusus untuk melakukan *screenshot* Command Prompt, pastikan warna latar (*background*) diubah menjadi warna putih dan warna teks diubah menjadi warna hitam.



DAFTAR ASISTEN PRAKTIKUM

Nama Asisten	NIM	E-mail	Keterangan
Syaiful Andy	23216119	syaifulandy@gmail.com	Koordinator
Muhammad Isnain H.	13213046	mihisnain@gmail.com	Asisten
Fadel Mahadika Putra	13213082	fadel.mahadika@gmail.com	Asisten
Yosi Aditya Nugroho	13213098	yosyosiyos@gmail.com	Asisten
Anggara Meristya	13213025	anggameristya@gmail.com	Asisten
Yudi Pratama	13212036	yudifrank@gmail.com	Asisten
Adinda Rana Trisanti	13213071	adindaranat@gmail.com	Asisten
Erik Yudistira	13213099	erik.yudistira31@gmail.com	Asisten
Bakti Satria Adhityatama	23216123	bakti.satria.a@gmail.com	Asisten
Fahmi Kurniawan	23216124	fahmikurniawan007@gmail.com	Asisten
Asep Hermansyah	13212006	asepherman99@gmail.com	Asisten

PERCOBAAN I

COMPILER BAHASA C DAN BAHASA ASSEMBLY INTEL® X86

Tujuan Praktikum

- Praktikan memahami tahap-tahap kompilasi program dalam bahasa C sebagai bahasa tingkat tinggi hingga diperoleh bahasa tingkat rendah yang dapat dieksekusi oleh mesin.
- Praktikan mampu melakukan kompilasi program bahasa C menggunakan *compiler* GCC beserta penggunaan makefile dan batch file.
- Praktikan memahami bahasa *assembly* dan mampu melakukan analisis terhadap bahasa *assembly* Intel® x86 yang dihasilkan oleh *compiler* GCC.
- Praktikan memahami penggunaan *stack memory* pada setiap *procedure call*.

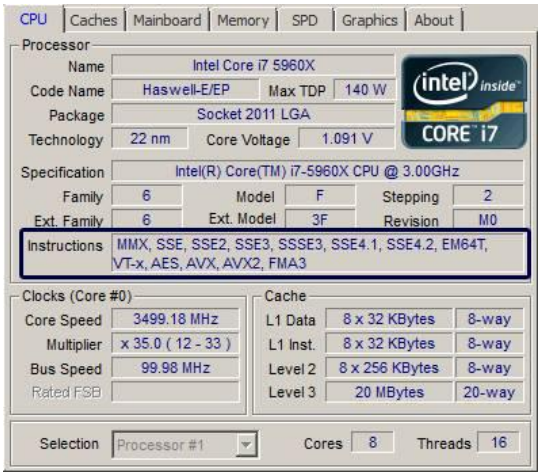
Perangkat Praktikum

- Komputer Desktop / Laptop dengan sistem operasi Microsoft® Windows™ 7/8/8.1
- *Compiler* GCC dalam paket program CodeBlocks untuk melakukan kompilasi program (penjelasan baca Lampiran I, program dapat diunduh di **http://el3111.bagus.my.id**).
- Notepad++ sebagai teks editor dan *heksadesimal editor* (baca Lampiran II).

Landasan Teoretis Praktikum

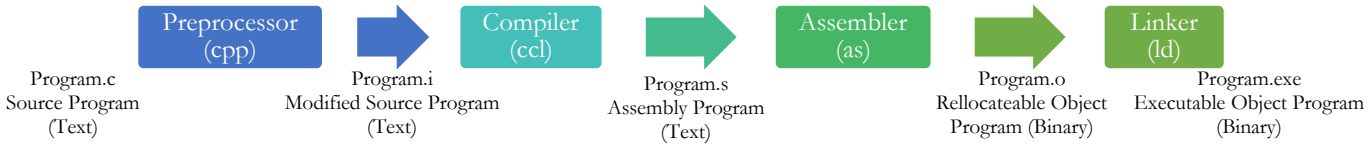
Kompilasi menggunakan GCC

Untuk membuat suatu program, bahasa tingkat tinggi cenderung banyak digunakan karena bahasa tingkat tinggi ini mudah dimengerti oleh manusia seperti halnya bahasa C. Sayangnya, bahasa tingkat tinggi tidak dapat dimengerti oleh mesin (mikroprosesor) sehingga tidak dapat dieksekusi. Oleh karena itu, diperlukan sebuah penerjemah bahasa pemrograman tingkat tinggi menjadi bahasa tingkat rendah yang berisi urutan instruksi yang dimengerti oleh mesin. Urutan instruksi tersebut kemudian dikemas dalam suatu bentuk *executable object program* yang disimpan dalam bentuk file biner.



Kumpulan instruksi yang dimengerti oleh mesin disebut *instruction set*. Dari sisi *instruction set*, terdapat dua penggolongan mesin (mikroprosesor) yaitu *complex instruction set computer* (CISC), contohnya mikroprosesor Intel®, dan *reduced instruction set computer* (RISC), contohnya MIPS32®. Beberapa mikroprosesor Intel® bahkan memiliki tambahan set instruksi seperti MMX, SSE, dan sebagainya.

Proses menerjemahkan baris kode program dalam bahasa C menjadi file *executable* dilakukan dalam empat langkah yaitu *preprocessor*, *compiler*, *assembler*, dan *linker* yang seluruhnya disebut sistem kompilasi.



- *Preprocessor*

  Semua perintah *preprocessor* yang ditulis dalam bahasa tingkat tinggi akan diproses terlebih dahulu oleh *preprocessor* sebelum *compiler* melaksanakan tugasnya. Beberapa tugas dari *preprocessor* ini adalah sebagai berikut.

  - Semua komentar dalam file program diganti dengan spasi satu buah.
  - Semua `\n` (*backslash-newline*) yang menandakan baris baru akan dihapus tidak peduli dimanapun dia berada. Fitur ini memungkinkan kita untuk membagi baris program yang panjang ke dalam beberapa baris tanpa mengubah arti.
  - Macro yang telah didefinisikan diganti dengan definisinya.

  Contohnya, pada perintah `#define MAX_ROWS 10`, *preprocessor* akan mengganti semua kata `MAX_ROWS` dengan `10`. Pada perintah `#include <stdio.h>`, *preprocessor* akan mengganti baris tersebut dengan isi file `stdio.h`.

- *Compiler*

  Compiler akan menerjemahkan bahasa tingkat tinggi C menjadi kode *assembly*. Kode *assembly* ini berisi instruksi-instruksi yang sesuai dengan *instruction set* yang dimiliki oleh mesin. File yang dihasilkan pada tahap ini masih berupa file teks (.s).

- *Assembler*

  Assembler akan menerjemahkan bahasa *assembly* menjadi file objek. File objek ini merupakan file biner (.o).

- *Linker*

  Linker akan menggabungkan file biner yang diperoleh pada tahap sebelumnya dengan file biner lain yang merupakan *dependency* dari program yang dibuat, contohnya *library* untuk menjalankan fungsi `printf`. Hasil dari *linker* berupa file biner *executable* (dalam platform Microsoft® Windows™, file ini memiliki akhiran .exe).

Untuk melakukan proses kompilasi menggunakan GCC, kita dapat menggunakan *Command Prompt* pada Microsoft® Windows™. Perhatikan bahwa GCC harus terpasang dan terkonfigurasi dengan benar (lihat pada lembar lampiran petunjuk instalasi dan konfigurasi GCC). Beberapa perintah untuk melakukan kompilasi antara lain sebagai berikut.

- Hanya melakukan proses *preprocessing*

```
gcc -E Program.c
```

  Eksekusi perintah tersebut akan menampilkan di layar *Command Prompt* kode `Program.c` setelah melalui proses *preprocessing*. Agar memperoleh *output* berupa file, dapat menggunakan tambahan perintah sebagai berikut.

```
gcc -E Program.c > Program.i
```

Eksekusi perintah tersebut akan menghasilkan file `Program.i` berisi kode `Program.c` yang telah melalui *preprocessing* pada folder yang sama dengan file `Program.c`. File ini dapat dibuka dengan teks editor contohnya Notepad++.

- Hanya melakukan proses *preprocessing* dan *compiling*

```
gcc -S Program.c
```

Eksekusi perintah tersebut akan menghasilkan file `Program.s` yang berisi baris instruksi *assembly* pada folder yang sama dengan `Program.c`. File ini dapat dibuka dengan teks editor contohnya Notepad++.

- Hanya melakukan proses *preprocessing*, *compiling*, dan *assembly*

```
gcc -c Program.c
```

Eksekusi perintah tersebut akan menghasilkan file `Program.o` yang merupakan file biner. File ini dapat dibuka dengan program *hex editor* contohnya HexEdit atau dengan menggunakan plugin HEX-Editor yang berjalan di Notepad++.

- Melakukan seluruh proses kompilasi (*preprocessing, compiling, assembly,* dan *linking*)

```
gcc -o Program.exe Program.c
```

Eksekusi perintah tersebut akan menghasilkan `Program.exe` yang dapat langsung dieksekusi (dijalankan). Kita juga dapat melakukan kompilasi dua file bahasa C sekaligus.

```
gcc -o Program.exe sub.c main.c
```

Disassembly menggunakan GCC

Selain dapat melakukan kompilasi, paket *compiler* GCC juga menyertakan sebuah *disassembler* yang mampu melakukan *disassembly* file biner (`.o` atau `.exe`) menjadi file *assembly* (`.s`) bernama Object Dump. Untuk melakukan *disassembly*, kita dapat menggunakan perintah berikut.

```
objdump -d Program.o
```

```
objdump -d Program.exe
```

Hasil dari proses *disassembly* ditampilkan pada jendela *Command Prompt*. Agar hasil dari proses *disassembly* dapat disimpan ke dalam suatu file, kita dapat menggunakan perintah berikut.

```
objdump -d Program.o > Program.s
```

```
objdump -d Program.exe > Program.s
```

Dengan demikian, hasil proses *disassembly* akan disimpan dalam file `Program.s`.

Optimisasi Program melalui Proses Kompilasi

GCC mendukung beberapa tingkat optimisasi program yang dapat dilakukan saat proses

kompilasi dilakukan. Terdapat beberapa tingkat optimisasi program yang dapat dipilih dengan menambahkan *flag* optimisasi saat melakukan kompilasi program. Umumnya optimisasi program merupakan *trade-off* antara *execution speed*, *program size*, *compilation time*, dan kemudahan dalam melakukan *debugging*.

```
gcc -O2 -o Program.exe Program.c
```

Flag `-O2` tersebut menandakan bahwa proses kompilasi dilakukan dengan optimisasi tingkat dua. Beberapa *flag* optimisasi yang dikenali oleh GCC adalah `-O0`, `-O1`, `-O2`, `-O3`, `-Os`, dan `-Ofast`. Perbedaan masing-masing level optimisasi diberikan sebagai berikut.

Makefile dan Batch File

Untuk suatu *project* yang terdiri atas beberapa file kode, tentu akan sangat merepotkan untuk melakukan kompilasi dengan menggunakan perintah kompilasi yang ditulis pada *command prompt* satu per satu untuk setiap file. GCC memiliki fitur makefile yang berfungsi untuk menulis daftar nama file kode di dalam *project* tersebut. Kita cukup memberikan GCC nama makefile lalu GCC akan melakukan proses kompilasi untuk semua file tersebut untuk kemudian menggabungkannya pada file *executable*. Makefile dapat bersifat sederhana hingga kompleks, bergantung pada sejauh mana kita menggunakan makefile untuk mengorganisasikan *project* kita. Contoh isi dari makefile adalah sebagai berikut.

```
all: contoh

contoh: main.o text.o
	gcc main.o text.o -o contoh.exe

main.o: main.c
	gcc -c main.c

text.o: text.c
	gcc -c text.c
```

GCC dapat diperintahkan untuk melakukan kompilasi makefile dengan perintah sebagai berikut.

```
mingw32-make -f makefile
```

Perintah tersebut akan melakukan kompilasi terhadap makefile yang diberikan menjadi sebuah program bernama `contoh.exe`. Program ini dihasilkan oleh hasil *linker* terhadap dua file objek bernama `main.o` dan `text.o` (tentunya termasuk dengan *library* yang lain yang dibutuhkan). Untuk memperoleh `main.o`, GCC harus melakukan kompilasi *source code* `main.c` menjadi file objek. Begitupula untuk memperoleh `text.o`, GCC harus melakukan kompilasi *source code* `text.c`.

Pada platform Microsoft® Windows™, terdapat sebuah file *shell script* bernama Windows™ Batch File. Kita dapat menuliskan perintah-perintah yang biasa kita tuliskan secara terpisah pada *command prompt* dalam suatu file yang disimpan dengan ekstensi .bat. Untuk mengeksekusi perintah-perintah tersebut, kita cukup menjalankan file .bat tersebut sehingga *command prompt* terbuka dan perintah-perintah yang kita tuliskan dieksekusi secara otomatis. Contoh Windows™ Batch File adalah sebagai berikut.

```
%~d0
cd "%~dp0"
gcc -O2 -E code.c > Program.l
gcc -O2 -S code.c
gcc -O2 -c code.c
```

```
gcc -O2 -o code.c
pause
objdump -d code.o > dump_o.dmp
objdump -d prog.exe > dump_exe.dmp
pause
```

Windows™ Batch File tersebut berisi perintah sebagai berikut. Perintah `%~d0` memerintahkan *command prompt* untuk berpindah *drive letter* ke *drive letter* yang sesuai dengan lokasi Windows™ Batch File berada. Selanjutnya, perintah `cd "%~dp0"` memerintahkan *command prompt* untuk berpindah folder ke lokasi Windows™ Batch File berada. Selanjutnya, *command prompt* mengeksekusi perintah yang memanggil GCC secara berurutan hingga berhenti akibat adanya perintah **pause**. Untuk melanjutkan eksekusi, kita cukup menekan sebarang tombol pada *keyboard* sehingga *command prompt* mengeksekusi perintah selanjutnya yaitu Object Dump.

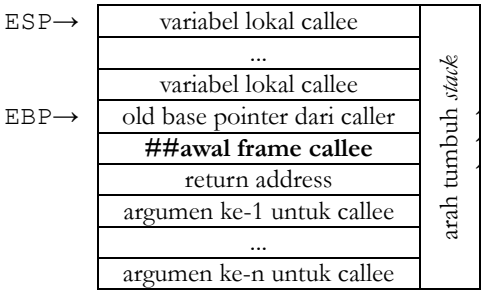
Instruksi dan Bahasa Assembly Intel® x86

Arsitektur mikroprosesor Intel® x86 merupakan salah satu arsitektur mikroprosesor yang banyak digunakan. Dengan mempelajari bahasa *assembly* dan instruksi Intel® x86, kita akan sangat terbantu dalam melakukan proses *debugging* dan optimisasi program yang kita buat. Dalam mikroprosesor Intel® x86, terdapat banyak *register* yang dapat digunakan. Namun, pada praktikum kali ini, kita cukup mempelajari beberapa *register* berikut.

- `EAX`, `EBX`, `ECX`, dan `EDX` adalah *register* 32-bit yang bersifat *general storage*.
- `ESI` dan `EDI` adalah *register* 32-bit yang digunakan sebagai *indexing register*. *Register* ini juga dapat digunakan sebagai *general storage*.
- `ESP` adalah *register* 32-bit yang digunakan sebagai *stack pointer*. Dengan demikian, `ESP` akan berisi nilai alamat (*address*) elemen puncak (*top element*) dari *stack*. Perlu diingat bahwa *stack* membesar dari alamat tinggi (*high address*) ke arah alamat rendah (*low address*). Dengan demikian, memasukkan elemen baru ke dalam *stack* akan mengurangi nilai alamat yang tersimpan pada `ESP` sedangkan mengeluarkan elemen dari dalam *stack* akan menambah `ESP`.
- `EBP` adalah *register* 32-bit yang digunakan sebagai *base pointer*. Dengan demikian, `EBP` akan berisi alamat dari *current activation frame* pada *stack*.
- `EIP` adalah register 32-bit yang digunakan sebagai *instruction pointer*. Dengan demikian, `EIP` akan berisi alamat dari instruksi selanjutnya yang akan dieksekusi.

Instruksi-instruksi yang digunakan pada Intel® x86 tidak akan dijelaskan di dalam modul praktikum ini. Praktikan dapat mempelajari lebih jauh mengenai instruksi-instruksi ini pada bab 3 di buku "Computer System – A Programmer's Perspective" yang ditulis oleh Bryant dan O'Hallaron.

Stack dan Procedure Call



Stack pada umumnya disusun atas beberapa *activation frame*. Setiap *frame* memiliki sebuah *base pointer* yang menunjukkan alamat tertinggi (*highest address*) pada *frame* tersebut. Karena *stack* tumbuh dari *high address* menuju *low address*, *base pointer* akan menunjukkan alamat tertinggi *frame* tersebut.

variabel lokal caller
...
variabel lokal caller
old base pointer (dari frame sebelumnya)
##awal frame caller

Ketika suatu program (*caller*) memanggil sebuah prosedur (*callee*), *caller* akan memasukkan argumen-argumen untuk memanggil *callee* dari argumen terakhir hingga argumen paling awal secara berurutan ke dalam *stack*. Selanjutnya, *caller* akan memasukkan *return address* ke dalam *stack*. Kemudian, *callee* memasukkan alamat *old base pointer* milik *caller* ke dalam *stack* dan memperbarui nilai *base pointer* yang sesuai dengan *frame callee* (nilai *base pointer* yang baru sama dengan nilai *stack pointer* setelah *old base pointer* disimpan ke dalam *stack*). Kemudian *callee* melakukan alokasi terhadap variabel lokal dan melakukan komputasi sesuai dengan fungsi *callee* tersebut.

Ketika *callee* selesai dieksekusi, *callee* akan menyimpan *return value* pada register EAX. Kemudian, *callee* akan membersihkan *frame*nya sendiri dengan mengganti alamat *base pointer* dengan *old base pointer* yang telah disimpan pada *stack*. Kemudian, *return address* digunakan untuk melanjutkan eksekusi instruksi pada *caller*.

Tugas Pendahuluan

1. Jelaskan perbedaan antara masing-masing pilihan optimisasi dalam GCC (`-O0`, `-O1`, `-O2`, `-O3`, `-Os`, dan `-Ofast`)!
2. Bahasa C merupakan bahasa yang banyak digunakan dalam membuat program pada beberapa *platform*. Sebagai contoh, bahasa C dapat digunakan untuk membuat program pada mikroprosesor berbasis Intel® x86. Bahasa C juga dapat digunakan untuk membuat program pada mikrokontroler AVR®. Di sisi lain, mikroprosesor Intel® x86 memiliki set instruksi yang jauh berbeda dibanding mikrokontroler AVR® ATmega. Menurut pengetahuan Anda tentang proses kompilasi bahasa C, apa yang menyebabkan bahasa C tetap dapat digunakan meskipun *platform*-nya berbeda?
3. Diberikan contoh program sangat sederhana dalam bahasa C sebagai berikut.

```
// Praktikum EL3111 Arsitektur Sistem Komputer
// Modul           :     1
// Percobaan       :     NA
// Tanggal         :     9 September 2014
// Kelompok        :     NA
// Rombongan       :     NA
// Nama (NIM) 1    :     Bagus Hanindhito(13211007)
// Nama (NIM) 2    :     Baharuddin Aziz (13211133)
// Nama File       :     sumofsquare.c
// Deskripsi       :     Demonstrasi procedure call dan stack
//                       Menghitung jumlah dari kuadrat bilangan

int square (int x)
{
    return x*x;
}

int squaresum (int y, int z)
{
```

```
    int temp1 = square(y);
    int temp2 = square(z);
    return temp1+temp2;
}

int main (void)
{
    int a = 5;
    int b = 9;
    return squaresum(a,b);
}
```

Hasil *output compiler* berupa file *assembly* diberikan sebagai berikut.

```
        .file "sumofsquare.c"
        .text
        .globl      _square
        .def  _square;    .scl  2;    .type 32;   .endef
_square:
        pushl %ebp
        movl  %esp, %ebp
        movl  8(%ebp), %eax
        imull 8(%ebp), %eax
        popl  %ebp
        ret
        .globl      _squaresum
        .def  _squaresum; .scl  2;    .type 32;   .endef
_squaresum:
        pushl %ebp
        movl  %esp, %ebp
        subl  $20, %esp
        movl  8(%ebp), %eax
        movl  %eax, (%esp)
        call  _square
        movl  %eax, -4(%ebp)
        movl  12(%ebp), %eax
        movl  %eax, (%esp)
        call  _square
        movl  %eax, -8(%ebp)
        movl  -8(%ebp), %eax
        movl  -4(%ebp), %edx
        addl  %edx, %eax
        leave
        ret
        .def  ___main;    .scl  2;    .type 32;   .endef
        .globl      _main
        .def  _main;      .scl  2;    .type 32;   .endef
_main:
        pushl %ebp
        movl  %esp, %ebp
        andl  $-16, %esp
        subl  $32, %esp
        call  ___main
        movl  $5, 28(%esp)
        movl  $9, 24(%esp)
        movl  24(%esp), %eax
        movl  %eax, 4(%esp)
        movl  28(%esp), %eax
        movl  %eax, (%esp)
```

```
        call   _squaresum
        leave
        ret
        .ident      "GCC: (tdm-2) 4.8.1"
```

a. Pada file *assembly* tersebut, terdapat barisan kode *assembly* (ditampilkan di samping) yang selalu dieksekusi di awal sebuah prosedur. Apa fungsi kode-kode *assembly* tersebut?

```
pushl %ebp
movl  %esp, %ebp
```

b. Gambarkan isi *stack* sebelum instruksi (`imull 8(%ebp),%eax`) pada saat prosedur `square` dipanggil pertama kali oleh prosedur `squaresum`! (isi *stack* saat `main` memanggil `squaresum` tidak perlu digambarkan)

c. Prosedur rekursif merupakan prosedur yang memanggil dirinya sendiri secara berulang-ulang hingga kondisi berhenti dipenuhi. Berdasarkan pengetahuan Anda tentang *procedure call* dan *stack* ini, bagaimanakah penggunaan *memory* pada prosedur rekursif?

Metodologi Praktikum

Sebelum melakukan praktikum, buatlah folder kerja sesuai dengan Nama, NIM, tanggal, dan modul praktikum (lihat petunjuk teknis pelaksanaan praktikum). Kerjakan masing-masing tugas di dalam folder kerja yang sesuai dengan nomor tugas tersebut. Kumpulkan tugas pendahuluan terlebih dahulu sebelum melaksanakan praktikum.

Tugas 1 : Proses Kompilasi Bahasa C Menggunakan GCC

1. Buatlah program dengan kode sebagai berikut menggunakan teks editor Notepad++. Sesuaikan identitas pada *header*.

```
// Praktikum EL3111 Arsitektur Sistem Komputer
// Modul           :     1
// Percobaan       :     NA
// Tanggal         :     9 September 2014
// Kelompok        :     NA
// Rombongan       :     NA
// Nama (NIM) 1    :     Bagus Hanindhito(13211007)
// Nama (NIM) 2    :     Baharuddin Aziz (13211133)
// Nama File       :     code.c
// Deskripsi       :     Demonstrasi proses kompilasi C
//                       Menjumlahkan deret bilangan sebanyak N_LOOP

#define N_LOOP 500
int main(void)
{
      int indeks;
      int accumulator;
      indeks = 0;
      accumulator = 0;
      while(indeks<N_LOOP)
      {
            accumulator = accumulator + indeks;
            indeks = indeks + 1;
      }
    return accumulator;
}
```

2. Simpan kode program tersebut dengan nama `code.c`.
3. *Preprocess* kode program tersebut menjadi `code.i` lalu lihat hasilnya dengan teks editor

Notepad++. Pastikan direktori kerja *command prompt* berada di direktori tugas 1.

```
gcc -E code.c > code.i
```

4. *Compile* kode program tersebut menjadi `code.s` lalu lihat hasilnya dengan teks editor Notepad++. Pastikan direktori kerja *command prompt* berada di direktori tugas 1.

```
gcc -S code.c
```

5. *Assemble* kode program tersebut menjadi `code.o` lalu lihat hasilnya dengan heksadesimal editor HexEdit. Pastikan direktori kerja *command prompt* berada di direktori tugas 1.

```
gcc -c code.c
```

6. Lakukan semua proses kompilasi terhadap kode program tersebut menjadi file *executable* `code.exe` lalu lihat hasilnya dengan heksadesimal editor HexEdit. Pastikan direktori kerja *command prompt* berada di direktori tugas 1.

```
gcc -o code.exe code.c
```

7. Buka file `code.o` dan `code.exe` dengan teks editor Notepad++. Bagaimana hasilnya?

Tugas 2 : Proses Kompilasi Bahasa C Menggunakan GCC dengan Bantuan Batch File

1. Salin (*copy*) file `code.c` dari folder tugas 1 ke dalam folder tugas 2.
2. Buatlah batch file dengan kode sebagai berikut menggunakan teks editor Notepad++.

```
%~d0
cd "%~dp0"
gcc -E code.c > code.i
gcc -S code.c
gcc -c code.c
gcc -o code.exe code.c
code.exe
pause
```

3. Simpan kode program tersebut dengan nama `batch.bat` pada folder yang sama dengan file `code.c`.
4. Klik dua kali pada file `batch.bat` sehingga muncul jendela *command prompt*.
5. Lihat semua file yang dihasilkan pada folder yang sama dengan file `code.c`.

Tugas 3 : *Disassembly* File Objek

1. Salin (*copy*) file `code.o` dan `code.exe` dari folder tugas 2 ke dalam folder tugas 3.
2. Gunakan `objdump` untuk melakukan *disassembly* file `code.o`.

```
objdump -d code.o > disassembly_code_o.asm
```

3. Gunakan `objdump` untuk melakukan *disassembly* file `code.exe`.

```
objdump -d code.exe > disassembly_code_exe.asm
```

4. Buka file `disassembly_code_o.asm` dan `disassembly_code_exe.asm` dengan teks editor Notepad++ dan bandingkan kedua file tersebut.

Tugas 4 : Optimisasi Kompilasi Program pada GCC

1. Salin (*copy*) file `code.c` dari folder tugas 2 ke dalam folder tugas 4.
2. Ubah nama file `code.c` tersebut menjadi `code_O0.c`

3. Salin file `code_O0.c` menjadi empat file baru dengan isi yang sama masing-masing memiliki nama `code_O1.c`, `code_O2.c`, `code_O3.c`, `code_Os.c`, dan `code_Ofast.c`.
4. Buatlah file `batch.bat` untuk melakukan kompilasi kelima file kode tersebut dengan pilihan optimisasi yang berbeda lalu secara otomatis melakukan proses *disassembly* kelima file objek yang dihasilkan.

```
%~d0
cd "%~dp0"
gcc -O0 -c code_O0.c
gcc -O1 -c code_O1.c
gcc -O2 -c code_O2.c
gcc -O3 -c code_O3.c
gcc -Os -c code_Os.c
gcc -Ofast -c code_Ofast.c
objdump -d code_O0.o > code_O0.s
objdump -d code_O1.o > code_O1.s
objdump -d code_O2.o > code_O2.s
objdump -d code_O3.o > code_O3.s
objdump -d code_Os.o > code_Os.s
objdump -d code_Ofast.o > code_Ofast.s
pause
```

5. Buka file `code_O0.s`, `code_O1.s`, `code_O2.s`, `code_O3.s`, `code_Os.s`, dan `code_Ofast.s` dengan teks editor lalu bandingkan kelimanya.

Tugas 5 : Kompilasi Beberapa File Kode dengan GCC

1. Buatlah file bernama `main_text.c` menggunakan teks editor Notepad++ dengan isi sebagai berikut. Sesuaikan identitas pada *header*.

```
// Praktikum EL3111 Arsitektur Sistem Komputer
// Modul           :     1
// Percobaan       :     NA
// Tanggal         :     9 September 2014
// Kelompok        :     NA
// Rombongan       :     NA
// Nama (NIM) 1    :     Bagus Hanindhito(13211007)
// Nama (NIM) 2    :     Baharuddin Aziz (13211133)
// Nama File       :     main_text.c
// Deskripsi       :     Demonstrasi MakeFile
//                       Memanggil prosedur test pada text.c
#include "text.h"
void main(void)
{
    test();
}
```

2. Buatlah file bernama `text.c` menggunakan teks editor Notepad++ dengan isi sebagai berikut. Sesuaikan identitas pada *header*.

```
// Praktikum EL3111 Arsitektur Sistem Komputer
// Modul           :     1
// Percobaan       :     NA
// Tanggal         :     9 September 2014
// Kelompok        :     NA
// Rombongan       :     NA
// Nama (NIM) 1    :     Bagus Hanindhito(13211007)
// Nama (NIM) 2    :     Baharuddin Aziz (13211133)
// Nama File       :     text.c
```

```
// Deskripsi      :     Demonstrasi MakeFile, Mencetak string ke layar
#include <stdio.h>
#include "text.h"
void test(void)
{
      printf("Arsitektur Sistem Komputer sangat menyenangkan!\n");
}
```

3. Buatlah file bernama `text.h` menggunakan teks editor Notepad++ dengan isi sebagai berikut. Sesuaikan identitas pada *header*.

```
// Praktikum EL3111 Arsitektur Sistem Komputer
// Modul          :     1
// Percobaan      :     NA
// Tanggal        :     9 September 2014
// Kelompok       :     NA
// Rombongan      :     NA
// Nama (NIM) 1   :     Bagus Hanindhito(13211007)
// Nama (NIM) 2   :     Baharuddin Aziz (13211133)
// Nama File      :     text.h
// Deskripsi      :     Demonstrasi MakeFile, Mencetak string ke layar

#ifndef TES_H
  #define TES_H 100
  void test(void);
#endif
```

4. Lakukan kompilasi file tersebut dengan *command prompt* untuk menghasilkan `main_text.exe`. Pastikan direktori kerja *command prompt* berada di direktori tugas 5.

```
gcc -o main_text.exe text.c main_text.c
```

5. Jalankan program `main_text.exe` untuk melihat hasilnya.

Tugas 6 : Penggunaan Makefile pada GCC

1. Salin (*copy*) file `main_text.c`, `text.c`, dan `text.h` dari folder tugas 5 ke folder tugas 6.
2. Buatlah makefile menggunakan teks editor Notepad++ dengan isi sebagai berikut. Harap berhati-hati dengan spasi karena dapat menyebabkan kompilasi mengalami kegagalan.

```
all: main_text.exe

main_text.exe: main_text.o text.o
      gcc main_text.o text.o -o main_text.exe

main_text.o: main_text.c
      gcc -c main_text.c

text.o: text.c
      gcc -c text.c
```

3. Simpan makefile tersebut dengan nama `makefile` (tanpa ekstensi / akhiran).
4. Gunakan *command prompt* untuk melakukan kompilasi dengan sintaks berikut. Pastikan direktori kerja *command prompt* berada di direktori tugas 6.

```
mingw32-make -f makefile
```

5. Bandingkan hasil tugas 5 dengan tugas 6 dengan melakukan eksekusi `main_text.exe` dan membandingkan hasil eksekusinya.

Tugas 7 : Header File

1. Pada tugas ini, Anda akan belajar untuk membuat dan memanfaatkan *header* file dan memahami bagaimana `extern` pada *header* file bekerja. Anda dapat menggunakan makefile atau batch file untuk melakukan kompilasi tugas ini. Namun, Anda disarankan untuk menggunakan makefile agar Anda dapat berlatih dalam membuat makefile.
2. Buatlah program dengan kode sebagai berikut menggunakan teks editor Notepad++.

```
// Praktikum EL3111 Arsitektur Sistem Komputer
// Modul           :     1
// Percobaan       :     NA
// Tanggal         :     9 September 2014
// Kelompok        :     NA
// Rombongan       :     NA
// Nama (NIM) 1    :     Bagus Hanindhito(13211007)
// Nama (NIM) 2    :     Baharuddin Aziz (13211133)
// Nama File       :     add.c
// Deskripsi       :     Demonstrasi header file
//                       Menjumlahkan dua bilangan

#define START_VAL 0

int accum = START_VAL;
int sum(int x, int y)
{
    int t = x + y;
    accum += t;
    return t;
}
```

3. Simpan kode program tersebut dengan nama `add.c`.
4. Buatlah sebuah program sederhana `main` dalam file `main.c` yang meminta pengguna memasukkan dua buah bilangan lalu menjumlahkan kedua bilangan tersebut dengan memanggil fungsi `sum` pada `add.c`.
5. Buatlah *header* file `add.h` yang akan di-*include* pada `main.c`. Gunakan `extern` agar variabel global `accum` pada `add.c` dapat dikenali dan diakses oleh `main` pada `main.c`.
6. Lakukan kompilasi file `add.c` dan `main.c` sehingga dihasilkan program `main.exe`.

```
gcc -o main.exe add.c main.c
```

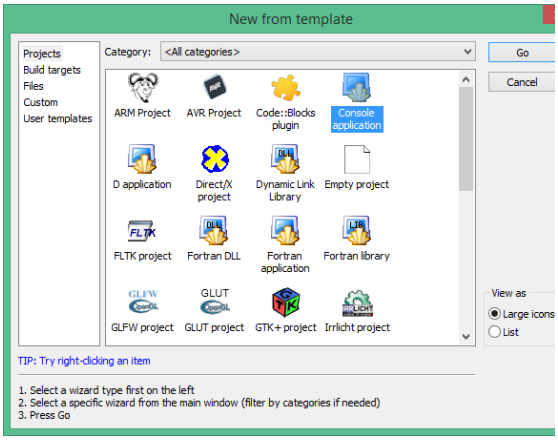
Tugas 8 : Pemanggilan Prosedur dan Stack Memory

1. Buka program CodeBlocks untuk membuat *project* baru. Pilih *Console Application* lalu klik Go sehingga *wizard Console Application* terbuka.
2. Klik *next* lalu pilih bahasa pemrograman dengan bahasa C lalu klik *next*.
3. Isi dengan *Project Title*, folder *project*, dan nama *project* lalu klik *next*.
4. Buka file `main.c` yang telah dihasilkan secara otomatis pada bagian *workspace*.
5. Hapus semua kode pada file `main.c` lalu salin semua kode pada tugas pendahuluan (file `sumofsquare.c`). Simpan file `main.c` dengan perintah *save*.
6. Pilih menu *Settings* lalu klik *Compiler*. Beri *check* pada pilihan *Produce debugging symbols [-g]* lalu klik OK.

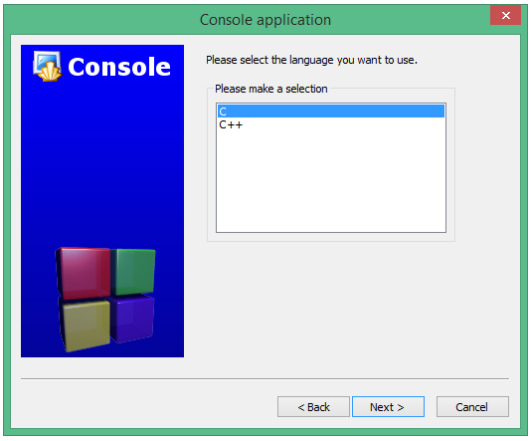
7. Klik kanan pada baris program lalu pilih *Add Breakpoint*. Beri semua *breakpoint* pada setiap baris program.
8. Pilih menu *Debug* lalu klik *Start / Continue*.
9. Pilih menu *Debug* lalu pilih *Debugging windows*. Buka *CPU Registers*, *Call stack*, *Disassembly*, *Memory dump*, dan *Watches*. Susun semua jendela yang terbuka sedemikan rupa agar tidak menghalangi *source code* program utama.
10. Klik pada jedela *source code* lalu tekan tombol F7 pada *keyboard*. Amati perubahan pada semua *Debugging windows*. Perhatikan juga tanda panah kuning pada baris kode. Tanda panah kuning ini merupakan posisi eksekusi program oleh *processor*.
11. Saat anak panah kuning berada pada kode `return squaresum(a,b);`, lihat isi *register* `ESP` dan `EBP` pada *CPU Registers*. Isikan nilai *register* `ESP` pada *address* di jendela *Memory*. Pilih Byte 128 atau yang lebih tinggi. Perhatikan pula jendela *Call stack*.
12. Catat nilai *memory* di antara *address* yang ditunjukkan oleh `EBP` dan `ESP`. Bisakah Anda merekonstruksi bentuk *stack* pada kondisi tersebut?
13. Klik jendela *source code*, lalu tekan tombol F7 pada *keyboard* untuk melanjutkan eksekusi program hingga berhenti pada kode `int temp1 = square(y);`. Perhatikan jendela *Call stack*. Perhatikan pula jendela *Memory*. Isikan nilai *register* `ESP` pada *address* di jendela *Memory*. Pilih Byte 128 atau yang lebih tinggi.
14. Catat nilai *memory* di antara *address* yang ditunjukkan oleh `EBP` dan `ESP`. Bisakah Anda merekonstruksi bentuk *stack* pada kondisi tersebut?
15. Lanjutan eksekusi program dengan menekan tombol F7 pada *keyboard* hingga berhenti pada kode `return x*x;`. Perhatikan jendela *Call stack*. Perhatikan pula jendela *Memory*. Isikan nilai *register* `ESP` pada *address* di jendela *Memory*. Pilih Byte 128 atau yang lebih tinggi.
16. Catat nilai *memory* di antara *address* yang ditunjukkan oleh `EBP` dan `ESP`. Bisakah Anda merekonstruksi bentuk *stack* pada kondisi tersebut?
17. Lanjutkan eksekusi program tersebut dengan menekan tombol F7 pada *keyboard* hingga berhenti pada kode `int temp2 = square(z);`. Perhatikan jendela *Call stack*. Perhatikan pula jendela *Memory*. Isikan nilai *register* `ESP` pada *address* di jendela *Memory*. Pilih Byte 128 atau yang lebih tinggi.
18. Catat nilai *memory* di antara *address* yang ditunjukkan oleh `EBP` dan `ESP`. Bisakah Anda merekonstruksi bentuk *stack* pada kondisi tersebut?
19. Lanjutan eksekusi program dengan menekan tombol F7 pada *keyboard* hingga berhenti pada kode `return x*x;`. Perhatikan jendela *Call stack*. Perhatikan pula jendela *Memory*. Isikan nilai *register* `ESP` pada *address* di jendela *Memory*. Pilih Byte 128 atau yang lebih tinggi.
20. Catat nilai *memory* di antara *address* yang ditunjukkan oleh `EBP` dan `ESP`. Bisakah Anda merekonstruksi bentuk *stack* pada kondisi tersebut?
21. Lanjutkan eksekusi program tersebut dengan menekan tombol F7 pada *keyboard* hingga berhenti pada kode `return temp1+temp2;`. Perhatikan jendela *Call stack*. Perhatikan pula jendela *Memory*. Isikan nilai *register* `ESP` pada *address* di jendela *Memory*. Pilih Byte 128 atau yang lebih tinggi.
22. Catat nilai *memory* di antara *address* yang ditunjukkan oleh `EBP` dan `ESP`. Bisakah Anda

merekonstruksi bentuk *stack* pada kondisi tersebut?

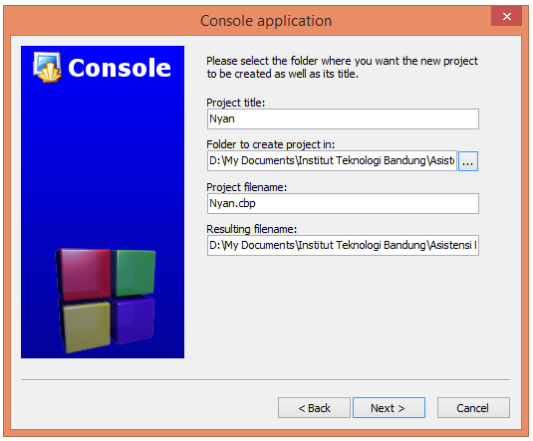
23. Lanjutkan eksekusi program tersebut dengan menekan tombol F7 pada *keyboard* hingga program selesai dieksekusi. Perhatikan jendela *Call stack*. Perhatikan pula jendela *Memory*. Isikan nilai *register* `ESP` pada *address* di jendela *Memory*. Pilih Byte 128 atau yang lebih tinggi.
24. Catat nilai *memory* di antara *address* yang ditunjukkan oleh `EBP` dan `ESP`. Bisakah Anda merekonstruksi bentuk *stack* pada kondisi tersebut?
25. Bandingkan isi *stack* untuk nilai *memory* pada setiap *state* yang Anda catat.



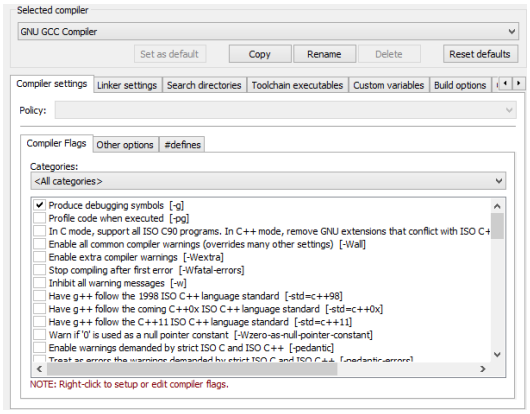
Langkah (1)



Langkah (2)

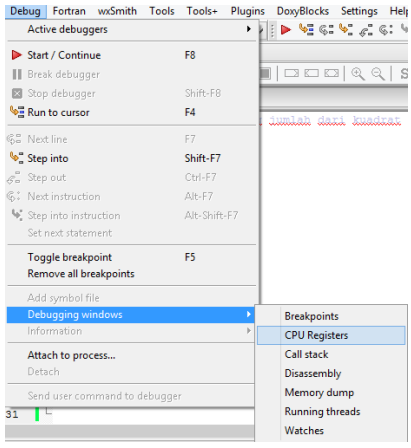


Langkah (3)

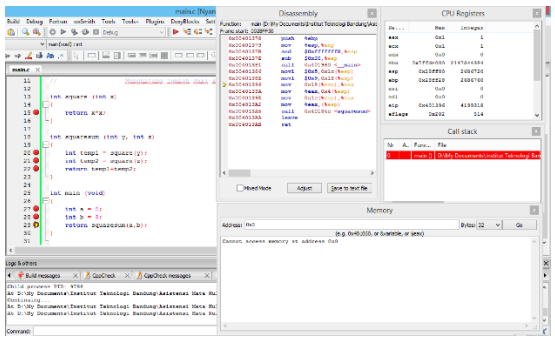


Langkah (6)

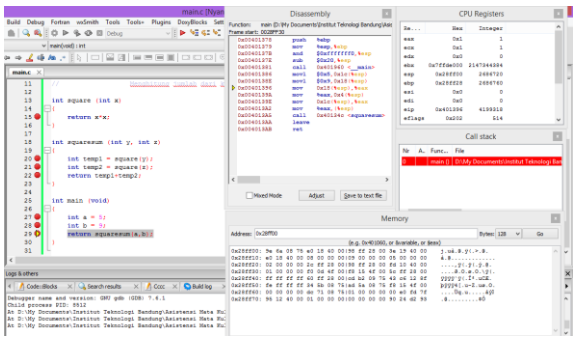
Langkah (7)



Langkah (8)



Langkah (9)



Langkah (10)

Tugas 9 : Program Fibonacci

Buatlah program penghitung deret fibonacci yang sederhana yaitu program yang akan menampilkan deret fibonacci dengan banyak angka (`n`) yang sesuai dengan masukkan pengguna. Contohnya, bila pengguna menginginkan banyak angka sebesar `n=7`, maka deret fibonacci memiliki bentuk seperti berikut.

```
1, 1, 2, 3, 5, 8, 13
```

Program ini terdiri atas tiga buah file kode. File `inputn.c` berisi kode-kode yang menangani proses meminta pengguna untuk memasukkan nilai `n` hingga menerima dan menyimpan nilai `n` yang dimasukkan oleh pengguna. Perhatikan bahwa nilai `n` memiliki batas minimum sebesar `n=2` sehingga diperlukan validasi *input*. File `fibo.c` berisi kode-kode yang menangani proses kalkulasi deret fibonnaci sesuai jumlah angka yang akan ditampilkan dan menampilkannya pada layar. File `fibo_main.c` yang berisi program utama yang memanggil prosedur-prosedur pada `inputn.c` dan `fibo.c`.

1. Implementasi prosedur-prosedur yang dibutuhkan pada `inputn.c` dan `fibo_main.c`. Jangan lupa membuat file *header* untuk masing-masing file kode tersebut.
2. Buatlah program utama pada file `fibo_main.c`.
3. Buatlah makefile untuk melakukan kompilasi seluruh file kode program.
4. Jalankan program yang telah dikompilasi lalu periksalah fungsionalitasnya.

Hasil dan Analisis

Berikut ini adalah pertanyaan-pertanyaan yang dapat membantu analisis pada laporan praktikum. Perhatikan bahwa analisis tidak hanya terbatas pada pertanyaan-pertanyaan di bawah ini.

1. Bagaimana proses sebuah kode file bahasa C diolah menjadi program *executable*?
2. Bagaimana pengaruh arsitektur mikroprosesor terhadap *compiler* bahasa C?
3. Mengapa untuk membuka file objek, baik file `.o` maupun `.exe`, tidak dapat dilakukan dengan teks editor biasa?
4. Mengapa hasil *disassembly* file `code.o` dan file `code.exe` menghasilkan file *assembly* yang berbeda? Mengapa file hasil *disassembly* `code.exe` memiliki isi lebih banyak dibanding file hasil disassembly `code.o`?
5. Bagaimana pengaruh pilihan optimisasi pada GCC saat kompilasi? Adakah cara untuk mengukur perbedaan kecepatan eksekusi dari program yang dihasilkan dengan pilihan optimisasi yang berbeda?

6. Apa perbedaan antara file makefile dan file batch? Jenis file yang manakah yang paling membantu dalam praktikum ini?
7. Apa fungsi dari file *header*?
8. Apa fungsi dari variabel `extern`?
9. Mengapa dalam *procedure call*, *memory* dimodelkan dengan *stack*?
10. Bagaimana susunan *stack* terhadap pemanggilan *procedure*?
11. Apa saja tanggung jawab *caller* dalam setiap pemanggilan *procedure*?
12. Apa saja tanggung jawab *callee* dalam setiap pemanggilan *procedure*?

Simpulan

Buatlah simpulan dari percobaan yang Anda lakukan dalam bentuk poin. Simpulan hendaknya menjawab tujuan praktikum ini.

Daftar Pustaka

Bryant, Randal, dan David O'Hallaron. *Computer Systems : A Programmer's Perspective 2nd Edition*. 2011. Massachusetts : Pearson Education Inc.

Patterson, David, dan John Hennessy. *Computer Organization and Design* : *The Hardware/Software Interface*. 2012. Waltham : Elsevier Inc.

PERCOBAAN II

POINTER, STRUCTURE, ARRAY, DAN OPERASI DALAM LEVEL BIT

Tujuan Praktikum

- Praktikan memahami representasi informasi dalam level bit yang disimpan pada *memory*.
- Praktikan mampu menggunakan operator-operator *bitwise* dalam bahasa C untuk mengolah informasi yang tersimpan dalam *memory*.
- Praktikan memahami fungsi *pointer* dan mampu menggunakan *pointer* untuk melakukan pengolahan data di dalam *memory*.
- Praktikan memahami *array* beserta representasinya dalam *memory* dan pengolahan informasinya dalam bahasa C.
- Praktikan memahami *structure* beserta representasinya dalam *memory* dan pengolahannya dalam bahasa C.

Perangkat Praktikum

- Komputer Desktop / Laptop dengan sistem operasi Microsoft® Windows™ 7/8/8.1
- *Compiler* GCC dalam paket program CodeBlocks untuk melakukan kompilasi program (program dan cara instalasi dapat dilihat pada **http://el3111.bagus.my.id**).
- Notepad++ sebagai teks editor.

Landasan Teoretis Praktikum

Tipe Data

Tipe data merupakan representasi data yang disimpan dalam *memory*. Tipe data menentukan operasi yang dapat dilakukan pada suatu data bertipe tertentu, rentang nilai yang mungkin dimiliki oleh data bertipe tertentu, arti dari data, dan cara menyimpan data tersebut dalam *memory*. Terdapat beberapa tipe data yang telah tersedia dalam bahasa C yang disebut *simple data type*.

Masing-masing tipe data memiliki ukuran yang berbeda-beda untuk disimpan di dalam *memory*. Dalam bahasa C, kita dapat menggunakan sintaks `sizeof` untuk mengetahui besar tipe data tersebut di dalam *memory*. Contohnya, untuk mengetahui ukuran tipe data integer, kita cukup menggunakan perintah `sizeof(int)` untuk mengetahui ukurannya. Kita juga dapat melihat rentang nilai yang direpresentasikan oleh masing-masing tipe data.

Tipe Data	Ukuran	Rentang Nilai
Char	1 Byte	-128 – 127
Unsigned Char	1 Byte	0 – 255
Signed Char	1 Byte	-128 – 127
Short	2 Byte	-32768 – 32767
Unsigned Short	2 Byte	0 – 65535
Signed Short	2 Byte	-32768 – 32767
Int	4 Byte	-2147483648 – 2147483647
Unsigned Int	4 Byte	0 – 4294967295
Signed Int	4 Byte	-2147483648 – 2147483647
Long	4 Byte	-2147483648 – 2147483647
Unsigned Long	4 Byte	0 – 4294967295

Signed Long	4 Byte	-2147483648 – 2147483647
Long Long	8 Byte	-9223372036854775808 – 9223372036854775807
Unsigned Long Long	8 Byte	0 – 18446744073709551615
Signed Long Long	8 Byte	9223372036854775808 – 9223372036854775807
Float	4 Byte	?
Double	8 Byte	?

Data pada tabel berikut dapat diketahui dengan program sederhana sebagai berikut. Perhatikan bahwa mesin berbeda dapat memberikan hasil yang berbeda berkaitan dengan besar tipe data.

```
// Praktikum EL3111 Arsitektur Sistem Komputer
// Modul           :     NA
// Percobaan       :     NA
// Tanggal         :     9 September 2014
// Kelompok        :     NA
// Rombongan       :     NA
// Nama (NIM) 1    :     Bagus Hanindhito(13211007)
// Nama (NIM) 2    :     Baharuddin Aziz (13211133)
// Nama File       :     sizeof.c
// Deskripsi       :     Mencari ukuran tipe data dan rentang nilai
#include <stdio.h>
#include <limits.h>
int main(void)
{
   int zero = 0;
   printf("Char Byte Size: %d\n", sizeof(char));
   printf("Unsigned Char Byte Size: %d\n", sizeof(unsigned char));
   printf("Signed Char Byte Size: %d\n", sizeof(signed char));
   printf("Short Byte Size: %d\n", sizeof(short));
   printf("Unsigned Short Byte Size: %d\n", sizeof(unsigned short));
   printf("Signed Short Byte Size: %d\n", sizeof(signed short));
   printf("Int Byte Size: %d\n", sizeof(int));
   printf("Unsigned Int Byte Size: %d\n", sizeof(unsigned int));
   printf("Signed Int Byte Size: %d\n", sizeof(signed int));
   printf("Long Byte Size: %d\n", sizeof(long));
   printf("Unsigned Long Byte Size: %d\n", sizeof(unsigned long));
   printf("Signed Long Byte Size: %d\n", sizeof(signed long));
   printf("Long Long Byte Size: %d\n", sizeof(long long));
   printf("Unsigned Long Long Byte Size: %d\n", sizeof(unsigned long long));
   printf("Signed Long Long Byte Size: %d\n", sizeof(signed long long));
   printf("Float Byte Size: %d\n", sizeof(float));
   printf("Double Byte Size: %d\n", sizeof(double));
   printf("Char Byte Range: %d - %d\n", CHAR_MIN, CHAR_MAX);
   printf("Unsigned Char Byte Size: %d - %d\n", zero, UCHAR_MAX);
   printf("Signed Char Byte Size: %d - %d\n", SCHAR_MIN, SCHAR_MAX);
   printf("Short Byte Size: %d - %d\n", SHRT_MIN, SHRT_MAX);
   printf("Unsigned Short Byte Size: %d - %d\n", zero, USHRT_MAX);
   printf("Signed Short Byte Size: %d - %d\n", SHRT_MIN, SHRT_MAX);
   printf("Int Byte Size: %d - %d\n", INT_MIN, INT_MAX);
   printf("Unsigned Int Byte Size: %d - %d\n", zero, UINT_MAX);
   printf("Signed Int Byte Size: %d - %d\n", INT_MIN, INT_MAX);
   printf("Long Byte Size: %d - %d\n", LONG_MIN, LONG_MAX);
   printf("Unsigned Long Byte Size: %d - %d\n", zero, ULONG_MAX);
   printf("Signed Long Byte Size: %d - %d\n", LONG_MIN, LONG_MAX);
   printf("Long Long Byte Size: %d - %d\n", LLONG_MIN, LLONG_MAX);
   printf("Unsigned Long Long Byte Size: %d - %d\n", zero, ULLONG_MAX);
   printf("Signed Long Long Byte Size: %d - %d\n", LLONG_MIN, LLONG_MAX);
   return 0;
}
```

Dengan adanya perbedaan ukuran masing-masing tipe data, diperlukan sebuah mekanisme *alignment* pada *memory* agar setiap data tersusun dengan baik di dalam *memory* dan dapat diproses oleh mikroprosesor. Dengan *alignment*, data-data variabel disimpan dalam lokasi *memory* yang memiliki *address offset* yang berupa kelipatan dari ukuran *word*. Hal ini akan menambah *performance* karena data disusun sesuai cara mikroprosesor menggunakan *memory*.

Operator *Bitwise* dalam Bahasa C

Bahasa C mendukung pengolahan informasi dalam level bit menggunakan operator *bitwise*. Berbeda dengan operator level byte, operator *bitwise* akan mengoperasikan data untuk setiap bit. Sedangkan operator level byte, data akan diolah dalam bentuk 1 byte (1 byte = 8 bit). Operator *bitwise* dapat digunakan pada berbagai tipe data seperti `char`, `int`, `short`, `long`, atau `unsigned`. Operator-operator *bitwise* dalam bahasa C didefinisikan sebagai berikut.

Simbol	Operator
&	Bitwise AND
|	Bitwise inclusive OR
^	Bitwise exclusive OR
<<	Left Shift
>>	Right Shift
~	Bitwise NOT (one's complement) (unary)

Bahasa C juga memiliki operator logika AND, inclusive OR, dan NOT. Operator ini sering tertukar dengan operator *bitwise*. Operator logika tersebut diberikan sebagai berikut. Pada operasi logika, setiap argumen bukan nol merepresentasikan TRUE, sedangkan argumen nol merepresentasikan FALSE. Ekspresi logika akan mengembalikan nilai 1 untuk TRUE dan nilai 0 untuk FALSE.

Simbol	Operator
&&	Logical AND
||	Logical OR
!	Logical NOT

Khusus untuk operator *Right Shift*, terdapat dua jenis operator *Right Shift*, yaitu *Logical Right Shift* dan *Arithmetic Right Shift*. *Logical Right Shift* akan mengisi bit MSB dengan nilai 0 *sementara Arithmetic Right Shift* akan mengisi bit MSB sesuai dengan tanda (*sign*) variabel tersebut menurut aturan *two's complement*. Untuk *Left Shift*, tidak ada perbedaan antara *Logical Left Shift* dan *Arithmetic Left Shift*. Pada umumnya, seluruh mesin dapat mendukung dua jenis operator *Right Shift* dan *Left Shift* ini.

Structure

Structure (`struct`) merupakan *complex data type* yang mendefinisikan daftar variabel yang akan ditempatkan dalam blok *memory* menggunakan satu nama. Dengan demikian, setiap variabel berbeda pada *structure* dapat diakses menggunakan sebuah *single pointer* atau dengan menggunakan nama *structure* itu sendiri. Pada *structure*, masing-masing variabel disimpan dalam blok *memory* yang kontigu yang biasanya memiliki *delimiter* berupa panjang *word*. Kita juga dapat menggunakan sintaks **`sizeof`** untuk memeriksa ukuran *structure* yang kita definisikan. Perlu diingat bahwa bahasa C tidak mengizinkan kita melakukan deklarasi rekursif terhadap *structure* (sebuah *structure* tidak boleh berisi *structure* bertipe yang sama dengan *structure* tersebut). Kita dapat menggunakan *pointer* untuk melakukannya. Beberapa mesin juga membutuhkan *alignment* data pada *memory* secara spesifik

sehingga ukuran *structure* dapat berbeda karena *compiler* secara otomatis melakukan *alignment* data-data pada *structure*, contohnya dengan *padding*.

Array

Array merupakan kumpulan lokasi penyimpanan data yang kontigu (berurutan) dengan tipe data yang sama. Setiap lokasi penyimpanan dalam sebuah *array* disebut elemen *array*. *Array* dialokasikan secara sekaligus dalam *memory* sesuai dengan ukurannya. Karena letak elemen yang berurutan, akses elemen *array* pada memory relatif lebih mudah dan cepat dibandingkan dengan struktur data *Linked-List*. Setiap elemen dalam *array* dapat diakses menggunakan indeks yang biasanya berupa bilangan bulat skalar bukan negatif. Dalam bahasa C, elemen pertama dalam *array* diberi indeks 0. Representasi *array* dalam *memory* dengan deklarasi `int nim[8]` adalah sebagai berikut (asumsikan *address* elemen ke-0 adalah `0x4000` dengan nilai elemen pertama adalah 1, nilai elemen kedua adalah 3, nilai elemen ketiga adalah 2, nilai elemen keempat adalah 1, nilai elemen kelima adalah 1, nilai elemen keenam adalah 0, nilai elemen ketujuh adalah 0, dan nilai elemen kedelapan adalah 7).

Nilai	1	3	2	1	1	0	0	7
Alamat	`0x4000`	`0x4004`	`0x4008`	`0x400C`	`0x4010`	`0x4014`	`0x4018`	`0x401C`
Indeks	`nim[0]`	`nim[1]`	`nim[2]`	`nim[3]`	`nim[4]`	`nim[5]`	`nim[6]`	`nim[7]`

Bahasa C mendukung deklarasi *array* secara statis maupun secara dinamis. *Array* statis memiliki ukuran yang tidak bisa diubah-ubah sedangkan *array* dinamis memiliki ukuran yang dapat ditentukan saat program sedang berjalan. *Array* dinamis dapat dideklarasikan dengan contoh `int pusheen[]`. Dengan demikian `pusheen` merupakan *array* yang memiliki elemen bertipe *integer* namun dengan banyak elemen yang belum didefinisikan. Untuk melakukan alokasi terhadap *array* `pusheen`, kita dapat menggunakan perintah `malloc` atau `calloc` dalam bahasa C. Untuk mengubah ukuran *array* dinamis yang telah dialokasikan, kita dapat menggunakan perintah `realloc`. Untuk melakukan dealokasi *array* dinamis, kita dapat menggunakan perintah `free`. Perhatikan bahwa pada beberapa kasus, perintah `realloc` dapat menyebabkan program tidak efisien contohnya saat *array* diubah ukurannya menjadi lebih besar dan sistem harus melakukan pemindahan elemen-elemen *array* ke posisi *memory* yang baru agar perbesaran ukuran *array* dapat dilakukan.

Array juga dapat memiliki elemen *array* berupa *array*. Dengan demikian, kita dapat mendefinisikan *array* n-dimensi. Contohnya, sebuah matriks merupakan array dengan dua dimensi. *Array* tersebut memiliki elemen *array* berupa *array*, contohnya `int pusheen[][5]` atau `int pusheen[4][5]`. Namun, karena *memory* komputer bersifat linear, komputer akan menyimpan *array* n-dimensi dalam bentuk linear juga. Hal ini menyebabkan kita harus memperhatikan urutan indeks untuk mengakses setiap elemen *array* n-dimensi karena hal ini akan berpengaruh terhadap *performance* dari program yang kita buat terlebih data array yang diolah cukup besar, contohnya seberapa baikkah program yang kita buat dalam memanfaatkan *cache memory* (*cache-friendly*).

Pointer

Pointer merupakan variabel yang menyimpan alamat *memory*. Dengan kata lain, *pointer* memiliki nilai alamat *memory* untuk melakukan referensi terhadap suatu objek yang tersimpan dalam *memory* komputer. Dengan menggunakan pointer, kita dapat memiliki akses terhadap *memory* secara langsung. Untuk setiap tipe data T, terdapat *pointer* ke T. *Deklarasi pointer* dalam bahasa C dapat dilakukan dengan mudah, contohnya `int *ptr` yang merupakan *pointer* yang melakukan referensi terhadap objek bertipe *integer*.

Pointer juga dapat digunakan untuk menunjukkan *structure* berdasarkan alamatnya di *memory*. Hal ini sangat berguna untuk melakukan *passing structure* ke atau dari sebuah fungsi hanya dengan memberikan alamat *structure* tersebut di *memory*. *Pointer* ini juga dapat di-dereferensi seperti halnya *pointer* lain dalam bahasa C, yaitu menggunakan operator *dereference* (`*`). Selain itu, juga terdapat operator yang sangat berguna yaitu `struct_name -> member` untuk melakukan dereferensi *pointer-to-struct* lalu mengakses nilai dari anggota *structure* tersebut. Operator tersebut memiliki ekuivalensi dengan `(*struct_name).member`. Sebetulnya, sebuah fungsi dapat langsung mengembalikan sebuah *structure* walaupun hal ini terkadang tidak efisien saat dijalankan.

Dalam *array*, indeks biasanya didefinisikan sebagai perhitungan matematika terhadap alamat *pointer*. Dengan demikian penulisan `array[i]` memiliki ekuivalensi dengan `*(array + i)`. Perhitungan matematika terhadap *pointer* untuk mengakses setiap elemen *array* dapat dilakukan karena *array* memiliki elemen yang tersusun secara kontigu (berurutan tanpa jeda).

Tugas Pendahuluan

1. Bagaimana tipe data `float` dan `double` disimpan dalam *memory* komputer? Berapakah rentang (nilai minimum dan nilai maksimum) dari tipe data `float` dan `double` di luar `NaN` dan `Inf`?
2. Seperti yang kita ketahui bahwa terdapat dua jenis operator *right shift*, yaitu *logical right shift* dan *arithmetic right shift*. Kedua jenis operator *right shift* ini memiliki simbol yang sama yaitu >>. Bagaimana caranya kita dapat menentukan apakah operator *right shift* yang digunakan adalah *logical right shift* atau *arithmetic right shift*? Berilah contohnya dalam bentuk sintaks bahasa C!
3. Diberikan dua buah deklarasi *structure* dengan elemen-elemen yang sama sebagai berikut.

```
typedef struct
{
    char kelas;
    short kode_matakuliah;
    int nim;
    char nilai_abjad;
    int nilai_angka;
} daftar_NA_1;

typedef struct
{
    char kelas;
    char nilai_abjad;
    short kode_matakuliah;
    int nim;
    int nilai_angka;
} daftar_NA_2;
```

   a. Berapakah ukuran *structure* `daftar_NA_1` dan `daftar_NA_2` dalam *memory*? Gambarkan pula bagaimana kedua *structure* ini disimpan dalam *memory*!
   b. Mengapa `daftar_NA_1` dan `daftar_NA_2` memiliki ukuran yang berbeda walaupun elemen-elemen penyusun *structure* sama namun berbeda urutan penulisan?
4. Diketahui deklarasi *array* dua dimensi . Bagaimana komputer menyimpan *array* ini di dalam *memory*? Gambarkan bentuk penyimpanan *array* dua dimensi ini di dalam *memory* komputer.
5. Diberikan contoh program sederhana sebagai berikut. Program ini akan digunakan pada saat praktikum sehingga disarankan praktikan telah menyalin *source code* program ini.

```
// Praktikum EL3111 Arsitektur Sistem Komputer
```

```
// Modul            :     2
// Percobaan        :     0
// Tanggal          :     7 November 2013
// Kelompok         :     VI
// Rombongan        :     A
// Nama (NIM) 1     :     Audra Fildza Masita (13211008)
// Nama (NIM) 2     :     Bagus Hanindhito (13211007)
// Nama File        :     printbitbyte.c
// Deskripsi        :     Demonstrasi Pointer
#include "printbitbyte.h"
#include <stdlib.h>
#include <stdio.h>
typedef unsigned char *byte_pointer;
// address  =     alamat dari variabel dalam memory
// size     =     ukuran variabel dalam memory (sizeof)
void printByte(byte_pointer address, int size)
{
    int i;
    for (i = size-1; i >= 0; i--)
    {
        printf(" %.2x", address[i]);
    }
    printf("\n");
}
void printBit(size_t const size, void const * const address)
{
    unsigned char *b = (unsigned char*) address;
    unsigned char byte;
    int i, j;
    int space;
    space=0;
    printf(" ");
    for (i=size-1;i>=0;i--)
    {
        for (j=7;j>=0;j--)
        {
            byte = b[i] & (1<<j);
            byte >>= j;
            printf("%u", byte);
            space++;
            if (space>=4) {
                  printf(" ");
                  space=0;
            }
        }
    }
    puts("");
}
```

Buatlah sebuah program sederhana yang menerima *input* sebuah bilangan *integer* lalu menampilkan representasinya baik dalam *bit* maupun *byte* menggunakan kedua fungsi tersebut. Program dapat dimodifikasi untuk tipe bilangan yang lain seperti *float* atau *double*.

Metodologi Praktikum

Sebelum melakukan praktikum, buatlah folder kerja sesuai dengan Nama, NIM, tanggal, dan modul praktikum (lihat petunjuk teknis pelaksanaan praktikum). Kerjakan masing-masing tugas di dalam folder kerja yang sesuai dengan nomor tugas tersebut. Kumpulkan tugas pendahuluan terlebih dahulu sebelum melaksanakan praktikum.

Tugas 1 : Fungsi XOR

Buatlah sebuah fungsi yang memiliki perilaku yang sama dengan operator *bitwise* XOR (^)! **Operator yang boleh** digunakan dalam merealisasikan fungsi ini hanya operator *bitwise* AND (`&`) dan operator *bitwise* NOT (~). Bila diperlukan Anda dapat menggunakan fungsi pada `printbitbyte.c`.

Prototype fungsi ini sebagai berikut.

```
int bitXor (int x, int y);
```

Contoh eksekusi fungsi ini adalah sebagai berikut.

```
result = bitXor(4, 5); //result = 1
```

Tugas 2 : Fungsi Ekstraksi Byte

Buatlah fungsi yang dapat melakukan ekstraksi byte ke-n dari suatu data X yang memiliki ukuran tertentu. Urutan byte pada data X diberi nomor 0 untuk LSB hingga 3 untuk MSB. Bila diperlukan, Anda dapat menggunakan fungsi pada `printbitbyte.c`.

Prototype fungsi ini sebagai berikut.

```
int getByte (int x, int n);
```

Contoh eksekusi fungsi ini adalah sebagai berikut.

```
result = getByte(0x12345678, 1); //result = 0x56
```

Tugas 3 : Fungsi Masking Byte

Buatlah fungsi yang dapat menghasilkan suatu *mask* dengan aturan seluruh bit di antara batas atas (`highbit`) dan batas bawah (`lowbit`) diset menjadi 1 sedangkan bit di luar batas atas (`highbit`) dan batas bawah (`lowbit`) diset menjadi 0. Asumsi yang digunakan adalah $0 \leq batas\ bawah \leq 31$ dan $0 \leq batas\ atas \leq 31$. Selain itu, bila $batas\ bawah > batas\ atas$ maka *masking* yang dihasilkan adalah `0x00000000`. Bila diperlukan Anda dapat menggunakan fungsi pada `printbitbyte.c`.

Prototype fungsi ini sebagai berikut.

```
int bitMask (int highbit, int lowbit);
```

Contoh eksekusi fungsi ini adalah sebagai berikut.

```
result = bitMask(5, 3); //result = 0x38
```

Tugas 4 : Fungsi Membalik Urutan Byte

Buatlah sebuah fungsi yang dapat membalik urutan byte dari suatu data X dengan menukar byte ke-0 dengan byte ke-3 dan byte ke-2 dengan byte ke-1. Urutan byte pada data berurutan dari byte ke-0 pada LSB dan byte ke-3 pada MSB. Bila diperlukan Anda dapat menggunakan fungsi pada `printbitbyte.c`.

Prototype fungsi ini sebagai berikut.

```
int reverseBytes (int x);
```

Contoh eksekusi fungsi ini adalah sebagai berikut.

```
result = reverseByte(0x01020304); //result = 0x04030201
```

Tugas 5 : Fungsi Pengurangan Byte

Buatlah sebuah fungsi yang dapat menghitung hasil pengurangan antara byte data pertama dikurangi byte data kedua. Sistem bilangan negatif yang digunakan adalah sistem *two's complement*. Operator pengurangan (-) **tidak boleh** digunakan dalam merealisasikan fungsi pengurangan byte ini. Hanya operator penjumlahan (+) dan invers (~) saja yang dapat digunakan untuk merealisasikan fungsi ini.

Prototype fungsi ini sebagai berikut.

```
int minBytes (int x, int y);
```

Contoh eksekusi fungsi ini adalah sebagai berikut.

```
result = minBytes (0x15, 0x07); //result = 0x0E
```

Tugas 6 : Fungsi Shift Register

Buatlah sebuah fungsi yang merepresentasikan sebuah *shift register* pada rangkaian sistem digital. Asumsikan bahwa jumlah bit sebanyak 32 bit dan setiap nilai yang dimasukkan ke dalam *shift register* secara bergantian adalah 5 bit. Nilai awal *shift register* harus `0x00000000`. Perlu diperhatikan bahwa pada *shift register*, *input* yang sebelumnya diberikan akan berpengaruh terhadap *state shift register* untuk *input* yang baru. Bila diperlukan Anda dapat menggunakan fungsi pada `printbitbyte.c`.

Prototype fungsi ini sebagai berikut.

```
int shiftRegister (int x);
```

Contoh eksekusi fungsi ini adalah sebagai berikut.

```
inisialisasi(); //global_var = 0x00000000
shiftRegister (0x04); //global_var = 0x00000004 (0b0000...0000000100)
shiftRegister (0x13); //global_var = 0x00000093 (0b0000...0010010011)
```

Tugas 7 : Program Enkripsi Sederhana

Buatlah sebuah program enkripsi (*encrypt*) sederhana yang berfungsi untuk menyamarkan 9 digit angka yang dimasukkan menggunakan *keyboard* oleh pengguna. Enkripsi dilakukan dengan menggunakan operasi XOR untuk setiap 8 bit dari 32 bit input dengan sebuah angka desimal 8 bit. Bila diperlukan Anda dapat menggunakan fungsi pada `printbitbyte.c` dan fungsi-fungsi lain yang telah Anda implementasikan pada tugas-tugas sebelumnya. Buat juga program untuk melakukan dekripsi (*decrypt*) sehingga dihasilkan bilangan aslinya. Gunakan makefile untuk mengkompilasi program ini.

Contohnya hasil eksekusi program ini adalah sebagai berikut. Diberikan bilangan input desimal `123456789` (dalam biner dapat ditulis `00000111 01011011 11001101 00010101`) dan bilangan desimal untuk input enkripsi sebesar `85`. Hasil bilangan *output* setelah melalui proses enkripsi adalah `1376688192` (dalam biner dapat ditulis `01010010 00001110 10011000 01000000`).

Tugas 8 : Pointer dalam Assembly

1. Buatlah program dengan kode sebagai berikut menggunakan teks editor Notepad++.

Sesuaikan informasi yang tertera pada *header*. Simpan program ini dengan nama `coba.c` pada folder kerja yang sesuai.

```
// Praktikum EL3111 Arsitektur Sistem Komputer
// Modul           :     3
// Percobaan       :     1A
// Tanggal         :     11 November 2013
// Kelompok        :     VI
// Rombongan       :     A
// Nama (NIM) 1    :     Audra Fildza Masita (13211008)
// Nama (NIM) 2    :     Bagus Hanindhito (13211007)
// Nama File       :     coba.c
void coba(int* x, int* y, int* z)
{
      // Kamus
      int a;
      int b;
      int c;
      int d;
      // Algoritma
      a = *x;
      b = *y;
      c = *z;
      d = a+b;
      *y = d;
      *z = b;
      *x = c;
```

2. Lakukan kompilasi program tersebut sehingga dihasilkan file *assembly* `coba.s`. Perhatikan angka-angka yang menunjukkan penggunaan *memory*. Apakah arti angka-angka tersebut? Gambarkan susunan *stack* pada *memory* yang digunakan oleh program tersebut.
3. Modifikasi program tersebut sehingga variabel `int` diubah menjadi variabel `double` lalu lakukan kompilasi ulang sehingga dihasilkan file *assembly* coba.s. Bandingkan dengan file *assembly* yang diperoleh pada langkah nomor 2. Apakah arti angka-angka tersebut? Gambarkan susunan *stack* pada *memory* yang digunakan oleh program tersebut.

Tugas 9 : Fungsi Membalik Urutan Array

Buatlah sebuah program yang dapat menerima beberapa karakter yang diberikan oleh pengguna, menyimpan karakter-karakter tersebut pada sebuah *array of character*, lalu menampilkannya di layar dengan susunan yang terbalik.

Contoh hasil eksekusi adalah sebagai berikut. Pengguna memberikan input karakter satu per satu sehingga membentuk susunan `HELLO`. Kemudian program akan menampilkannya di layar dengan susunan `OLLEH`.

Tugas 10 : Matriks Nama

Buatlah sebuah program yang dapat menerima input yang diberikan berupa nama orang yang dimasukkan karakter per karakter kemudian menyimpan data nama orang tersebut dalam *array of nama_orang*. Dengan mengingat bahwa *nama_orang* adalah *array of character*, maka program tersebut harus dapat merealisasikan *array of array of character*. Program ini kemudian juga dapat menampilkan isi dari *array of array of character* pada layar sesuai dengan urutan yang benar.

Tugas 11 : Matriks Nama dengan Pointer

Buatlah program yang sama dengan Tugas 10 namun terdapat modifikasi pada struktur data. Pada tugas ini, realisasi *array of array of character* diganti dengan *array of pointer to array of character*. Program

ini kemudian juga dapat menampilkan isi dari *array of pointer to array of character* pada layar sesuai dengan urutan yang benar. Bandingkan hasil implementasi pada tugas 10 dengan tugas 11 terutama dari segi efisiensi penggunaan *memory* dan pengaksesan *memory*.

Tugas 12 : Perkalian Matriks

Buatlah fungsi yang dapat melakukan perkalian dua buah matriks. Keluaran dari fungsi ini merupakan matriks hasil perkalian tersebut. Perhatikan syarat-syarat dua buah matriks dapat dikalikan berkaitan dengan dimensi masing-masing matriks termasuk dimensi matriks yang dihasilkan. Disarankan menggunakan matriks dinamis yang dialokasikan bergantung pada permintaan pengguna. Dalam melakukan operasi perkalian matriks, gunakanlah *loop* dua tingkat dengan pengaksesan yang *cache-friendly*. Terdapat beberapa cara mendefinisikan matriks dalam fungsi dalam bahasa C.

```
void mulMatriks(int A[m][n], int B[m][n], int C[m][n]);
```

Prototype fungsi ini memunculkan pertanyaan mengenai cara pemakaian fungsi ini dan kemungkinan implementasinya khususnya berkaitan dengan matriks C. Apakah matriks C hanya berupa pointer yang dialokasikan *memory*-nya dalam fungsi tersebut atau matriks C sudah dialokasikan dalam *memory* sehingga cukup diubah saja nilainya. Kecenderungan implementasi fungsi ini adalah matriks C telah dialokasikan terlebih dahulu. Risiko yang dihadapi adalah matriks C harus memiliki dimensi yang sesuai dengan hasil perkalian matriks A dan B.

```
int** mulMatriks (int A[m][n], int B[m][n]);
```

Prototype fungsi tersebut menjawab kelemahan *prototype* sebelumnya karena matriks C kini dialokasikan dalam fungsinya sesuai dengan dimensi matriks C yang diperlukan (*array* dinamis). Sayangnya, dengan fungsi ini kita tidak memiliki informasi tentang ukuran matriks A dan B padahal kita memerlukan informasi tersebut dalam memanggil fungsi ini. Salah satu cara untuk menyimpan informasi mengenai ukuran matriks adalah menggunakan variabel global. Sayangnya, penggunaan variabel global tidak disenangi karena membuat alur dan struktur program menjadi kurang jelas dan implementasi fungsi menjadi kurang fleksibel.

```
int** mulMatriks(int m, int n, int o, int A[m][n], int B[n][o]);
```

Prototype ini mengatasi permasalahan dari *prototype* sebelumnya. Variabel penyimpan ukuran matriks harus diberikan pada fungsi ini melalui parameter `m`, `n`, dan `o`. Sayangnya, hal ini membuat jumlah parameter fungsi menjadi banyak. Salah satu solusi adalah menyimpan matriks dalam suatu *structure*. Contohnya sebagai berikut.

```
struct Matriks
{
	int jumlahBaris;
	int jumlahKolom;
	int** nilai;
};
```

Dengan demikian *prototype* fungsi menjadi sebagai berikut.

```
struct Matriks mulMatriks(struct Matriks A, struct Matriks B);
```

Atau apabila ingin lebih efisien dalam hal penggunaan *memory*, kita dapat menggunakan sebagai berikut.

```
struct Matriks* mulMatriks(struct Matriks* pA, struct Matriks* pB);
```

Tugas 13 : Penjumlahan Biner dengan Array

Buatlah sebuah program yang dapat melakukan simulasi operasi penjumlahan dan pengurangan pada level bit dengan representasi *two's complement* menggunakan *array*. *Array* yang digunakan terdiri atas 8 bit saja dan hanya boleh diisi oleh angka 1 atau 0. Buatlah makefile untuk melakukan kompilasi

program ini. Bila diperlukan, Anda dapat menggunakan fungsi pada `printbitbyte.c`.
Contohnya adalah sebagai berikut. Misalkan ada penjumlahan antara 7 dan 8. Angka 7 akan dimasukkan ke dalam *array* berurutan untuk setiap bit menjadi 00000111 dan angka 8 akan dimasukkan ke dalam *array* berurutan untuk setiap bit menjadi 00001111. Kemudian hasil penjumlahannya adalah 00001111. Hasil penjumlahan ini kemudian diubah menjadi bilangan desimal dengan membaca isi *array* tersebut.

Hasil dan Analisis

Berikut ini adalah pertanyaan-pertanyaan yang dapat membantu analisis pada laporan praktikum. Perhatikan bahwa analisis tidak hanya terbatas pada pertanyaan-pertanyaan di bawah ini.

1. Bagaimana representasi tipe data *float* dan *double* dalam *memory*?
2. Mengapa urutan penulisan *structure* menentukan ukuran dari *structure* tersebut?
3. Bagaimana cara melakukan penulisan *structure* agar *memory* yang digunakan lebih efisien?
4. Apa saja perbedaan *structure* dengan *array*?
5. Apa perbedaan *array* dinamis dan *array* statis?
6. Bagaimana cara mengakses elemen dari *array*?
7. Bagaimana *n-dimensional array* direpresentasikan dalam *memory*?
8. Apa saja operasi-operasi yang dapat dilakukan pada operasi level bit dalam bahasa C?
9. Bagaimana prinsip pengurangan pada sistem bilangan *two's complement*?
10. Apa saja yang dapat dilakukan oleh *pointer*? Mengapa penggunaan *pointer* dapat berbahaya apabila tidak digunakan sesuai alokasi alamatnya?
11. Apa perbedaan *pointer to char*, *pointer to integer, pointer to double* dari segi bahasa *assembly*?
12. Apa perbedaan *pointer to char*, *pointer to integer, pointer to double* dari segi perhitungan *address*?
13. Apa algoritma perkalian matriks yang Anda pilih? Mengapa Anda menggunakan algoritma tersebut?
14. Apa perbedaan implementasi *array of pointer to array of character* dengan *array of character of character*?

Simpulan

Buatlah simpulan dari percobaan yang Anda lakukan dalam bentuk poin. Simpulan hendaknya menjawab tujuan praktikum ini.

Daftar Pustaka

Bryant, Randal, dan David O'Hallaron. *Computer Systems : A Programmer's Perspective 2nd Edition*. 2011. Massachusetts : Pearson Education Inc.

Patterson, David, dan John Hennessy. *Computer Organization and Design* : *The Hardware/Software Interface*. 2012. Waltham : Elsevier Inc.

Kernighan, Brian, dan Dennis Ritchie. *The C Programming Language 2nd edition*. 1988. Englewood Cliffs : Prentice Hall.

PERCOBAAN III
SYNTHESIZABLE MIPS32® MICROPROCESSOR BAGIAN I : INSTRUCTION SET, REGISTER, DAN MEMORY

Tujuan Praktikum

- Praktikan memahami arsitektur mikroprosesor MIPS32® beserta *datapath* eksekusinya.
- Praktikan memahami *instruction set* dari MIPS32® dan dapat membuat program sederhana dalam bahasa *assembly* yang dapat dieksekusi pada MIPS32®.
- Praktikan dapat melakukan simulasi eksekusi program MIPS32® pada program simulasi SPIM dan memahami cara setiap instruksi dieksekusi.
- Praktikan dapat membuat *instruction memory*, *data memory* dan *register* dari MIPS32® dalam kode VHDL yang *synthesizable* dan dapat disimulasikan dengan Altera® Quartus® II v9.1sp2.

Perangkat Praktikum

- Komputer Desktop / Laptop dengan sistem operasi Microsoft® Windows™ 7/8/8.1
- Altera® Quartus® II v9.1sp2 Web Edition atau Altera® Quartus® II v9.1sp2 Subscription Edition. (Altera® Quartus® II v10 atau yang lebih baru juga dapat digunakan, namun tidak terdapat simulator. Praktikan harus menggunakan Mentor Graphics® ModelSim® untuk melakukan simulasi).
- PCSpim sebagai simulator MIPS32® (untuk Microsoft® Windows™ 8/8.1, diperlukan mengunduh dan memasang Microsoft® .Net Framework 3.5 terlebih dahulu).
- Notepad++ sebagai teks editor.

Landasan Teoretis Praktikum

Bahasa VHDL

VHDL (Very High Speed Integrated Circuit Hardware Description Language) atau VHSIC Hardware Description Language merupakan bahasa untuk mendeskripsikan perangkat keras yang digunakan dalam desain elektronik digital dan *mixed-signal*, contohnya Field-Programmable Gate Array (FPGA) atau Integrated Circuit (IC). Sistem digital sangat erat kaitannya dengan sinyal. Sinyal dapat dianalogikan sebagai *wire* dan dapat berubah ketika *input* berubah. Dalam VHDL, terdapat definisi sinyal bernama `std_logic` yang sesuai dengan standar IEEE 1164. Terdapat sembilan jenis nilai sinyal yang didefinisikan dalam `std_logic`. Untuk menggunakan nilai sinyal standar `std_logic`, kita dapat menggunakan *library* yang telah tersedia yaitu `ieee.std_logic_1164.all`.

Simbol	Arti
`U`	Unknown
`X`	Forcing Unknown
`0`	Forcing 0
`1`	Forcing 1
`Z`	High Impedance
`W`	Weak Unknown
`L`	Weak 0
`H`	Weak 1
`-`	Don't Care

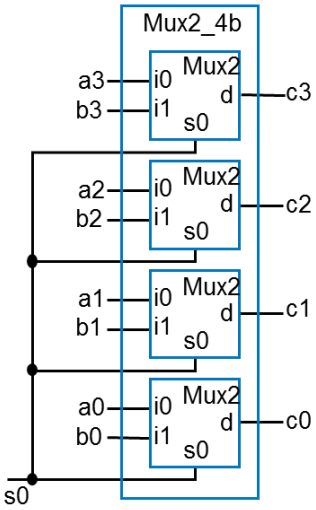
Tidak seperti bahasa Verilog HDL, VHDL merupakan bahasa yang *case insensitive*. Abstraksi utama dalam bahasa VHDL disebut entitas desain (*design entity*) yang terdiri atas *input*, *output*, dan fungsi yang didefinisikan secara benar. Entitas desain dalam VHDL terdiri atas dua bagian.

- Deklarasi Entitas (*entity declaration*) yang mendefinisikan antarmuka entitas tersebut terhadap dunia luar (contohnya *port input* dan *port output*).
- Arsitektur Entitas (*entity architecture*) yang mendefinisikan fungsi dari entitas (contohnya rangkaian logika di dalam entitas tersebut). Pendefinisian arsitektur dapat dilakukan secara *behavioral* maupun secara *structural*.

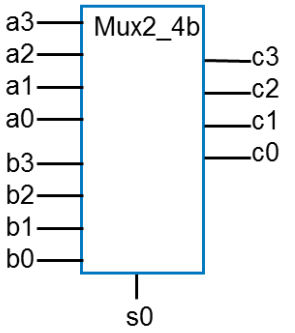
Setiap entitas desain harus disimpan dalam file VHDL yang terpisah dengan nama file sesuai dengan nama entitas yang dideklarasikan (contohnya `nama_entity.vhd`). Berikut ini *template* deklarasi sebuah entitas dan arsitektur entitas tersebut dalam bahasa VHDL.

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.all;

ENTITY <nama_entity> IS
      PORT(
                  <nama_port_1> : <tipe_port> STD_LOGIC;
                  <nama_port_2> : <tipe_port> STD_LOGIC_VECTOR(n DOWNTO 0)
            );
END <nama_entity>;

ARCHITECTURE <nama_arsitektur> OF <nama_entity> IS
BEGIN
      <fungsi yang didefinisikan>
END <nama_arsitektur>;
```

Setiap entitas desain dalam file VHDL yang berbeda dapat dipanggil dan dirangkai menjadi rangkaian yang lebih besar. Hal ini sangat penting dilakukan dalam melakukan desain *hardware*. Pertama, *hardware* yang akan didesain harus kita pecah-pecah menjadi komponen-komponen logika yang cukup kecil, contohnya menjadi *multiplexer*, *adder*, *flip-flop*, dan sebagainya. Kemudian, kita mendesain masing-masing komponen logika tersebut dan melakukan simulasi fungsional dan simulasi *timing* untuk setiap komponen untuk meyakinkan bahwa setiap komponen dapat berfungsi dengan baik. Setelah itu, kita menggabungkan masing-masing komponen untuk membentuk entitas desain yang lebih besar (*top level entity*).



Langkah pertama dalam membentuk *top level entity* adalah dengan mendefinisikan *top level entity* tersebut seperti halnya kita membuat entitas desain biasa. Kemudian, pada arsitektur *top level entity*, kita memanggil desain entitas lain menggunakan *construct component*. *Construct component* ini memiliki isi yang sama persis dengan deklarasi entitas desain yang akan dipanggil oleh *top level entity*. Kemudian, kita harus melakukan instansiasi masing-masing komponen dan menghubungkan *port input* dan *port output* dari masing-masing komponen dengan *top level design* atau dengan komponen lain.

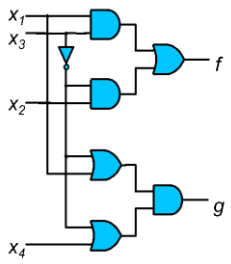


Contoh berikut digunakan untuk merealisasikan *2-to-1 multiplexer* 4-bit dari empat buah *2-to-1 multiplexer 1-bit*.

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

ENTITY Mux2_4b IS
   PORT   (
              A_IN     : IN     STD_LOGIC_VECTOR (3 DOWNTO 0);
              B_IN     : IN     STD_LOGIC_VECTOR (3 DOWNTO 0);
              S_IN     : IN     STD_LOGIC;
              C_OUT    : OUT    STD_LOGIC_VECTOR (3 DOWNTO 0)
          );
END Mux2_4b;
ARCHITECTURE Structural OF Mux2_4b IS
   COMPONENT Mux2 IS
      PORT    (
                 A     : IN STD_LOGIC;
                 B     : IN STD_LOGIC;
                 S     : IN STD_LOGIC;
                 D     : OUT STD_LOGIC
              );
   END COMPONENT;

BEGIN
   mux2_0 :   Mux2
              PORT MAP
                 (
                   A      => A_IN(0),
                   B      => B_IN(0),
                   S      => S_IN,
                   D      => C_OUT(0)
                 );
   mux2_1 :   Mux2
              PORT MAP
                 (
                   A      => A_IN(1),
                   B      => B_IN(1),
                   S      => S_IN,
                   D      => C_OUT(1)
                 );
   mux2_2 :   Mux2
              PORT MAP
                 (
                   A      => A_IN(2),
                   B      => B_IN(2),
                   S      => S_IN,
                   D      => C_OUT(2)
                 );
   mux2_3 :   Mux2
              PORT MAP
                 (
                   A      => A_IN(3),
                   B      => B_IN(3),
                   S      => S_IN,
                   D      => C_OUT(3)
                 );
END Structural;
```

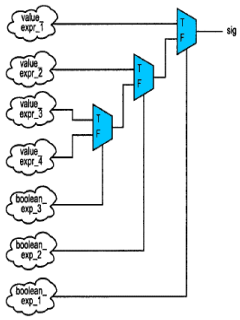
Terdapat tiga jenis *concurent signal assignment* (CSA) dalam bahasa VHDL, yaitu *simple* CSA, *conditional* CSA, dan *selected* CSA. Ketiga jenis *concurent signal assignment* tersebut dijelaskan menggunakan contoh sebagai berikut.

- *Simple* CSA. *Assignment* sinyal dilakukan dengan ekspresi logika biasa. Hasil implementasi *Simple* CSA akan berupa gerbang logika biasa.



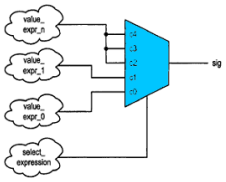

```
ARCHITECTURE  Behavioral OF Example IS
      BEGIN
        f <= (x1 AND x3) OR (NOT x3 AND x2);
        g <= (NOT x3 OR x1) AND (NOT x3 OR x4);
END Behavioral;
```

- *Conditional* CSA. *Assignment* sinyal dilakukan dengan *construct* `WHEN-ELSE`. Hasil implementasi *Conditional* CSA akan berupa kumpulan *2-to-1 multiplexer* yang disusun secara bertahap dengan `boolean_expr` sebagai selektor dan `value_expr` sebagai nilai sinyal yang dapat dipilih.




```
signal_name <=  value_expr_1 WHEN boolean_expr_1 ELSE
                value_expr_2 WHEN boolean_expr_2 ELSE
                value_expr_3 WHEN boolean_expr_3 ELSE
                ...
                value_expr_n;
```

- *Selected* CSA. *Assignment* sinyal dilakukan dengan *construct* `WITH-SELECT`. Hasil implementasi *Selected* CSA akan berupa satu buah *n-to-1 multiplexer* dengan `select_expression` sebagai selektor dan `value_expr_3` sebagai nilai sinyal yang dapat dipilih.




```
WITH select_expression SELECT
      signal_name <=    value_expr_1 WHEN choice_1,
                        value_expr_2 WHEN choice_2,
                        value_expr_3 WHEN choice_3,
                        ...
                        value_expr_n WHEN OTHERS;
```

Selain *concurent signal assignment*, dalam bahasa VHDL juga dikenal dengan *construct* `PROCESS` yang berfungsi melakukan *assignment* sinyal secara se kuensial. Sebuah proses (`PROCESS`) dapat dianalogikan sebagai bagian dari rangkaian yang dapat aktif dan dapat nonaktif. Sebuah proses akan diaktifkan ketika sinyal-sinyal (`SIGNAL`) dalam daftar sensitivitas (*sensitivity list*) mengalami perubahan nilai. Ketika diaktifkan, semua ekspresi dan pernyataan (*statement*) akan dieksekusi secara sekuensial hingga akhir dari proses tersebut.

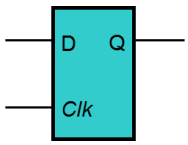
```
PROCESS (sensitivity_list)
    declarations;
    BEGIN
    sequential_statement_1;
    sequential_statement_2;
    ...
    sequential_statement_n;
END PROCESS;
```

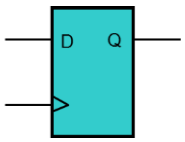
Terdapat dua jenis *construct* yang digunakan dalam *construct* `PROCESS`, yaitu *construct* `IF-THEN-ELSE` dan *construct* `CASE`. Kedua jenis *construct* tersebut diberikan sebagai contoh berikut ini.

```
ARCHITECTURE Behavioral OF mux2to1 IS
   BEGIN
      PROCESS (w0, w1, s)
        BEGIN
           IF s = '0' THEN
              f <= w0;
           ELSE
              f <= w1;
           END IF;
      END PROCESS;
END Behavioral;
```

```
ARCHITECTURE Behavioral OF mux2to1 IS
    BEGIN
       PROCESS (w0, w1, s)
            BEGIN
               CASE s IS
                  WHEN '0' =>
                     f <= w0;
                  WHEN OTHERS =>
                     f <= w1;
               END CASE;
       END PROCESS;
END Behavioral;
```

Dalam bahasa VHDL, kita juga dapat mendefinisikan beberapa jenis elemen *memory*. Dua jenis elemen *memory* yang sering digunakan dalam bahasa VHDL adalah *Gated* D Latch dan D Flip-flop. *Gated* D Latch memiliki karakteristik yaitu *output* akan berubah mengikuti *input* saat *clock high* (atau *clock low*, tergantung implementasi). Sedangkan D Flip-flop memiliki karakteristik yaitu *output* akan berubah mengikuti *input* saat transisi *clock* dari *low* ke *high* (atau *high* ke *low*, tergantung implementasi). Untuk elemen *memory* lain seperti *Gated* S-R Latch, T Flip-flop, dan JK Flip-flop juga dapat diimplementasikan pada bahasa VHDL namun mereka jarang digunakan.

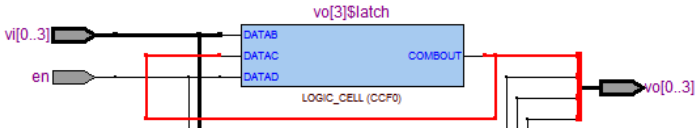




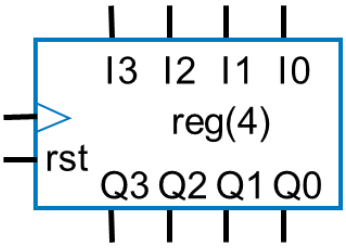

```
ENTITY latch IS
  PORT  (
           D    : IN     STD_LOGIC;
           Clk  : IN     STD_LOGIC;
           Q    : OUT    STD_LOGIC
        );
END latch;

ARCHITECTURE Behavioral OF latch IS
  BEGIN
    PROCESS ( D, Clk )
       BEGIN
          IF Clk = '1' THEN
             Q <= D;
          END IF;
    END PROCESS;
END Behavioral;
```

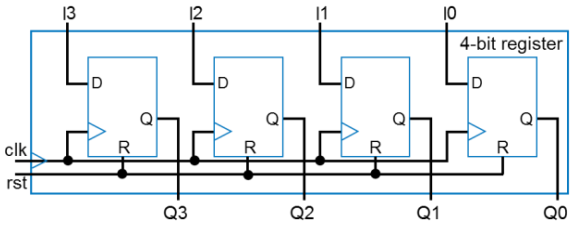
```
ENTITY flipflop IS
   PORT (
           D    : IN    STD_LOGIC ;
           Clk  : IN    STD_LOGIC ;
           Q    : OUT   STD_LOGIC
         );
END flipflop ;

ARCHITECTURE Behavior OF flipflop IS
BEGIN
   PROCESS ( Clock )
    BEGIN
    IF Clock'EVENT AND Clock='1' THEN
          Q <= D ;
       END IF ;
   END PROCESS ;
END Behavior ;
```

Penggunaan Latch dalam implementasi rangkaian menggunakan bahasa VHDL sebaiknya dihindari kecuali kita mengetahui apa yang kita lakukan. Dalam proses sintesis, implementasi Latch ini akan memberikan kita *warning*. Sebagian besar perangkat FPGA milik Altera tidak memiliki elemen dasar berupa Latch. Dengan demikian sebuah Latch harus dibuat menggunakan Logic Cell. Sayangnya, hal ini membutuhkan sebuah *feedback* pada Logic Cell untuk mengimplementasikan fungsi *memory*. Hal ini akan menyebabkan analisis *timing* statis tidak dapat dilakukan.



Salah satu komponen *memory* yang paling sering digunakan adalah *register*. *Register* terdiri atas beberapa buah *flip-flop* yang disusun sedemikian rupa sehingga membentuk elemen penyimpanan. *Register* juga dipakai untuk mengimplementasikan rangkaian sekuensial contohnya *finite state machine*.





Altera® Quartus® II

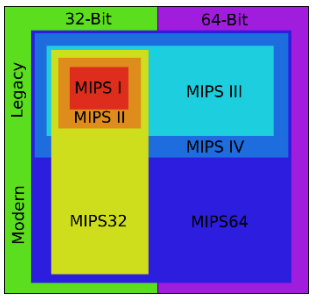
Pada modul praktikum ini tidak akan dibahas terlalu dalam cara-cara melakukan simulasi pada Altera® Quartus® II karena diasumsikan praktikan telah memperoleh pengalaman menggunakan program ini baik untuk simulasi fungsional dan simulasi *timing* saat mengambil Praktikum Sistem Digital pada tingkat II. Versi Altera® Quartus® II yang disarankan untuk digunakan dalam praktikum ini adalah Altera® Quartus® II v9.1sp2 karena pada versi ini terdapat simulator fungsional dan *timing* yang telah terintegrasi. Versi Altera® Quartus® II yang lebih baru tidak terdapat simulator fungsional dan *timing* sehingga praktikan harus menggunakan Mentor Graphics® ModelSim® untuk melakukan simulasi.

Langkah pertama untuk menggunakan Altera® Quartus® II adalah membuat *project* terlebih dahulu. Untuk membuat *project*, gunakan *new project wizard* kemudian ikuti petunjuk-petunjuk yang ada. Beri lokasi dan nama *project* yang diinginkan. Pilih dokumen-dokumen yang akan dimasukkan ke dalam *project* (kita dapat melewati langkah ini terlebih dahulu). Kemudian pilih *device* yang akan digunakan. Untuk praktikum ini, kita tidak akan melakukan implementasi pada FPGA karena praktikum ini hanya berupa simulasi saja. Oleh karena itu, kita dapat memilih FPGA dengan spesifikasi tertinggi baik untuk Altera® Cyclone™ maupun Altera® Stratix™. Setelah *project* dibuat, kita dapat mulai bekerja di dalamnya.

Untuk melakukan simulasi, kita harus melakukan kompilasi terhadap *project* yang kita buat. Kompilasi yang dilakukan bisa kompilasi penuh maupun hanya *Analysis & Synthesis* saja. Kompilasi penuh akan memakan waktu yang lebih lama karena semua proses meliputi *Analysis & Synthesis*, *Fitter*, dan *Assembler* akan dilakukan. Kompilasi penuh ini akan memberi kita gambaran terutama dari sisi *timing analysis*. Sedangkan dengan *Analysis & Synthesis*, kita telah mendapat rangkaian yang kita buat dan dapat dilakukan simulasi fungsional.

Mikroprosesor MIPS32®

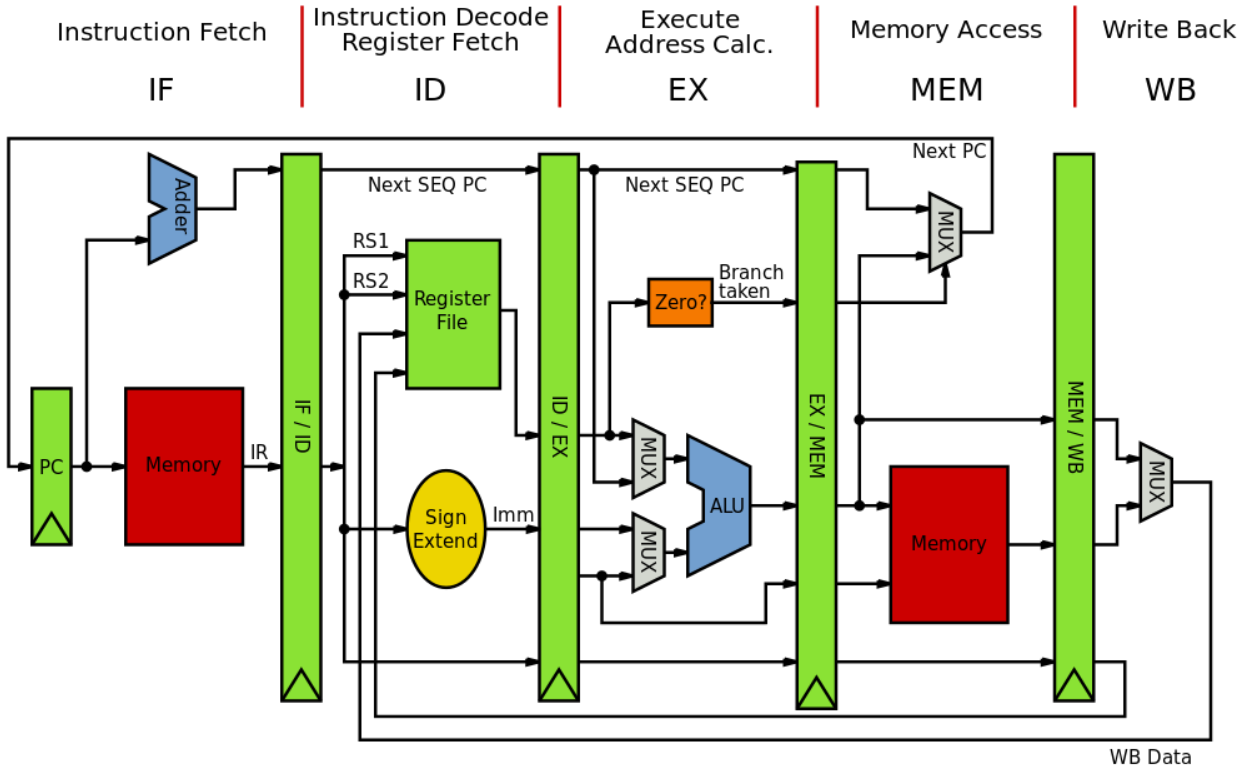
MIPS32® (Microprocessor without Interlocked Pipeline Stages) merupakan sebuah mikroprosesor 32-bit yang dikembangkan oleh MIPS Technologies. Mikroprosesor ini merupakan *reduced instruction set computer* (RISC). Mikroprosesor ini sering digunakan sebagai bahan pembelajaran mata kuliah Arsitektur Sistem Komputer diberbagai universitas dan sekolah teknik.



Beberapa turunan arsitektur MIPS®. Sumber: Wikipedia

Dalam kehidupan nyata, arsitektur mikroprosesor MIPS® sering digunakan dalam sistem *embedded* seperti perangkat Windows™ CE, *router*, *residential gateway*, dan konsol *video game* seperti Sony® PlayStation®,

Sony® PlayStation® 2 (PS2™), dan Sony® PlayStation® Portable (PSP®).



Arsitektur MIPS® menggunakan *pipeline* dengan lima tahap eksekusi. Sumber: Wikipedia

Terdapat lima tahap yang dilakukan ketika mikroprosesor MIPS32® melakukan eksekusi suatu instruksi. Kelima tahap tersebut adalah sebagai berikut.

- Instruction Fetch (IF)

  Tahap *instruction fetch* berfungsi mengatur aliran instruksi yang akan diolah pada tahap berikutnya. Instruksi yang sedang dijalankan merupakan instruksi yang berasal dan disimpan dari *memory*. Pada arsitektur ini, *memory* dipisahkan menjadi dua bagian yaitu *instruction memory* yang berfungsi menyimpan instruksi-instruksi yang akan dieksekusi dan *data memory* yang berfungsi untuk menyimpan data untuk menghindari *structural hazard*. Dengan demikian, arsitektur ini menganut *Harvard Architecture*.

- Instruction Decode (ID)

  Tahap berikutnya, instruksi yang telah diambil (*fetched*) dari *instruction memory* berpindah ke tahap *instruction decode*. Pada tahap ini, instruksi dengan lebar 32-bit akan dipecah sesuai format instruksi yang digunakan. Penjelasan mengenai *decoding* instruksi ini dapat dilihat pada bagian selanjutnya.

- Execute / Address Calculation (EX)

  Tahap ini merupakan tahap sebagian besar operasi aritmatika dan logika pada *arithmetic and logical unit* (ALU) dilakukan. Pada tahap ini juga terdapat tempat untuk meneruskan alamat register kembali ke tahap *instruction decode* sebagai deteksi *hazard*.

- Data Memory (MEM)

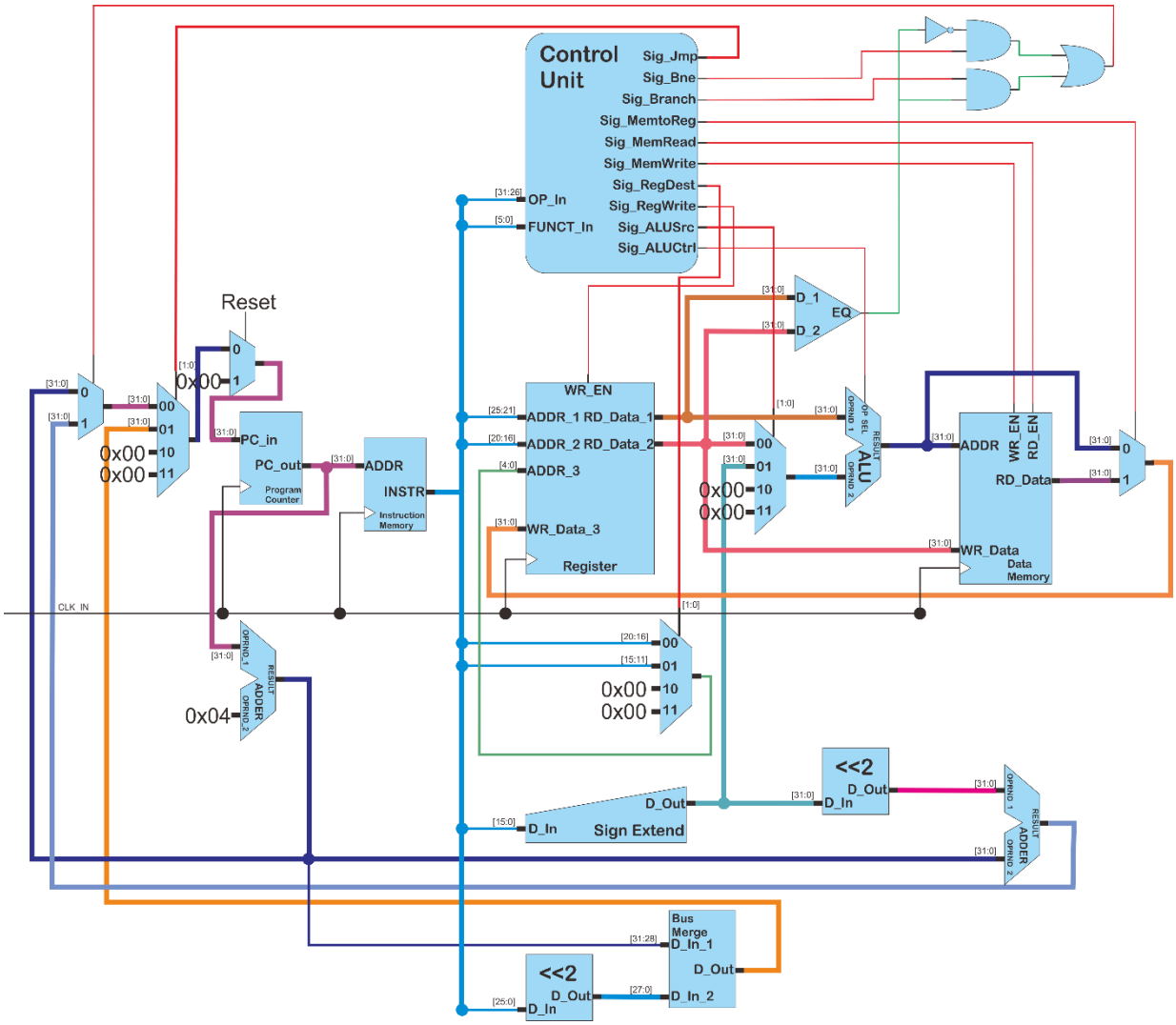
  Pada tahap ini, data disimpan dan/atau diambil dari *data memory*. *Data memory* hanya dapat

disimpan atau dibaca jika ada sinyal `MemRead` dan/atau `MemWrite` yang sesuai sehingga operasi baca dan/atau tulis pada *data memory* dapat dilakukan.

- Write Back (WB)

  Tahap terakhir ini digunakan untuk mengalirkan data dari *data memory* atau hasil perhitungan *arithmetic and logical unit* (ALU) ke *register* untuk dapat menjalankan instruksi selanjutnya.

Dalam praktikum ini, kita akan melakukan implementasi mikroprosesor MIPS32® yang sederhana. Mikroprosesor MIPS32® yang akan diimplementasikan tidak memiliki *pipeline* dan semua instruksi selesai dieksekusi dalam satu siklus *clock*. Dengan demikian, kita akan membuat mikroprosesor Single-Cycle MIPS32® menggunakan bahasa VHDL yang *synthesizable*. Diagram arsitektur mikroprosesor Single-Cycle MIPS32® yang akan kita buat diberikan sebagai berikut.



Instruction Set dan *Register* Mikroprosesor MIPS32®

Mikroprosesor MIPS32® memiliki set instruksi yang sederhana dibandingkan dengan mikroprosesor milik Intel®. Sebelum kita melangkah lebih jauh untuk melihat instruksi-instruksi dasar pada MIPS32®, kita perlu melihat *register* yang tersedia pada MIPS32®. Terdapat 32 buah *register* pada MIPS32® yang masing-masing register memiliki kegunaannya masing-masing. Semua *register* pada MIPS32® dapat diakses menggunakan *address* dengan lebar 5-bit. Tabel berikut merupakan daftar *register* yang tersedia dalam MIPS32® beserta fungsinya masing-masing.

Nama *Register*	Alamat	Fungsi
`$zero`	0	Nilai konstan 0
`$at`	1	Penyimpan Sementara Assembler
`$v0-$v1`	2-3	Penyimpan nilai dari hasil fungsi dan penyelesaian ekspresi
`$a0-$a3`	4-7	Penyimpan Argumen
`$t0-$t7`	8-15	Penyimpan sementara
`$s0-$s7`	16-23	Penyimpan sementara untuk pemanggilan fungsi
`$t8-$t9`	24-25	Penyimpan sementara
`$k0-$k1`	26-27	Digunakan oleh Kernel OS
`$gp`	28	*Pointer* global
`$sp`	29	*Pointer stack*
`$fp`	30	*Pointer frame*
`$ra`	31	*Return Address*

MIPS32® memiliki instruksi dengan lebar 32-bit. Instruksi-instruksi yang dimiliki MIPS32® dapat dilihat lebih lengkap pada lembar lampiran. Terdapat tiga buah format dasar dari instruksi MIPS32®. Ketiga format dasar instruksi tersebut adalah instruksi tipe-R, instruksi tipe-I, dan insruksi tipe-J. Format ketiga instruksi dasar tersebut dapat dilihat pada gambar berikut. Komponen dari ketiga format dasar instruksi tersebut dijelaskan pada tabel selanjutnya.

Bit	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R	opcode						rs					rt					rd					shamt					funct					
I	opcode						rs					rt					immediate															
J	opcode						address																									

Komponen	Keterangan
`opcode`	Menunjukkan jenis operasi yang akan dilakukan oleh instruksi tersebut. Khusus untuk instruksi tipe-R, opcode selalu bernilai 0x00.
`rs, rt, rd`	Menentukan alamat (nomor) dari *source register* (instruksi tipe-R dan instruksi tipe-I), *temporary register* (instruksi tipe-R dan instruksi tipe-I), dan *destination register* (instruksi tipe-R).
`shamt`	Menunjukkan jumlah penggeseran bit (*shift amount*) pada instruksi tipe-R.
`funct`	Memilih operasi matematika yang akan dilakukan pada instruksi tipe-R
`immediate`	Menentukan nilai konstanta yang menunjukkan *operand* yang konstan atau *address*.
`address`	Alamat tujuan pada *instruction memory* yang akan dieksekusi setelahnya.

Dalam praktikum ini, mikroprosesor Single-Cycle MIPS32® yang akan diimplementasikan harus dapat menjalankan sembilan buah instruksi sebagai berikut.

Instruksi	Tipe	`opcode`	`funct`	Keterangan
`add`	R	`000000`	`100000`	Operasi penjumlahan
`sub`	R	`000000`	`100010`	Operasi pengurangan
`beq`	I	`000100`	`------`	Pencabangan bila sama (*Branch-if-Equal*)
`bne`	I	`000101`	`------`	Pencabangan bila tidak sama (*Branch-if-Not-Equal*)
`addi`	I	`001000`	`------`	Operasi penjumlahan dengan konstanta
`lw`	I	`100011`	`------`	Mengambil data dari *data memory* (*load word*)
`sw`	I	`101011`	`------`	Menyimpan data ke *data memory* (*save word*)
`jmp`	J	`000010`	`------`	Menuju ke instruksi pada *address* tertentu (*jump*)
`nop`	-	`000000`	`000000`	Tidak ada operasi (*no operation*). Digunakan untuk menambahkan jeda satu siklus setelah instruksi *branching* dilakukan.

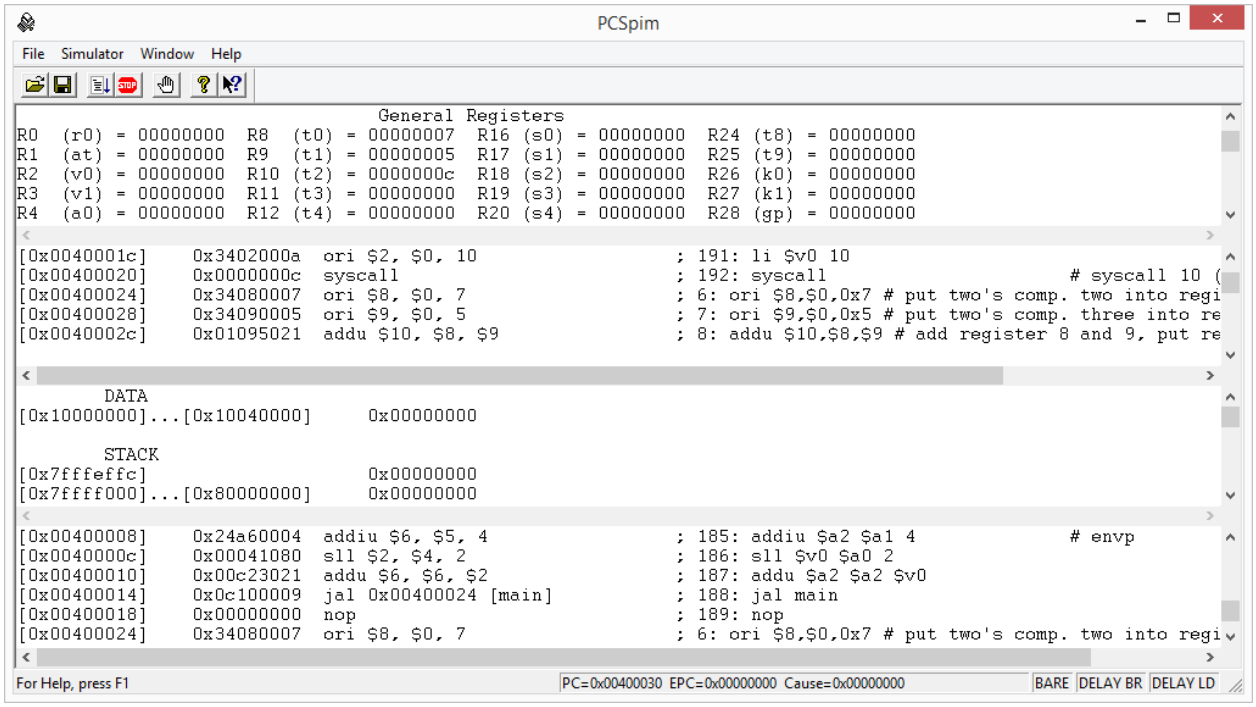
Sebelum kita mendesain mikroprosesor Single-Cycle MIPS32®, ada baiknya kita mempelajari terlebih dahulu bagaimana sebuah instruksi dieksekusi oleh mikroprosesor MIPS32® tersebut. Kita dapat menggunakan sebuah simulator untuk melakukan eksekusi program yang kita buat pada mikroprosesor MIPS32® lalu melihat hasilnya. Simulator MIPS32® yang akan digunakan dalam praktikum ini adalah PCSpim. Simulator PCSpim dapat diunduh di **http://el3111.bagus.my.id**.

Perhatikan bahwa program ini membutuhkan Microsoft® .Net Framework 2.0 untuk dapat berjalan. Bagi praktikan yang menggunakan Microsoft® Windows® XP, Microsoft® Windows® 8, dan Microsoft® Windows® 8.1 harus memasang Microsoft® .Net Framework 2.0 terlebih dahulu. Microsoft® .Net Framework 2.0 dapat diunduh di **http://el3111.bagus.my.id** dalam paket instalasi Microsoft® .Net Framework 3.5.

Setelah instalasi selesai PCSpim, kita dapat langsung menjalankan PCSpim dari *start menu*. Khusus untuk pengguna Microsoft® Windows® edisi 64-bit, terkadang PCSpim akan memberikan pesan *error* karena tidak dapat menemukan file `exceptions.s`. Untuk mengatasinya, pilih menu Simulator lalu klik submenu *Settings*. Pada bagian *Load exception file*, ganti alamat file `exceptions.s`.

Sebelum : `C:\Program Files\PCSpim\exceptions.s`

Sesudah : `C:\Program Files (x86)\PCSpim\exceptions.s`



Jendela dari PCSpim dibagi menjadi empat bagian. Bagian pertama merupakan *Register Display* yang berisi isi dari setiap register pada MIPS32® meliputi 32 *general purpose register* dan beberapa *floating point register* serta beberapa *register* yang lain. Isi dari setiap *register* yang ditampilkan dalam format heksadesimal. Bagian kedua merupakan *Text Display* yang berisi program dalam bahasa *assembly*, kode instruksi dalam heksadesimal, dan alamat instruksi tersebut. Bagian ketiga merupakan *Data and Stack Display* yang berisi isi *memory* dalam MIPS32® yang menampung data-data serta *stack*. Bagian keempat merupakan SPIM *Message* yang berisi laporan dari simulator ketika terjadi galat.

Bila dalam program bahasa *assembly* yang kita buat terdapat perintah untuk menampilkan sesuatu ke layar (mirip dengan `printf` dalam bahasa C), maka *output* ke layar tersebut akan ditampilkan

dalam jendela konsol termasuk apabila program meminta pengguna memasukkan *input*. Untuk memulai penggunaan PCSpim pertama kali, Anda akan diminta untuk menjalankan program sederhana.

Buatlah program dalam bahasa *assembly* dengan menyalin kode program di bawah ini menggunakan teks editor Notepad++. Simpan file tersebut dengan nama `add.asm`. Kalimat di sebelah kanan tanda `#` merupakan komentar dan tidak akan dieksekusi oleh simulator. Ubah konfigurasi PCSpim agar menjalankan simulasi menggunakan *Bare Machine* dengan membuka menu Simulator lalu submenu *Settings*.

```
# Program untuk menjumlahkan 7 dengan 5
            .text
            .globl main
main:
            ori $8,$0,0x07     # masukkan angka 7 ke register 8
            ori $9,$0,0x05     # masukkan angka 5 ke register 9
            addu $10,$8,$9     # jumlahkan dan simpan hasilnya di register 10
# akhir dari program
```

Buka file `add.asm` menggunakan PCSpim dengan membuka menu *File* lalu *Open*. Bila terjadi kesalahan sintaks dalam pemrograman bahasa *assembly*, PCSpim akan mengeluarkan pesan galat. Periksa kembali program yang dibuat lalu simpan program tersebut sebelum dibuka kembali menggunakan PCSpim. Bila program berhasil dibuka, kita dapat melihat bahwa file bahasa *assembly* telah diterjemahkan menjadi instruksi-instruksi dalam bahasa heksadesimal dan disimpan dalam *instruction memory*.

Untuk memulai eksekusi, kita harus mengeset nilai *program counter* (`PC`). *Program counter* (`PC`) merupakan bagian dari mikroprosesor yang menyimpan *address* instruksi yang akan dieksekusi. Pada bagian *Register Display*, terlihat bahwa `PC` bernilai `0x00000000`. Ubah nilai `PC` tersebut menjadi `0x00400000` dengan membuka menu Simulator, lalu submenu *Set Value*. Tuliskan `PC` pada kotak isian paling atas dan `0x00400000` pada kotak isian paling bawah. Hal ini dilakukan karena program yang kita buat dimulai pada *address* tersebut.

Tekan tombol F10 pada *keyboard* untuk melakukan eksekusi satu instruksi. Tekan tombol F10 hingga instruksi dari program yang kita tulis dapat dieksekusi. Perhatikan bahwa saat instruksi pertama dilakukan, nilai *register* `8` berubah menjadi `0x07` dan PC berubah menjadi `0x00400004`. Tekan kembali tombol F10 pada *keyboard* untuk melakukan eksekusi satu instruksi berikutnya dan perhatikan yang terjadi pada *register* `9`. Tekan kembali tombol F10 pada *keyboard* untuk melakukan eksekusi satu instruksi berikutnya dan perhatikan yang terjadi pada *register* `10`. Hasil penjumlahan kedua bilangan tersebut disimpan pada *register* `10`.

Tugas Pendahuluan

1. Jelaskan bagaimana MIPS32® melakukan eksekusi sebuah instruksi dan jelaskan format tiga instruksi dasar yang dapat dieksekusi oleh MIPS32® beserta penjelasannya untuk setiap bit instruksi! Berikan pula masing-masing lima contoh penggunaan instruksi untuk masing-masing format instruksi dasar!
2. Tentukan nilai *opcode* dan *funct* dalam biner, tipe instruksi, dan arti instruksi dari instruksi-instruksi di bawah ini.

`sll`	`sub`	`nor`	`addi`	`xori`	`j`
`srl`	`and`	`slt`	`slti`	`lui`	`jal`
`jr`	`or`	`beq`	`andi`	`lw`	`addiu`

add xor bne ori sw sltiu

3. Diberikan program dalam bahasa *assembly* berikut ini untuk dieksekusi dalam MIPS32®. Program ini meminta pengguna untuk memasukkan nilai dalam *ounce* lalu melakukan konversi dari *ounce* ke *pound* dan *ounce* dan menampilkan hasil konversinya. Gunakan teks editor Notepad++ untuk menyalin program ini dan menyimpannya dalam file `contoh_ounces.asm`

```
# contoh_ounces.asm
# Konversi dari ounces ke pounds dan ounce.

            .data
   prompt:  .asciiz "Masukkan massa dalam ounces: "
   pout:    .asciiz " Pounds\n"
   ozout:   .asciiz " Ounces\n"
            .text
            .globl main

   main:    addu $s0, $ra, $0       # simpan $31 dalam $16
            li $v0,4                # tampilkan perintah
            la $a0,prompt
            syscall
            li $v0,5                # baca input pengguna
            syscall
            li $t1,16               # 1 pound = 16 ounce
            divu $v0,$t1
            mflo $a0
            li $v0,1                # tampilkan nilai pound
            syscall
            li $v0,4                # tampilkan str "pounds"
            la $a0,pout
            syscall
            mfhi $a0                # tampilkan nilai ounce
            li $v0,1
            syscall
            li $v0,4                # tampilkan str "ounces"
            la $a0,ozout
            syscall
            addu $ra, $0, $s0
            jr $ra
# akhir dari program
```

   a. Simulasikan program tersebut dalam PCSpim lalu *screenshot* hasil yang ditampilkan dalam *console*!. Perhatikan bahwa PCSpim perlu dikonfigurasi untuk melakukan simulasi menggunakan *pseudoinstruction* dengan memilih menu Simulator lalu submenu *settings*. Aktifkan *allow pseudo instruction* lalu nonaktifkan pilihan *bare machine*.
   b. Konversi kode bahasa *assembly* tersebut ke dalam bahasa C dan lakukan kompilasi menggunakan GCC untuk kemudian dijalankan dalam komputer Anda. *Screenshot* hasil yang ditampilkan dalam *console*!.
   c. Bandingkan bahasa *assembly* program untuk dieksekusi pada mikroprosesor MIPS32® dan bahasa *assembly* hasil kompilasi oleh GCC untuk dieksekusi pada mikroprosesor Intel® x86. Apa komentar Anda?

4. Buatlah program dalam bahasa *assembly* untuk dieksekusi dalam mikroprosesor MIPS32®. Program ini menerima *input* berupa total bahan bakar yang dikonsumsi oleh mobil dalam satuan liter dan total jarak yang ditempuh oleh mobil dalam satuan kilometer dengan jumlah bahan bakar tersebut. Kemudian program melakukan perhitungan untuk rata-rata konsumsi

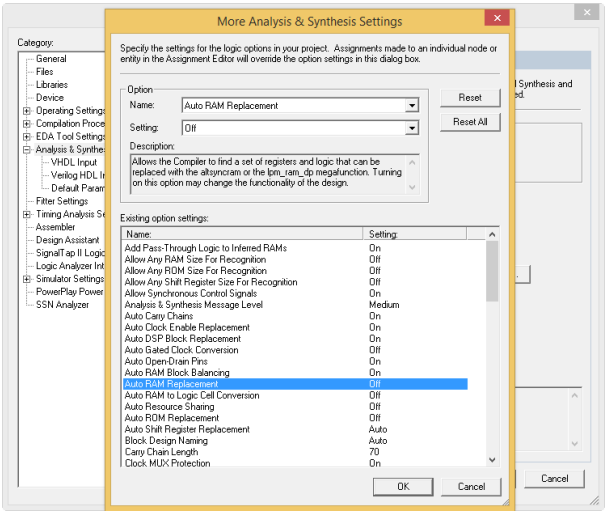
bahan bakar per kilometer serta jarak yang dapat ditempuh dalam satuan kilometer menggunakan 1 liter bahan bakar. Sertakan kode bahasa *assembly* Anda dan *screenshot* hasil yang ditampilkan pada *console*.

5. Buatlah program dalam bahasa *assembly* untuk dieksekusi dalam MIPS32® dengan fungsionalitas yang sama dengan program dalam bahasa C berikut ini. Simulasikan program ini dalam PCSpim dan *screenshot* hasil yang ditampilkan dalam *console*. (Petunjuk: gunakan `bne`, `beq`, atau `j` untuk merealisasikan *loop*.

```
void main()
{
	int a;
	int p;
	int x;
	a = 0;
	x = 1;
	printf("Masukkan jumlah loop: ");
	scanf("%d", &p);
	while (a<p)
	{
		x = x * 2;
		a = a + 1;
	}
	printf("Hasil iterasi: %d\n", x);
}
```

Metodologi Praktikum

Sebelum melakukan praktikum, buatlah folder kerja sesuai dengan Nama, NIM, tanggal, dan modul praktikum (lihat petunjuk teknis pelaksanaan praktikum). Kerjakan masing-masing tugas di dalam folder kerja yang sesuai dengan nomor tugas tersebut. Kumpulkan tugas pendahuluan terlebih dahulu sebelum melaksanakan praktikum.



Untuk setiap *project* pada Altera® Quartus II, gunakan konfigurasi berikut ini. Pilih menu *Assignment* lalu pilih submenu *Settings*. Pada *category Analysis & Synthesis*, klik *More Settings*. Ubah pengaturan *Auto RAM Replacement* dan *Auto ROM Replacement* ke `OFF`. Hal ini akan mencegah *synthetizer* untuk menggunakan Altera® MegaFunction untuk merealisasikan blok *memory* yang kita buat bila tidak dibutuhkan penggunaan Altera® MegaFunction. Praktikan disarankan membaca terlebih dahulu mengenai Altera® MegaFunction ALTSYNCRAM pada bagian lampiran.

Tugas 1 : Perancangan *Instruction Memory*

Dalam mikroprosesor Single-Cycle MIPS32® yang akan kita realisasikan, *instruction memory* memiliki lebar data sebesar 32-bit dan lebar *address* sebesar 32-bit. Untuk praktikum ini, kita akan merealisasikan *instruction memory* yang dapat menampung 32 buah instruksi. Dengan demikian, tidak semua *address* terpakai dalam praktikum ini. Hanya 32 *address* paling awal saja yang dipakai. *Instruction*

memory memiliki sebuah *port input* yang menerima *address* dengan lebar 32-bit dan sebuah *port output* yang mengeluarkan instruksi dengan lebar data 32-bit. Terdapat pula *port input clock* untuk mengendalikan rangkaian ini.

1. Implementasikan desain *instruction memory* dalam bahasa VHDL dengan menyalin kode VHDL berikut ini menggunakan teks editor Notepad++. Sesuaikan identitas pada *header*. Simpan file ini dengan nama `instrucMEM.vhd`.

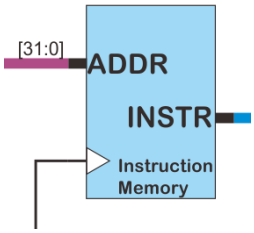
```
-- Praktikum EL3111 Arsitektur Sistem Komputer
-- Modul           :     4
-- Percobaan       :     1
-- Tanggal         :     18 November 2013
-- Kelompok        :     VI
-- Rombongan       :     A
-- Nama (NIM) 1    :     Audra Fildza Masita (13211008)
-- Nama (NIM) 2    :     Bagus Hanindhito (13211007)
-- Nama File       :     instrucMEM.vhd
-- Deskripsi       :     Implementasi instruction memory

LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_ARITH.ALL;
USE IEEE.STD_LOGIC_UNSIGNED.ALL;
LIBRARY altera_mf;
USE altera_mf.altera_mf_components.ALL;

ENTITY instrucMEM IS
   PORT  (
            ADDR          : IN std_logic_vector (31 DOWNTO 0);
            clock         : IN std_logic;
            reset         : IN std_logic;
            INSTR         : OUT std_logic_vector (31 DOWNTO 0)
        );
END ENTITY;

ARCHITECTURE behavior OF instrucMEM IS
TYPE ramtype IS ARRAY (31 DOWNTO 0) OF std_logic_vector (31 DOWNTO 0);
SIGNAL mem: ramtype;
   BEGIN
      PROCESS (reset,ADDR,mem)
      BEGIN
        IF (reset='1') THEN
             INSTR <= (OTHERS => '0');
        ELSE
             INSTR <= mem(conv_integer (ADDR));
        END IF;
      END PROCESS;
-- Isi dalam instruction memory
mem(0) <= X"00000022";
mem(1) <= X"8c010000";
mem(2) <= X"8c020004";
mem(3) <= X"8c030008";
mem(4) <= X"00842022";
mem(5) <= X"00822020";
mem(6) <= X"0043282a";
mem(7) <= X"10a00002";
mem(8) <= X"00411020";
mem(9) <= X"1000fffb";
mem(10) <= X"ac040000";
mem(11) <= X"1000ffff";
```

```
END behavior;
```

2. Simulasikan secara fungsional dan *timing* desain tersebut setelah melakukan *analysis & synthesis* terlebih dahulu. Perhatikan apakah *waveform* telah sesuai dengan *address* yang dimasukkan.

Tugas 2 : Perancangan *Instruction Memory* dengan Altera® MegaFunction ALTSYNCRAM

Pada bagian ini, kita akan memanfaatkan sebuah *template* desain yang telah tersedia dalam Altera® Quartus® II yaitu Altera® MegaFunction ALTSYNCRAM. *Template* ini dapat digunakan untuk merealisasikan *synchronous* RAM dan ROM dalam desain kita. Selain itu, kita dapat menggunakan inisialisasi isi *memory* dari file eksternal berformat `.mif` (*memory initialization file*).



1. Implementasikan desain *instruction memory* dalam bahasa VHDL dengan menyalin kode VHDL berikut ini menggunakan teks editor Notepad++. Sesuaikan identitas pada *header*. Simpan file ini dengan nama `instruction_memory.vhd`.

```
-- Praktikum EL3111 Arsitektur Sistem Komputer
-- Modul           :     4
-- Percobaan       :     1
-- Tanggal         :     18 November 2013
-- Kelompok        :     VI
-- Rombongan       :     A
-- Nama (NIM) 1    :     Audra Fildza Masita (13211008)
-- Nama (NIM) 2    :     Bagus Hanindhito (13211007)
-- Nama File       :     instruction_memory.vhd

LIBRARY ieee;
USE ieee.std_logic_1164.all;
LIBRARY altera_mf;
USE altera_mf.all;

ENTITY instruction_memory IS
 PORT  (
          ADDR       : IN STD_LOGIC_VECTOR (31 DOWNTO 0);    -- alamat
          clock      : IN STD_LOGIC := '1';                  -- clock
          INSTR      : OUT STD_LOGIC_VECTOR (31 DOWNTO 0)    -- output
       );
END ENTITY;

ARCHITECTURE structural OF instruction_memory IS
SIGNAL sub_wire0 : STD_LOGIC_VECTOR (31 DOWNTO 0);
-- signal keluaran output
COMPONENT altsyncram
-- komponen memori
   GENERIC
      (
          init_file             : STRING;    -- name of the .mif file
          operation_mode        : STRING;    -- the operation mode
          widthad_a             : NATURAL;   -- width of address_a[]
          width_a               : NATURAL    -- width of data_a[]
      );

   PORT
      (
          clock0                : IN STD_LOGIC ;
          address_a             : IN STD_LOGIC_VECTOR (31 DOWNTO 0);
          q_a                   : OUT STD_LOGIC_VECTOR (31 DOWNTO 0)
```

```
      );
END COMPONENT;



BEGIN
   INSTR <= sub_wire0;
   altsyncram_component : altsyncram
   GENERIC MAP
      (
          init_file => "imemory.mif",
          operation_mode => "ROM",
          widthad_a => 32,
          width_a => 32
      )
   PORT MAP
      (
            clock0 => clock,
            address_a => ADDR,
            q_a => sub_wire0
      );
END structural;
```

Perhatikan bahwa bila desain tidak dapat disintesis akibat tidak mencukupinya FPGA yang digunakan untuk merealisasikan elemen *memory*, modifikasi berikut ini harus dilakukan. Ubah lebar *address* menjadi 16-bit. Dengan demikian, beberapa konfigurasi harus diubah.

Sebelum : `ADDR : IN STD_LOGIC_VECTOR (31 DOWNTO 0);`
Sesudah : `ADDR : IN STD_LOGIC_VECTOR (15 DOWNTO 0);`

Sebelum : `address_a : IN STD_LOGIC_VECTOR (31 DOWNTO 0);`
Sesudah : `address_a : IN STD_LOGIC_VECTOR (15 DOWNTO 0);`

Sebelum : `widthad_a => 32,`
Sesudah : `widthad_a => 16,`

Dengan demikian, kita harus berhati-hati dalam memasukkan *address* karena lebar *address* bukan lagi 32-bit melainkan 16-bit.

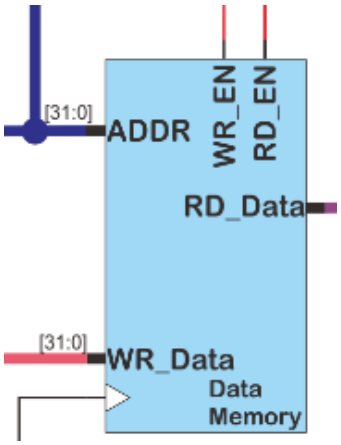
2. Buatlah inisialisasi *memory* dalam bentuk file *memory initialization file* menggunakan teks editor Notepad++ dengan menyalin isi file berikut ini. Simpan file ini dengan nama `imemory.mif` pada lokasi yang sama dengan tempat `instruction_memory.vhd` berada.

```
WIDTH=32;                   -- number of bits of data per word
DEPTH=256;                  -- the number of addresses
ADDRESS_RADIX=HEX;
DATA_RADIX=HEX;
CONTENT
BEGIN
      00 : 8c020000;
      04 : 8c030001;
      08 : 00430820;
      0C : ac010003;
      10 : 1022ffff;
      14 : 1021fffa;
      06 : 0043282a;
      07 : 10a00002;
      09 : 1000fffb;
      10 : ac040000;
      11 : 1000ffff;
```

```
END;
```

3. Simulasikan secara fungsional dan *timing* desain tersebut setelah melakukan *analysis & synthesis* terlebih dahulu. Perhatikan apakah *waveform* telah sesuai dengan *address* yang dimasukkan. Bandingkan dengan hasil simulasi pada Tugas 1.

Tugas 3 : Perancangan *Data Memory* dengan Altera® MegaFunction ALTSYNCRAM



Dalam mikroprosesor Single-Cycle MIPS32® yang akan kita realisasikan, *data memory* memiliki lebar data sebesar 32-bit dan lebar *address* sebesar 32-bit. Untuk praktikum ini, kita akan merealisasikan *data memory* yang dapat menampung 256 buah data. Dengan demikian, tidak semua *address* terpakai dalam praktikum ini. Hanya 256 *address* paling awal saja yang dipakai. *Data memory* memiliki sebuah *port input* yang menerima *address* dengan lebar 32-bit dan sebuah *port output* yang mengeluarkan data yang dibaca dengan lebar data 32-bit. *Data memory* juga memiliki *port input* untuk memasukkan data yang akan ditulis dengan lebar data 32-bit. Terdapat pula *port input clock* untuk mengendalikan rangkaian ini, *port input* untuk mengaktifkan mode tulis (*write enable*), dan *port input* untuk mengaktifkan mode baca (*read enable*). Karena desain yang mirip dengan desain *instruction memory*, kita akan memanfaatkan kembali Altera® MegaFunction ALTSYNCRAM untuk merealisasikan rangkaian ini.

1. Implementasikan desain *data memory* dalam bahasa VHDL dengan deklarasi entitas sebagai berikut. Gunakan ALTSYNCRAM untuk merealisasikan desain *data memory*. Penggunaan ALTSYNCRAM dapat dilihat pada kode selanjutnya. Simpan file ini dengan nama `data_memory.vhd`.

```
ENTITY data_memory IS
  PORT (
          ADDR       : IN STD_LOGIC_VECTOR (31 DOWNTO 0); -- alamat
          WR_EN      : IN STD_LOGIC;         --Indikator Penulisan
          RD_EN      : IN STD_LOGIC;         --Indikator Pembacaan
          clock      : IN STD_LOGIC := '1'; -- clock
          RD_Data    : OUT STD_LOGIC_VECTOR (31 DOWNTO 0);
          WR_Data    : IN STD_LOGIC_VECTOR (31 DOWNTO 0)
       );
END ENTITY;
```

```
COMPONENT altsyncram
-- komponen memori
GENERIC
  (
    init_file           : STRING;     -- name of the .mif file
    operation_mode      : STRING;     -- the operation mode
    widthad_a           : NATURAL;    -- width of address_a[]
    width_a             : NATURAL     -- width of data_a[]
  );

PORT
  (
    wren_a     : IN STD_LOGIC;        -- Write Enable Activation
    clock0     : IN STD_LOGIC ;       -- Clock
    address_a  : IN STD_LOGIC_VECTOR (31 DOWNTO 0);    -- Address Input
```

```
    q_a         : OUT STD_LOGIC_VECTOR (31 DOWNTO 0);    -- Data Output
    data_a      : IN STD_LOGIC_VECTOR (31 DOWNTO 0)      -- Data Input
 );
END COMPONENT;
```

```
altsyncram_component : altsyncram
      GENERIC MAP
         (
           init_file           => "dmemory.mif",
           operation_mode      => "SINGLE_PORT",
           widthad_a           => 32,
           width_a             => 32
         )
      PORT MAP
         (
           wren_a              => ??????????, -- isi yang sesuai
           clock0              => clock,
           address_a           => ADDR,
           q_a                 => ??????????, -- isi yang sesuai
           data_a              => WR_Data
         );
```

Perhatikan bahwa bila desain tidak dapat disintesis akibat tidak mencukupinya FPGA yang digunakan untuk merealisasikan elemen *memory*, modifikasi berikut ini harus dilakukan. Ubah lebar *address* menjadi 8-bit dan lebar data menjadi 8-bit Dengan demikian, beberapa konfigurasi harus diubah sebagai berikut.

```
Sebelum   :  ADDR         : IN STD_LOGIC_VECTOR (31 DOWNTO 0);
Sesudah   :  ADDR         : IN STD_LOGIC_VECTOR (7 DOWNTO 0);

Sebelum   :  RD_Data      : OUT STD_LOGIC_VECTOR (31 DOWNTO 0);
Sesudah   :  RD_Data      : OUT STD_LOGIC_VECTOR (7 DOWNTO 0);

Sebelum   :  WR_DATA      : IN STD_LOGIC_VECTOR (31 DOWNTO 0)
Sesudah   :  WR_DATA      : IN STD_LOGIC_VECTOR (7 DOWNTO 0)

Sebelum   :  address_a    : IN STD_LOGIC_VECTOR (31 DOWNTO 0);
Sesudah   :  address_a    : IN STD_LOGIC_VECTOR (7 DOWNTO 0);

Sebelum   :  q_a          : OUT STD_LOGIC_VECTOR (31 DOWNTO 0);
Sesudah   :  q_a          : OUT STD_LOGIC_VECTOR (7 DOWNTO 0);

Sebelum   :  data_a       : IN STD_LOGIC_VECTOR (31 DOWNTO 0)
Sesudah   :  data_a       : IN STD_LOGIC_VECTOR (7 DOWNTO 0)

Sebelum   :  widthad_a    => 32,
Sesudah   :  widthad_a    => 8,

Sebelum   :  width_a      => 32
Sesudah   :  width_a      => 8
```

Dengan demikian, kita harus berhati-hati dalam memasukkan *address* karena lebar *address* bukan lagi 32-bit melainkan 8-bit. Hal yang sama juga berlaku pada data karena lebar data bukan lagi 32-bit melainkan 8-bit.

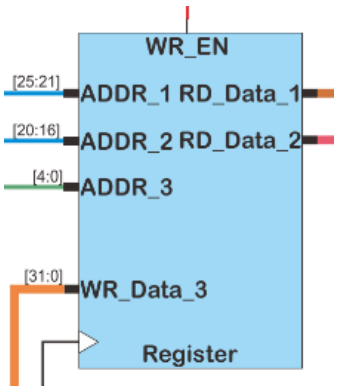
2. Pastikan bahwa sinyal *write enable* dan *read enable* tidak dalam posisi *high* secara bersamaan.
3. Sinkronisasikan sinyal *write enable* dan *read enable* dengan sinyal `wren_a` pada ALTSYNCRAM

pada saat *falling edge* dari *clock*.

4. Buatlah file `dmemory.mif` sebagai *memory initialization file* seperti yang dilakukan pada Tugas 2.
5. Simulasikan secara fungsional dan *timing* desain tersebut setelah melakukan *analysis & synthesis* terlebih dahulu. Perhatikan apakah *waveform* telah sesuai dengan *address* yang dimasukkan.

Tugas 4 : Perancangan Register

Dalam mikroprosesor Single-Cycle MIPS32® yang akan kita realisasikan, terdapat 32 buah *register* yang masing-masing memiliki lebar data 32-bit. *Register* ini memiliki dua buah *port input* untuk memasukkan *address* 32-bit dari data yang akan dibaca dan memiliki satu buah *port input* untuk memasukkan *address* 32-bit tempat data akan ditulis. Selain itu terdapat dua buah *port output* tempat keluarnya data 32-bit yang dibaca dan satu buah *port input* tempat masuknya data 32-bit yang akan ditulis. Terdapat pula *port input* untuk *clock* dan *port input* untuk sinyal untuk mengaktifkan mode tulis (*write enable*). Penggunaan urutan *register* ini sesuai dengan tabel *register* MIPS32® pada landasan teoretis praktikum. Perhatikan bahwa nilai *register* 0 harus tetaplah nol.



1. Implementasikan desain *register* dalam bahasa VHDL dengan deklarasi entitas sebagai berikut. Simpan file ini dengan nama `reg_file.vhd`.

```
ENTITY reg_file IS
  PORT  (
          clock     : IN STD_LOGIC;                     -- clock
          WR_EN     : IN STD_LOGIC;                     -- write enable
          ADDR_1    : IN STD_LOGIC_VECTOR (4 DOWNTO 0); -- Input 1
          ADDR_2    : IN STD_LOGIC_VECTOR (4 DOWNTO 0); -- Input 2
          ADDR_3    : IN STD_LOGIC_VECTOR (4 DOWNTO 0); -- Input 3
          WR_Data_3 : IN STD_LOGIC_VECTOR (31 DOWNTO 0);-- write data
          RD_Data_1 : OUT STD_LOGIC_VECTOR(31 DOWNTO 0);-- read data 1
          RD_Data_2 : OUT STD_LOGIC_VECTOR(31 DOWNTO 0) -- read data 2
        );
END ENTITY;
```

2. Untuk elemen *memory*, gunakan deklarasi elemen *memory* yang sama dengan yang digunakan pada Tugas 1.

```
TYPE ramtype IS ARRAY (31 DOWNTO 0) OF STD_LOGIC_VECTOR(31 DOWNTO 0);
      SIGNAL mem: ramtype;
```

3. Sinkronisasikan proses baca tulis yang terjadi pada *register*. Proses pembacaan data pada *register* berlangsung pada saat *falling edge clock* sedangkan penulisan data pada *register* berlangsung pada saat *rising edge clock*.
4. Simulasikan secara fungsional dan *timing* desain tersebut setelah melakukan *analysis & synthesis* terlebih dahulu. Perhatikan apakah *waveform* telah sesuai dengan *address* yang dimasukkan.

Hasil dan Analisis

Berikut ini adalah pertanyaan-pertanyaan yang dapat membantu analisis pada laporan praktikum. Perhatikan bahwa analisis tidak hanya terbatas pada pertanyaan-pertanyaan di bawah ini.

1. Bagaimana tahap-tahap eksekusi instruksi pada mikroprosesor MIPS32®?
2. Apa perbedaan mikroprosesor MIPS32® dengan mikroprosesor Single-Cycle MIPS32®

yang direalisasikan pada praktikum ini?

3. Apa fungsi register 0 yang selalu berisi nilai 0 pada mikroprosesor MIPS32®?
4. Bagaimana instruksi-instruksi tipe R, tipe J, dan tipe I di-*decode*?
5. Bagaimana *program counter* bekerja dan apa fungsinya?
6. Apa yang terjadi pada *program counter* saat terjadi pencabangan program (*branch*)?
7. Bagaimana mikroprosesor MIPS32® menangani proses perkalian dua buah bilangan 32-bit karena bisa saja perkalian menghasilkan bilangan 64-bit?
8. Apa fungsi Altera® Megafunction ALTSYNCRAM?
9. Apa saja keuntungan menggunakan ALTSYNCRAM bila dibandingkan dengan desain *memory* manual? Manakah yang Anda pilih?
10. Bagaimana cara agar proses pembacaan dan penulisan tidak terjadi pada waktu yang sama?
11. Mengapa diperlukan modifikasi pada lebar data dan lebar *address* pada desain yang kita buat? Adakah solusi untuk mengimplementasikan *memory* lebih besar?
12. Bagaimana hasil simulasi fungsional dan *timing* setiap komponen yang telah direalisasikan?
13. Bagaimana hasil simulasi *instruction memory*, *data memory*, dan *register* dalam simulasi fungsional dan *timing*?

Simpulan

Buatlah simpulan dari percobaan yang Anda lakukan dalam bentuk poin. Simpulan hendaknya menjawab tujuan praktikum ini.

Daftar Pustaka

Patterson, David, dan John Hennessy. *Computer Organization and Design* : *The Hardware/Software Interface*. 2012. Waltham : Elsevier Inc.

PERCOBAAN IV
SYNTHESIZABLE MIPS32® MICROPROCESSOR BAGIAN II : ARITHMETIC AND LOGICAL UNIT (ALU) DAN CONTROL UNIT (CU)

Tujuan Praktikum

- Praktikan memahami arsitektur mikroprosesor MIPS32® beserta *datapath* eksekusinya.
- Praktikan dapat membuat *Arithmetic and Logical Unit* (ALU) dari MIPS32® dalam kode VHDL yang *synthesizable* dan dapat disimulasikan dengan Altera® Quartus® II v9.1sp2.
- Praktikan dapat membuat *Control Unit* (CU) dari MIPS32® dalam kode VHDL yang *synthesizable* dan dapat disimulasikan dengan Altera® Quartus® II v9.1sp2.

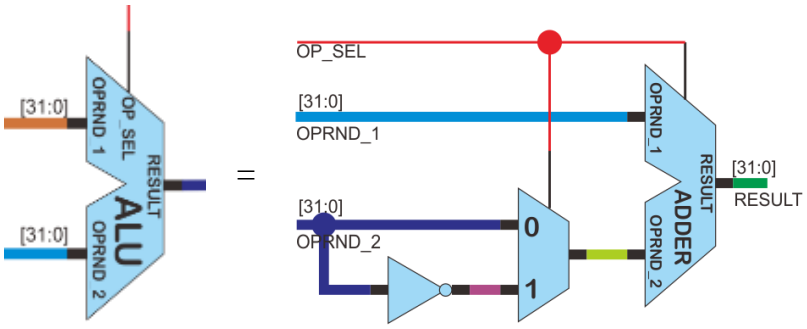
Perangkat Praktikum

- Komputer Desktop / Laptop dengan sistem operasi Microsoft® Windows™ 7/8/8.1
- Altera® Quartus® II v9.1sp2 Web Edition atau Altera® Quartus® II v9.1sp2 Subscription Edition. (Altera® Quartus® II v10 atau yang lebih baru juga dapat digunakan, namun tidak terdapat simulator. Praktikan harus menggunakan Mentor Graphics® ModelSim® untuk melakukan simulasi).
- Notepad++ sebagai teks editor.

Landasan Teoretis Praktikum

Arithmetic and Logical Unit (ALU)

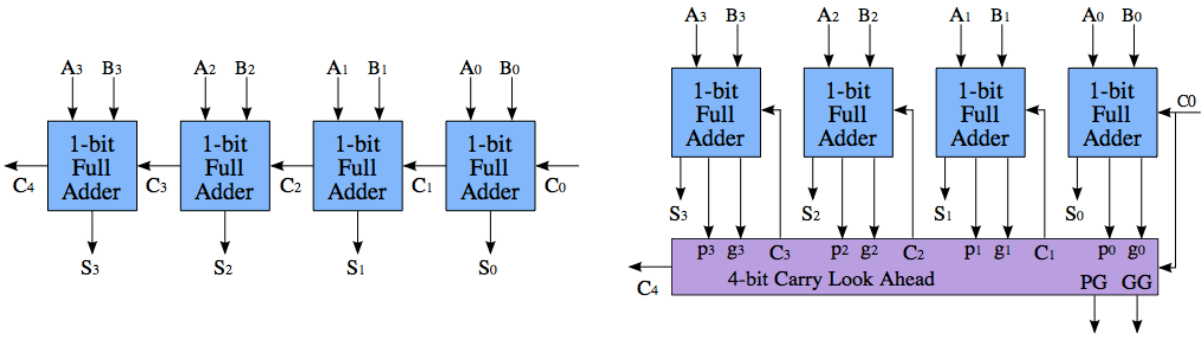
Dalam sistem elektronik digital, sebuah *arithmetic and logical unit* (ALU) adalah rangkaian digital yang berfungsi untuk melakukan perhitungan *integer* dan operasi logika. ALU merupakan blok pembangun dasar dari sebuah mikroprosesor. Mikroprosesor modern meliputi *central processing unit* dan *graphics processing unit* memiliki ALU yang sangat kompleks untuk melakukan perhitungan. Dalam mikroprosesor modern, digunakan sistem representasi bilangan *two's complement*.



Pada mikroprosesor Single-Cycle MIPS32® yang akan kita realisasikan dalam praktikum ini, terdapat *arithmetic and logical unit* (ALU) yang sangat sederhana. ALU ini memiliki lebar data *input* sebesar 32-bit untuk memasukkan dua buah *operand* dan memiliki lebar data *output* sebesar 32-bit untuk mengeluarkan hasil komputasi. ALU ini hanya dapat menangani dua operasi matematika saja yaitu penjumlahan dan pengurangan. Untuk operasi penjumlahan, ALU memanfaatkan blok *adder*. Sedangkan untuk operasi pengurangan, ALU memanfaatkan sifat bilangan *two's complement*. Dengan demikian, pengurangan merupakan penjumlahan dengan bilangan negatif. Oleh karena itu, *operand*

kedua dapat diubah menjadi bilangan negatif dengan memanfaatkan prinsip *two's complement* yaitu rumus $-X = \sim X + 1$. Setelah itu, *adder* akan menjumlahkan kedua *operand* tersebut seperti biasa. Untuk memilih operasi penjumlahan dan pengurangan, terdapat *2-to-1 multiplexer* yang akan memilih arah *operand* kedua berasal. Untuk penjumlahan, selektor *multiplexer* bernilai 0 sedangkan untuk pengurangan selektor *multiplexer* bernilai 1. Selain itu, *carry-in* untuk *adder* juga ditentukan dari operasi yang dilakukan. Untuk penjumlahan, *carry-in* bernilai 0 sedangkan untuk pengurangan, *carry-in* untuk bernilai 1. Dengan demikian, kedua sinyal ini (*carry-in* dan selektor multiplexer) dapat dihubungkan menjadi satu sinyal yaitu `OP_SEL`. Untuk melakukan *inverting operand* kedua, digunakan gerbang NOT dengan lebar data 32-bit.

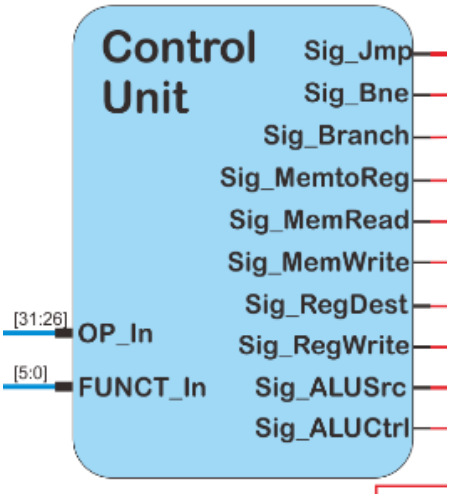
Untuk mendesain *adder*, ada beberapa arsitektur *adder* yang dapat dipilih. Masing-masing arsitektur memiliki kelebihan dan kekurangan yang dapat ditinjau dari segi kecepatan, konsumsi daya, dan konsumsi area. Dua contoh arsitektur *adder* adalah *ripple carry adder* dan *carry-lookahead adder*. *Ripple carry adder* merupakan *adder* yang relatif sederhana. Kelemahan *adder* ini adalah dari segi kecepatan karena setiap bit tidak dapat dijumlahkan secara bersamaan. Tahap *adder* yang lebih tinggi harus menunggu *carry* yang dibawa dari tahap *adder* yang lebih rendah. Pada *carry-lookahead adder*, setiap tahap *adder* dapat menghitung *carry* yang dia terima sehingga tidak perlu menunggu propagasi *carry* dari tahap sebelumnya. Kelebihan *carry-lookahead adder* harus dibayar dengan penambahan rangkaian logika yang akan mengkonsumsi luas area.



Ripple carry adder (kiri) dan *Carry-lookahead adder* (kanan). Sumber: Wikipedia

Control Unit (CU)

Control Unit (CU) merupakan komponen dari sebuah mikroprosesor yang berfungsi untuk mengarahkan operasi-operasi yang dilakukan oleh mikroprosesor tersebut. CU mengatur komunikasi dan koordinasi antarkomponen mikroprosesor menggunakan sinyal-sinyal kontrol. CU juga membaca dan menerjemahkan instruksi-instruksi yang diproses untuk menentukan urutan pemrosesan data.



Pada mikroprosesor Single-Cycle MIPS32® yang akan kita realisasikan dalam praktikum ini, terdapat *control unit* (CU) yang sangat sederhana. CU menerima `opcode` dan `funct` dari instruksi setelah di-*decode* untuk menentukan nilai dari sinyal-sinyal kontrol yang dikeluarkan. Terdapat sepuluh sinyal

kontrol yang keluar dari CU ini yang dijelaskan sebagai berikut.

Nama Sinyal	Lebar	Tujuan	Fungsi
`Sig_Jmp`	2 bit	*2-to-1 Mux* pada *Program Counter*	Menunjukkan adanya instruksi *jump* sehingga *program counter* dapat diset sesuai dengan *address* hasil kalkulasi.
`Sig_Bne`	1 bit	Gerbang OR 2 *Input*	Menunjukkan adanya instruksi `bne` untuk memilih hasil pencabangan pada *program counter*.
`Sig_Branch`	1 bit	Gerbang OR 2 *Input*	Menunjukkan adanya instruksi `beq` untuk memilih hasil pencabangan pada *program counter*.
`Sig_MemtoReg`	1 bit	*2-to-1 Mux* pada *Data Memory*	Memilih data untuk *writeback*, apakah berasal dari *data memory* atau ALU.
`Sig_MemRead`	1 bit	*Data Memory*	Sinyal yang mengaktifkan operasi baca pada *data memory*.
`Sig_MemWrite`	1 bit	*Data Memory*	Sinyal yang mengaktifkan operasi tulis pada *data memory*.
`Sig_RegDest`	2 bit	*4-to-1 Mux* pada *Register*	Memilih *register* yang akan dijadikan sebagai *destination register*.
`Sig_RegWrite`	1 bit	*Register*	Sinyal yang mengaktifkan operasi tulis pada *register*.
`Sig_ALUSrc`	2 bit	*4-to-1 Mux* pada ALU	Memilih data *operand* kedua yang akan masuk ke ALU, apakah dari *register* atau dari *immediate*.
`Sig_ALUCtrl`	1 bit	ALU	Memilih operasi yang akan dilakukan pada ALU apakah penjumlahan atau pengurangan.

Terdapat sembilan instruksi yang dapat dieksekusi oleh mikroprosesor Single-Cycle MIPS32® yang akan kita realisasikan dalam praktikum ini. Kesembilan instruksi tersebut akan menentukan nilai sinyal yang dikeluarkan oleh *control unit* karena setiap instruksi membutuhkan penanganan dan aliran data yang berbeda-beda. Berikut ini tabel nilai sinyal *control unit* untuk setiap instruksi yang dapat dieksekusi.

Instruksi	Tipe	Sig_Jmp	Sig_Bne	Sig_Branch	Sig_MemtoReg	Sig_MemRead
add	R	00	0	0	0	0
sub	R	00	0	0	0	0
beq	I	00	0	1	0	0
bne	I	00	1	0	0	0
addi	I	00	0	0	0	0
lw	I	00	0	0	1	1
sw	I	00	0	0	0	0
jmp	J	01	0	0	0	0
nop	-	00	0	0	0	0

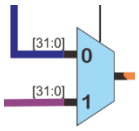
Instruksi	Tipe	Sig_MemWrite	Sig_RegDest	Sig_RegWrite	Sig_ALUSrc	Sig_ALUCtrl
add	R	0	01	1	0	00
sub	R	0	01	1	0	01
beq	I	0	--	0	-	--
bne	I	0	--	0	-	--
addi	I	0	00	1	1	00
lw	I	0	00	1	1	00
sw	I	1	00	0	1	00
jmp	J	0	--	0	-	--
nop	-	0	00	0	0	00

Untuk mengatur sinyal-sinyal kontrol tersebut, *control unit* mendeteksi setiap instruksi menggunakan

`opcode` dan `funct`.

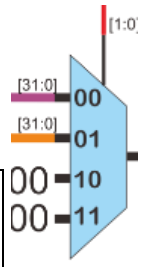
Tugas Pendahuluan

1. Buatlah komponen *2-to-1 multiplexer* dengan lebar data 32-bit dalam bahasa VHDL lalu simulasikan dalam simulasi fungsional dan *timing*. Jangan lupa untuk melampirkan kode VHDL dan hasil simulasinya dalam lembar jawaban.



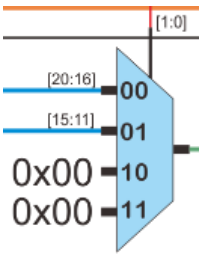

```
ENTITY mux_2to1_32bit IS
   PORT  (
            D1 : IN   std_logic_vector (31 DOWNTO 0); -- Data Input 1
            D2 : IN   std_logic_vector (31 DOWNTO 0); -- Data Input 2
            Y  : OUT  std_logic_vector (31 DOWNTO 0); -- Selected Data
            S  : IN   std_logic                       -- Selector
         );
END mux_2to1_32bit;
```

2. Buatlah komponen *4-to-1 multiplexer* dengan lebar data 32-bit dalam bahasa VHDL lalu simulasikan dalam simulasi fungsional dan *timing*. Jangan lupa untuk melampirkan kode VHDL dan hasil simulasinya dalam lembar jawaban.




```
ENTITY mux_4to1_32bit IS
   PORT  (
            D1 : IN   std_logic_vector (31 DOWNTO 0); -- Data
Input 1
            D2 : IN   std_logic_vector (31 DOWNTO 0); -- Data
Input 2
            D3 : IN   std_logic_vector (31 DOWNTO 0); -- Data
Input 3
            D4 : IN   std_logic_vector (31 DOWNTO 0); -- Data
Input 4
            Y  : OUT  std_logic_vector (31 DOWNTO 0); --
Selected Data
            S  : IN   std_logic_vector (1 DOWNTO 0);  --
Selector
         );
END mux_4to1_32bit;
```

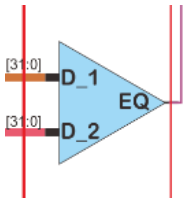
3. Buatlah komponen *4-to-1 multiplexer* dengan lebar data 5-bit dalam bahasa VHDL lalu simulasikan dalam simulasi fungsional dan *timing*. Jangan lupa untuk melampirkan kode VHDL dan hasil simulasinya dalam lembar jawaban.




```
ENTITY mux_4to1_5bit IS
   PORT  (
            D1 : IN   std_logic_vector  (4  DOWNTO  0);  --
Data Input 1
            D2 : IN   std_logic_vector  (4  DOWNTO  0);  --
Data Input 2
            D3 : IN   std_logic_vector  (4  DOWNTO  0);  --
Data Input 3
            D4 : IN   std_logic_vector  (4  DOWNTO  0);  --
Data Input 4
            Y  : OUT  std_logic_vector  (4  DOWNTO  0);  --
Selected Data
```

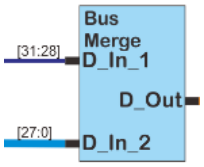
```
            S  : IN    std_logic_vector (1 DOWNTO 0); -- Selector
        );
END mux_4to1_5bit;
```

4. Buatlah komponen komparator dengan dua buah *input* dengan lebar data 32-bit dalam bahasa VHDL lalu simulasikan dalam simulasi fungsional dan *timing*. Komparator akan menghasilkan *output high* saat kedua *input* sama. Komparator akan menghasilkan *output low* saat kedua *input* berbeda. Jangan lupa untuk melampirkan kode VHDL dan hasil simulasinya dalam lembar jawaban.




```
ENTITY comparator IS
   PORT  (
            D_1     : IN   STD_LOGIC_VECTOR (31 DOWNTO 0);
            D_2     : IN   STD_LOGIC_VECTOR (31 DOWNTO 0);
            EQ      : OUT  STD_LOGIC  -- Hasil Perbandingan EQ
        );
END comparator;
```

5. Buatlah komponen *bus merging* yang menerima dua buah *input* dengan lebar 4 bit dan 28 bit untuk digabung menjadi satu buah *output* dengan lebar 32-bit dalam bahasa VHDL lalu simulasikan dalam simulasi fungsional dan *timing*. Jangan lupa untuk melampirkan kode VHDL dan hasil simulasinya dalam lembar jawaban.



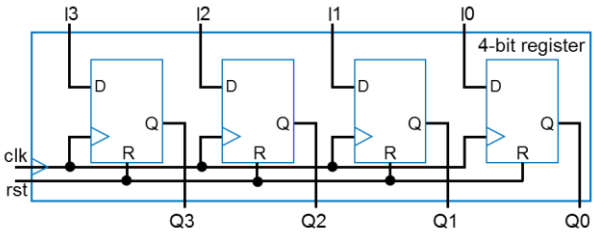

```
ENTITY bus_merger IS
   PORT  (
            DATA_IN1  : IN    STD_LOGIC_VECTOR (3 DOWNTO 0);
            DATA_IN2  : IN    STD_LOGIC_VECTOR (27 DOWNTO 0);
            DATA_OUT  : OUT   STD_LOGIC_VECTOR (31 DOWNTO 0)
   );
END bus_merger;
```

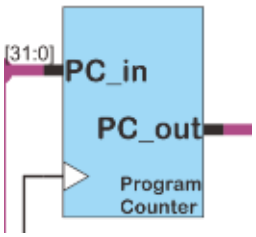
Metodologi Praktikum

Sebelum melakukan praktikum, buatlah folder kerja sesuai dengan Nama, NIM, tanggal, dan modul praktikum (lihat petunjuk teknis pelaksanaan praktikum). Kerjakan masing-masing tugas di dalam folder kerja yang sesuai dengan nomor tugas tersebut. Kumpulkan tugas pendahuluan terlebih dahulu sebelum melaksanakan praktikum.

Tugas 1 : Perancangan *Program Counter*

Buatlah *program counter* yang berupa satu buah *register* dengan lebar 32-bit dalam bahasa VHDL. Sebuah *register* dengan lebar data 1-bit dapat direalisasikan menggunakan satu buah D Flip-flop. Dengan demikian, untuk merealisasikan satu buah *register* dengan lebar 32-bit diperlukan 32 buah D Flip-flop. Gunakan D Flip-flop yang menerima data saat *rising edge clock*. Simulasikan *program counter* ini secara fungsional dan *timing*.



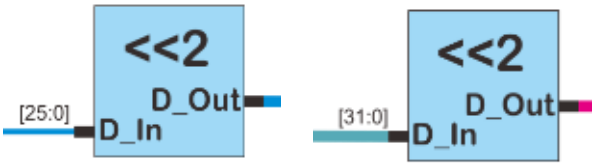



```
ENTITY program_counter IS
  PORT  (
            clk     : IN   STD_LOGIC;
            PC_in   : IN   STD_LOGIC_VECTOR (31 DOWNTO 0);
            PC_out  : OUT  STD_LOGIC_VECTOR (31 DOWNTO 0)
        );
END program_counter;
```

Tugas 2 : Perancangan *Left Shifter* Dua Kali

Buatlah dua jenis *left shifter* dua kali dalam bahasa VHDL dengan spesifikasi sebagai berikut.

- *Left shifter* jenis pertama memiliki *input* data dengan lebar 32-bit dan *output* data dengan lebar 32-bit.
- *Left shifter* jenis kedua memiliki *input* data dengan lebar 26-bit dan *output* data dengan lebar 28-bit.



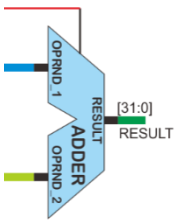

```
ENTITY lshift_32_32 IS
    PORT (
            D_IN        : IN STD_LOGIC_VECTOR (31 DOWNTO 0); -- Input 32-bit;
            D_OUT       : OUT STD_LOGIC_VECTOR (31 DOWNTO 0) -- Output 32-bit;
        );
END lshift_32_32;
```

```
ENTITY lshift_26_28 IS
    PORT (
            D_IN        : IN STD_LOGIC_VECTOR (25 DOWNTO 0); -- Input 26-bit;
            D_OUT       : OUT STD_LOGIC_VECTOR (27 DOWNTO 0) -- Output 28-bit;
        );
END lshift_26_28;
```

Simulasikan kedua *left shifter* ini secara fungsional dan *timing*.

Tugas 3 : Perancangan *Carry-Lookahead Adder* 32-bit

Buatlah sebuah *carry-lookahead adder* dalam bahasa VHDL yang mampu menjumlahakan dua buah *input* dengan lebar data masing-masing 32-bit dan mengeluarkan hasil penjumlahan dalam bentuk *output* dengan lebar data 32-bit. *Adder* ini hendaknya dapat menerima *carry-in* sebesar satu bit untuk mengeset *carry-in* pada tahap pertama dari *adder* ini. Simulasikan *carry-lookahead adder* ini secara fungsional dan *timing*.



```
ENTITY cla_32 IS
  PORT  (
          OPRND_1  : IN  STD_LOGIC_VECTOR(31 DOWNTO 0);      -- Operand 1
          OPRND_2  : IN  STD_LOGIC_VECTOR (31 DOWNTO 0);     -- Operand 2
          C_IN     : IN  STD_LOGIC;                          -- Carry In
          RESULT   : OUT STD_LOGIC_VECTOR (31 DOWNTO 0);     -- Result
          C_OUT    : OUT STD_LOGIC                           -- Overflow
        );

END cla_32;
```

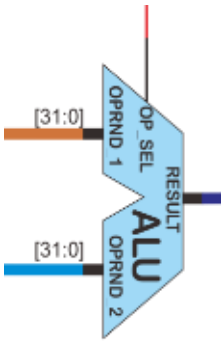
Tugas 4 : Sign Extender

Buatlah sebuah *sign extender* dalam bahasa VHDL yang menerima data *input* sebesar 16-bit dan mengeluarkan data *output* sebesar 32-bit. Prinsip *sign extension* mengikuti aturan bilangan *two's complement* dengan tetap mempertahankan nilai bit MSB. Simulasikan *sign extender* ini secara fungsional dan *timing*.


```
ENTITY sign_extender IS
  PORT  (
          D_In    :IN    std_logic_vector (15 DOWNTO 0); -- Data Input 1
          D_Out   :OUT   std_logic_vector (31 DOWNTO 0)  -- Data Input 2
        );
END sign_extender;
```

Tugas 5 : *Arithmetic and Logical Unit* (ALU)

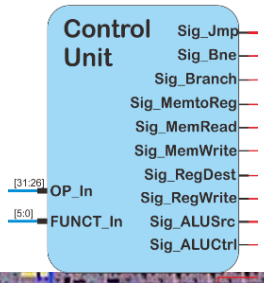
Buatlah sebuah *Arithmetic and Logical Unit* (ALU) dalam bahasa VHDL dengan memanfaatkan *carry-lookahead adder* yang telah dibuat pada tugas 3. ALU menerima dua buah *operand* sebagai *input* dengan masing-masing memiliki lebar data 32-bit. ALU akan memberikan data hasil perhitungan melalui *output* dengan lebar 32-bit. ALU juga memiliki selektor untuk memilih operasi yang akan dilakukan, apakah penjumlahan atau pengurangan. Apabila selektor bernilai `0x00`, maka operasi yang dilakukan adalah penjumlahan. Apabila selektor bernilai `0x01`, maka operasi yang dilakukan adalah pengurangan. Simulasikan ALU ini secara fungsional dan *timing*.




```
ENTITY ALU IS
  PORT  (
          OPRND_1  : IN   std_logic_vector (31 DOWNTO 0); -- Data Input 1
          OPRND_2  : IN   std_logic_vector (31 DOWNTO 0); -- Data Input 2
          OP_SEL   : IN   std_logic_vector (1 DOWNTO 0);  -- Operation Select
          RESULT   : OUT  std_logic_vector (31 DOWNTO 0)  -- Data Output
        );
END ALU;
```

Tugas 6 : *Control Unit* (CU)

Buatlah sebuah *Control Unit* (CU) dalam bahasa VHDL untuk mikroprosesor Single-Cycle MIPS32® Anda. Untuk melakukan *assignment* sinyal kontrol terhadap opcode dan funct, Anda dapat memanfaatkan *concurent signal assignment* (CSA) baik



conditional CSA atau *selected* CSA. Anda juga dapat melakukan *assignment* sinyal menggunakan *construct* IF-THEN-ELSE atau *construct* CASE dalam sebuah PROCESS.

Implementasikan masing-masing kondisi sinyal kontrol sesuai dengan tabel pada landasan teoretis praktikum ini. Simulasikan CU ini secara fungsional dan *timing*.

```
ENTITY cu IS
  PORT  (
           OP_In         : IN STD_LOGIC_VECTOR (5 DOWNTO 0);
           FUNCT_In      : IN STD_LOGIC_VECTOR (5 DOWNTO 0);
           Sig_Jmp       : OUT STD_LOGIC_VECTOR (1 DOWNTO 0);
           Sig_Bne       : OUT STD_LOGIC;
           Sig_Branch    : OUT STD_LOGIC;
           Sig_MemtoReg  : OUT STD_LOGIC;
           Sig_MemRead   : OUT STD_LOGIC;
           Sig_MemWrite  : OUT STD_LOGIC;
           Sig_RegDest   : OUT STD_LOGIC_VECTOR (1 DOWNTO 0);
           Sig_RegWrite  : OUT STD_LOGIC;
           Sig_ALUSrc    : OUT STD_LOGIC;
           Sig_ALUCtrl   : OUT STD_LOGIC_VECTOR (1 DOWNTO 0)
       );
END cu;
```

Hasil dan Analisis

Berikut ini adalah pertanyaan-pertanyaan yang dapat membantu analisis pada laporan praktikum. Perhatikan bahwa analisis tidak hanya terbatas pada pertanyaan-pertanyaan di bawah ini.

1. Bagaimana mikroprosesor MIPS32® mendeteksi jenis-jenis instruksi yang diberikan?
2. Bagaimana sebuah *control unit* mengatur jalannya aliran data (*datapath*) pada mikroprosesor MIPS32®?
3. Apakah fungsi *program counter*? Bagaimana cara ia bekerja?
4. Apakah yang terjadi pada *program counter* saat terdapat instruksi *branch* (`beq` atau `bne`)?
5. Mengapa diperlukan *sign extender* dalam desain Single-Cycle MIPS32® di praktikum ini?
6. Mengapa diperlukan *bus merger* dalam desain Single-Cycle MIPS32® di praktikum ini?
7. Berapa maksimum nilai konstanta pada operasi matematika yang melibatkan nilai konstan pada mikroprosesor MIPS32®?
8. Mengapa kecepatan penjumlahan dari *carry-lookahead adder* lebih baik dibanding *ripple carry adder*?
9. Apa fungsi komparator dalam menentukan ada atau tidaknya *branch*?
10. Bagaimana hasil simulasi fungsional dan *timing* setiap komponen yang telah direalisasikan?
11. Bagaimana sebuah *control unit* bekerja? Apa saja komponen-komponen mikroprosesor MIPS32® yang diatur oleh *control unit* ini?

Simpulan

Buatlah simpulan dari percobaan yang Anda lakukan dalam bentuk poin. Simpulan hendaknya menjawab tujuan praktikum ini.

Daftar Pustaka

Patterson, David, dan John Hennessy. *Computer Organization and Design : The Hardware/Software Interface*. 2012. Waltham : Elsevier Inc.

PERCOBAAN V
SYNTHESIZABLE MIPS32® MICROPROCESSOR BAGIAN III : *TOP LEVEL DESIGN* DAN *TESTBENCH*

Tujuan Praktikum

- Praktikan memahami arsitektur mikroprosesor MIPS32® beserta *datapath* eksekusinya.
- Praktikan memahami *instruction set* dari MIPS32® dan dapat membuat program sederhana dalam bahasa *assembly* yang dapat dieksekusi pada MIPS32®.
- Praktikan dapat melakukan simulasi eksekusi program MIPS32® pada program simulasi SPIM dan memahami cara setiap instruksi dieksekusi.
- Praktikan dapat menggabungkan komponen-komponen desain yang telah dibuat dari praktikum sebelumnya dalam kode VHDL untuk membuat *top level design* dari mikroprosesor Single-Cycle MIPS32® yang *synthesizable* dan dapat disimulasikan dengan Altera® Quartus® II v9.1sp2.
- Praktikan dapat membuat *testbench* untuk menguji desain mikroprosesor Single-Cycle MIPS32® dalam kode VHDL dan dapat disimulasikan dengan Altera® Quartus® II v9.1sp2.

Perangkat Praktikum

- Komputer Desktop / Laptop dengan sistem operasi Microsoft® Windows™ 7/8/8.1
- Altera® Quartus® II v9.1sp2 Web Edition atau Altera® Quartus® II v9.1sp2 Subscription Edition. (Altera® Quartus® II v10 atau yang lebih baru juga dapat digunakan, namun tidak terdapat simulator. Praktikan harus menggunakan Mentor Graphics® ModelSim® untuk melakukan simulasi).
- PCSpim sebagai simulator MIPS32® (untuk Microsoft® Windows™ 8/8.1, diperlukan mengunduh dan memasang Microsoft® .Net Framework 3.5 terlebih dahulu).
- Notepad++ sebagai teks editor.

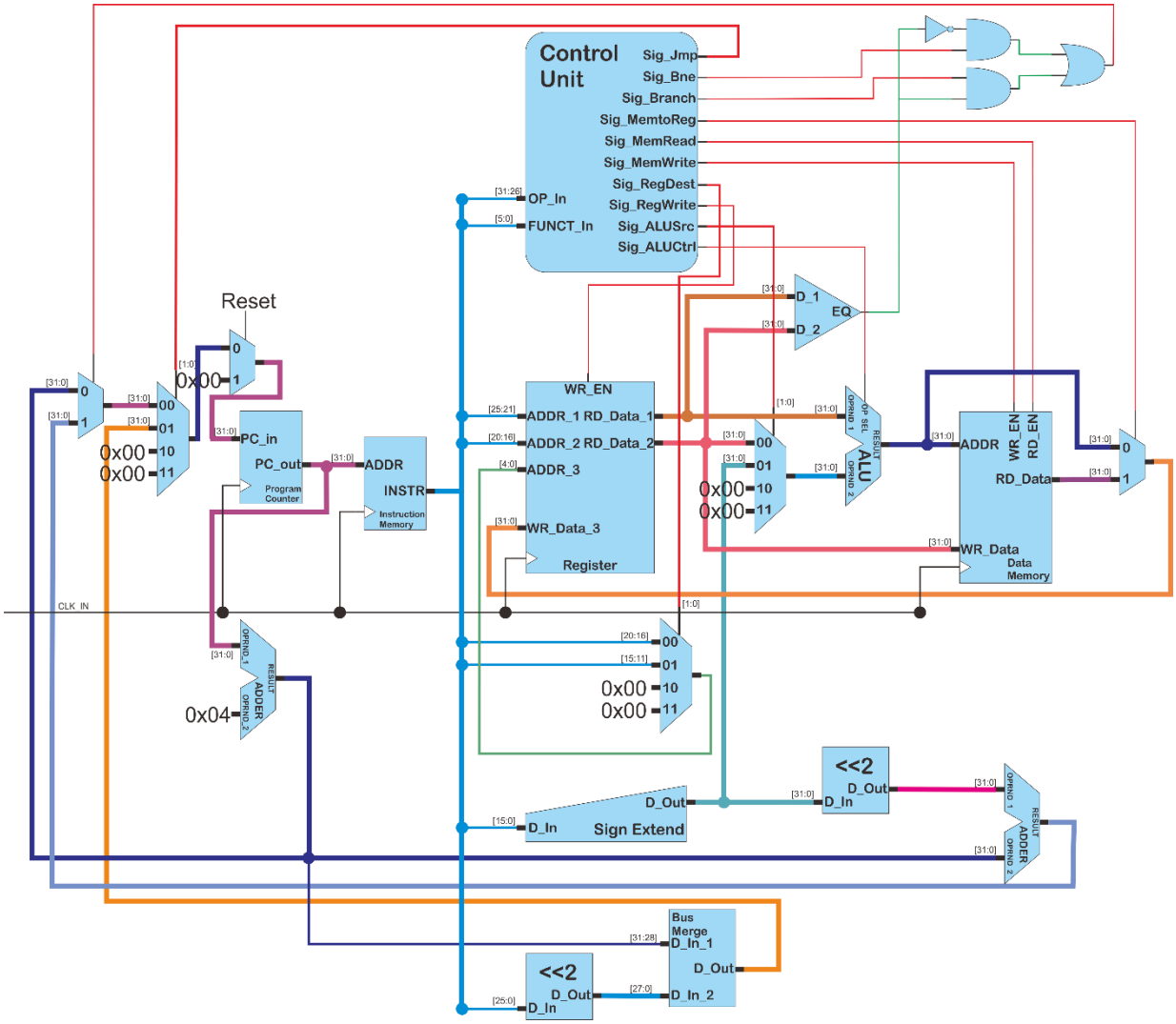
Landasan Teoretis Praktikum

Datapath dan *Control*

Dalam desain mikroprosesor Single-Cycle MIPS32®, rangkaian digital dapat dikelompokkan menjadi dua macam, yaitu *datapath* (jalur data) dan *control* (kontrol). *Datapath* merupakan komponen dari mikroprosesor yang melakukan operasi aritmetik serta melakukan penyimpanan data. Dalam *datapath* pula kelima tahap pemrosesan instruksi meliputi *instruction fetch*, *instruction decode*, *execute*, *memory access*, dan *write back* dilaksanakan. Sedangkan *control* merupakan komponen dari mikroprosesor yang mengatur *datapath* berdasarkan instruksi yang sedang dieksekusi.

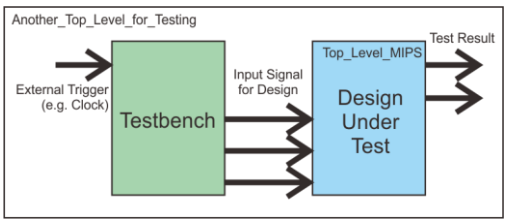
Bagian kontrol telah direpresentasikan oleh *control unit* yang telah kita desain pada praktikum sebelumnya. Untuk bagian *datapath*, kita perlu menggabungkan komponen-komponen yang telah kita buat meliputi *multiplexer*, ALU, *register*, *data memory*, *instruction memory*, dan sebagainya untuk membentuk sebuah jalur yang dapat dilewati oleh data. Dengan demikian, *control* dan *datapath* tidak dapat dipisahkan dalam desain sebuah mikroprosesor. Datapath dan *control* akan membentuk sebuah

desain mikroprosesor Single-Cycle MIPS32® yang disebut sebuah *top-level design* (desain paling atas). *Top-level design* pada umumnya hanya berisi *port mapping* dari satu komponen ke komponen lain.



Menggunakan *Testbench* untuk Simulasi

Dalam melakukan simulasi sebuah rangkaian digital, biasanya kita memberikan sinyal *input* secara manual melalui sebuah *waveform editor*. Cara ini boleh dibilang sederhana namun akan sangat tidak efektif apabila kita melakukan simulasi secara berulang-ulang. Cara lain untuk melakukan simulasi suatu rangkaian digital tanpa memberikan *input* satu per satu menggunakan *waveform editor* adalah menggunakan *testbench*.



Pada dasarnya, *testbench* terdiri atas kode VHDL atau Verilog HDL, tergantung pada implementasi. *Testbench* sendiri dapat berisi sebuah desain yang menyimpan nilai-nilai sinyal *input* yang harus diberikan kepada desain yang sedang diuji (*design under test*). Kemudian, *testbench* ini akan

mengeluarkan masing-masing *input* yang harus diberikan kepada desain yang sedang diuji berdasarkan suatu *trigger*, misalnya *clock*.

Tugas Pendahuluan

1. Diberikan sebuah kode program dalam bahasa C sebagai berikut.

```
int sum = 0;
int i = 0;
while (i != 10)
{
        sum = sum +1;
        i = i + 1;
}
```

   a. Ubahlah kode dalam bahasa C tersebut ke dalam bahasa *assembly* MIPS32® menggunakan instruksi-instruksi yang dapat dimengerti oleh mikroprosesor Single-Cycle MIPS32® yang diimplementasikan dalam praktikum ini. Jangan lupa untuk menggunakan instruksi `nop` setelah instruksi yang berkaitan dengan *branching* dilaksanakan.

   b. Ubahlah program bahasa *assembly* yang telah dibuat pada (a) menjadi file objek yang berisi urutan bilangan biner untuk masing-masing instruksi. Contohnya, apabila terdapat instruksi dalam bahasa *assembly* `addi $s1, $s1, 1`, maka kode biner yang bersesuaian adalah `0x22310001`.

2. Diberikan urutan instruksi dalam bahasa *assembly* MIPS32® sebagai berikut.

```
addi $s0, $0, 19
addi $s1, $0, 21
bne $s2, $s3, 8
nop
sub $s3, $s0, $s1
addi $s3, $s3, 0
addi $s4, $s0, 4
sw $s1, ($s4)
lw $s5, ($s4)
add $s5, $s5, $0
j 00000000
nop
```

   a. Jelaskan maksud dari bahasa *assembly* tersebut.

   b. Ubahlah program bahasa *assembly* menjadi file objek yang berisi kode biner yang merepresentasikan masing-masing instruksi.

3. Apa yang dimaksud *datapath*? Bagaimana hubungan antara *datapath* dengan *control*?

Metodologi Praktikum

Sebelum melakukan praktikum, buatlah folder kerja sesuai dengan Nama, NIM, tanggal, dan modul praktikum (lihat petunjuk teknis pelaksanaan praktikum). Kerjakan masing-masing tugas di dalam folder kerja yang sesuai dengan nomor tugas tersebut. Kumpulkan tugas pendahuluan terlebih dahulu sebelum melaksanakan praktikum.

Tugas 1 : Implementasi *Top-Level Design* MIPS32®

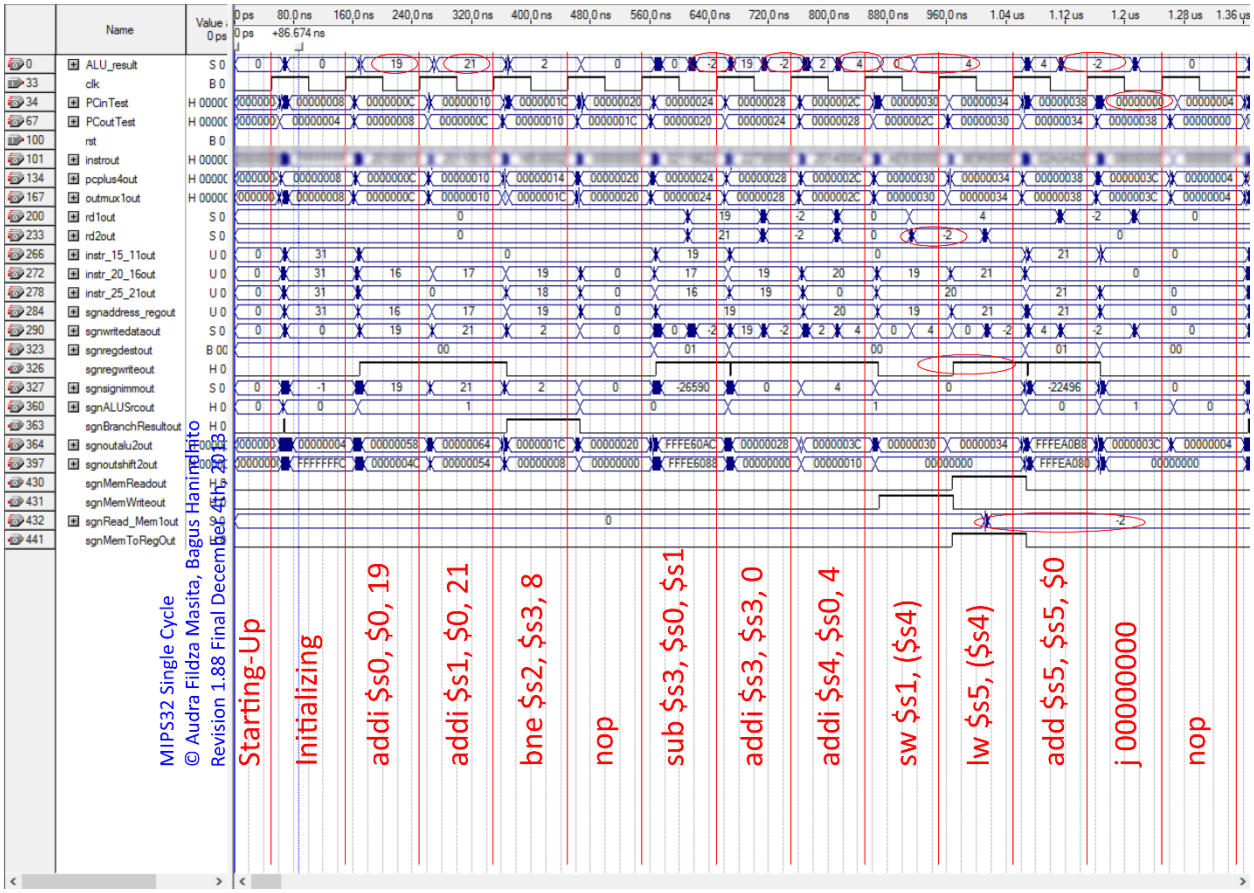
1. Gabungkan seluruh komponen yang telah diimplementasikan dan telah diuji menjadi sebuah desain yang besar untuk merealisasikan *top-level design*.

2. Hubungkan masing-masing komponen dengan sinyal yang sesuai.

3. Lakukan simulasi fungsional dan *timing* pada *top-level design* menggunakan instruksi-instruksi yang telah dikerjakan pada tugas pendahuluan nomor 1. Amati hasilnya.
4. Lakukan simulasi fungsional dan *timing* pada *top-level design* menggunakan instruksi-instruksi yang telah dikerjakan pada tugas pendahuluan nomor 2. Amati hasilnya.

Tugas 2 : Pengujian Menggunakan TestBench

Buatlah sebuah *testbench* yang digunakan untuk melakukan pengujian *top-level design* yang telah kita buat. Lakukan simulasi fungsional dan *timing* terhadap *testbench* ini.



Hasil dan Analisis

Berikut ini adalah pertanyaan-pertanyaan yang dapat membantu analisis pada laporan praktikum. Perhatikan bahwa analisis tidak hanya terbatas pada pertanyaan-pertanyaan di bawah ini.

1. Bagaimana hasil simulasi fungsional terhadap *top-level design*?
2. Bagaimana hasil simulasi *timing* terhadap *top-level design*?
3. Bagaimana *datapath* eksekusi instruksi `add` dan `sub` pada mikroprosesor Single-Cycle MIPS32®? Gambarkan diagram aliran datanya pada diagram blok komponen Single-Cycle MIPS32®!
4. Bagaimana *datapath* eksekusi instruksi `addi` pada mikroprosesor Single-Cycle MIPS32®? Gambarkan diagram aliran datanya pada diagram blok komponen Single-Cycle MIPS32®!
5. Bagaimana *datapath* eksekusi instruksi `beq` dan `bne` pada mikroprosesor Single-Cycle MIPS32®? Gambarkan diagram aliran datanya pada diagram blok komponen Single-Cycle MIPS32®!

6. Bagaimana *datapath* eksekusi instruksi `j` (jump) pada mikroprosesor Single-Cycle MIPS32®? Gambarkan diagram aliran datanya pada diagram blok komponen Single-Cycle MIPS32®!
7. Bagaimana *datapath* eksekusi instruksi `lw` pada mikroprosesor Single-Cycle MIPS32®? Gambarkan diagram aliran datanya pada diagram blok komponen Single-Cycle MIPS32®!
8. Bagaimana *datapath* eksekusi instruksi `sw` pada mikroprosesor Single-Cycle MIPS32®? Gambarkan diagram aliran datanya pada diagram blok komponen Single-Cycle MIPS32®!
9. Mengapa setiap pemanggilan instruksi yang berpotensi *branch*, diperlukan instruksi `nop` sebagai penjeda?
10. Bagaimana hasil simulasi *timing* dan fungsional keseluruhan desain ini?
11. Bagaimana sebuah *testbench* bekerja untuk menguji sebuah desain?

Simpulan

Buatlah simpulan dari percobaan yang Anda lakukan dalam bentuk poin. Simpulan hendaknya menjawab tujuan praktikum ini.

Daftar Pustaka

Patterson, David, dan John Hennessy. *Computer Organization and Design* : *The Hardware/Software Interface*. 2012. Waltham : Elsevier Inc.

LAMPIRAN I : INSTALASI GCC PADA MICROSOFT® WINDOWS™

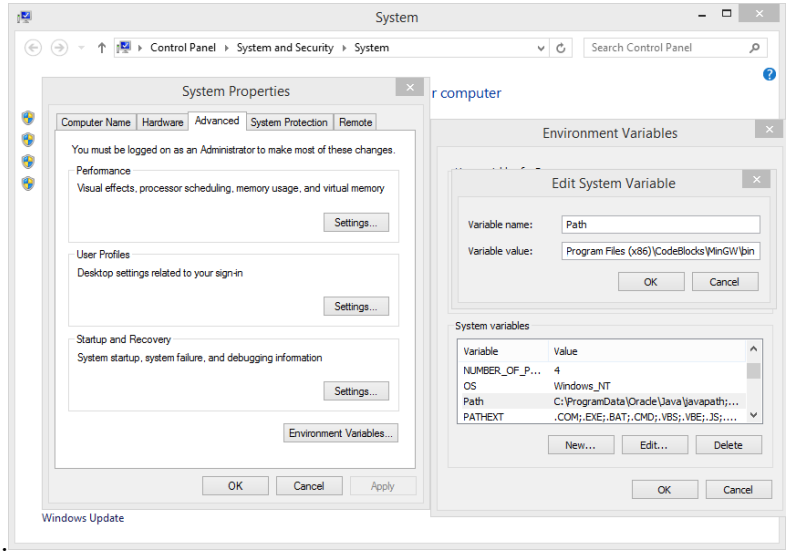
Berikut ini dijelaskan tata cara melakukan instalasi GCC pada komputer bersistem operasi Microsoft® Windows™ 7/8/8.1/10.

1. Unduh perangkat lunak CodeBlocks versi 16.01 (atau yang lebih baru) yang terintegrasi dengan MinGW. Biasanya nama file yang diberikan adalah `codeblocks-16.01mingw-setup.exe`.

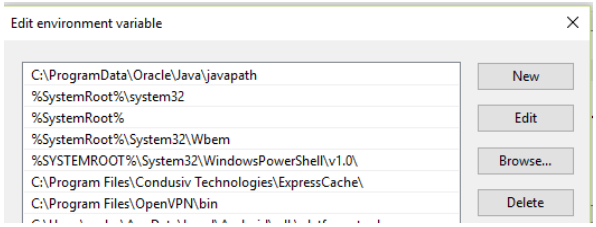
2. Lakukanlah instalasi pada perangkat lunak CodeBlocks yang telah Anda unduh. Gunakan *default settings* apabila langkah ini cukup membingungkan.

3. Periksa isi folder `C:\Program Files (x86)\CodeBlocks\MinGW\bin` (untuk Microsoft® Windows™ 64-bit) atau `C:\Program Files\CodeBlocks\MinGW\bin` (untuk Microsoft® Windows™ 32-bit). Pastikan terdapat file `gcc.exe`, `mingw32-make.exe`, dan `objdump.exe`.

4. Tambahkan *environment variable* pada kotak isian PATH. Isikan alamat lokasi file MinGW berada.



5. Untuk Windows 10, tampilanya sedikit berbeda. Setelah *click* Edit, kemudian akan muncul jendela *Edit environment variable*. Pada jendela ini *click* New, kemudian baru masukkan alamat lokasi file MinGW berada.

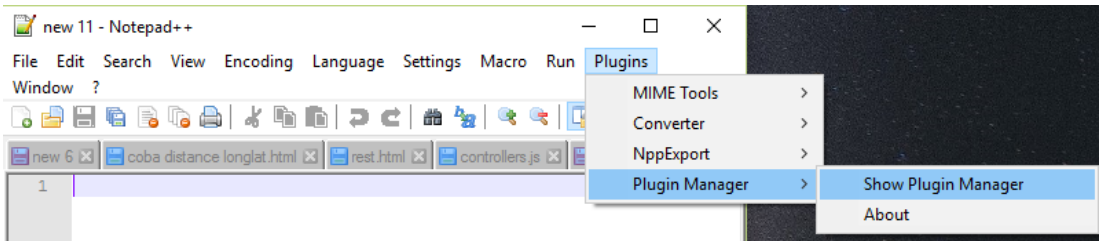


6. Hidupkan ulang komputer Anda (*restart*). Setelah *restart*, seharusnya file-file sudah dapat diakses dengan normal melalui *command prompt*.

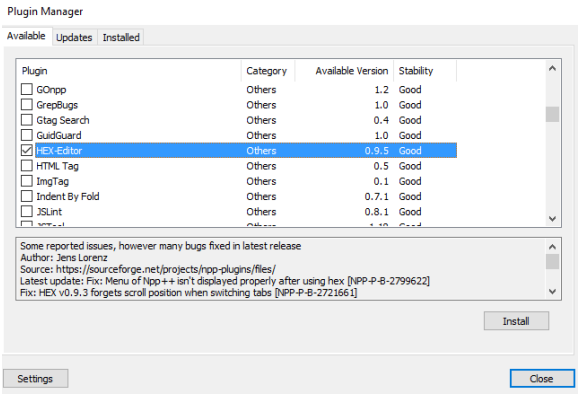
LAMPIRAN II : INSTALASI PLUGIN HEX-EDITOR PADA NOTEPAD ++

Berikut ini dijelaskan tata cara melakukan instalasi *plugin* Hex-Editor pada program Notepad ++.

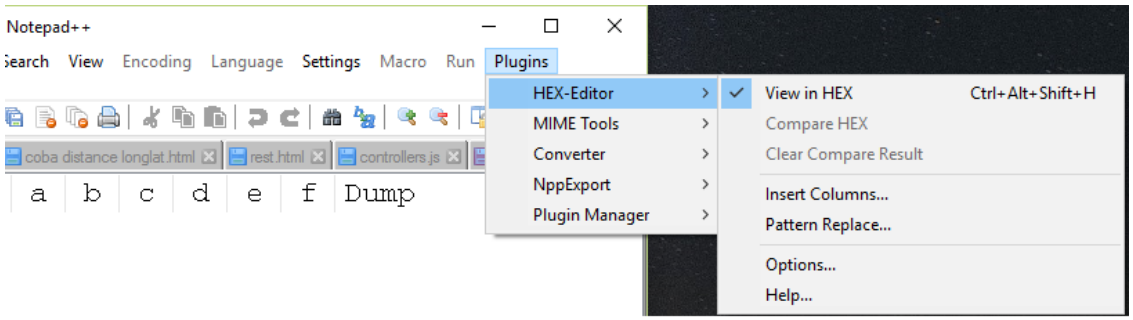
1. Buka aplikasi Notepad ++.
2. Setelah Notepad ++ terbuka, buka menu *Plugins,* klik *Plugin Manager*, kemudian klik *Show Plugin Manager.*



3. Setelah itu, centang *Plugin HEX-Editor*, dan klik *Install.*



4. Tunggu hingga proses *download* dan *install* selesai. Instalasi mungkin akan gagal bila terhubung ke internet yang menggunakan *proxy*. Setelah selesai proses instalasi, klik *yes* maka aplikasi Notepad ++ akan di *restart.*
5. Apabila telah ter-*install,* maka di dalam menu *Plugins* akan ada HEX-Editor, klik View in HEX bila ingin membaca file dalam mode Hexadecimal.



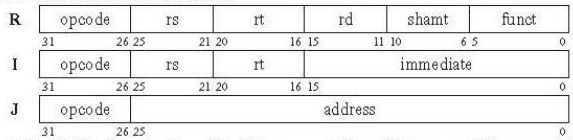
MIPS Reference Data

CORE INSTRUCTION SET

NAME, MNEMONIC		FOR-MAT	OPERATION (in Verilog)		OPCODE / FUNCT (Hex)
Add	add	R	R[rd] = R[rs] + R[rt]	(1)	0 / 20_{hex}
Add Immediate	addi	I	R[rt] = R[rs] + SignExtImm	(1,2)	8_{hex}
Add Imm. Unsigned	addiu	I	R[rt] = R[rs] + SignExtImm	(2)	9_{hex}
Add Unsigned	addu	R	R[rd] = R[rs] + R[rt]		0 / 21_{hex}
And	and	R	R[rd] = R[rs] & R[rt]		0 / 24_{hex}
And Immediate	andi	I	R[rt] = R[rs] & ZeroExtImm	(3)	c_{hex}
Branch On Equal	beq	I	if(R[rs]==R[rt]) PC=PC+4+BranchAddr	(4)	4_{hex}
Branch On Not Equal	bne	I	if(R[rs]!=R[rt]) PC=PC+4+BranchAddr	(4)	5_{hex}
Jump	j	J	PC=JumpAddr	(5)	2_{hex}
Jump And Link	jal	J	R[31]=PC+8;PC=JumpAddr	(5)	3_{hex}
Jump Register	jr	R	PC=R[rs]		0 / 08_{hex}
Load Byte Unsigned	lbu	I	R[rt]={24'b0,M[R[rs]+SignExtImm](7:0)}	(2)	24_{hex}
Load Halfword Unsigned	lhu	I	R[rt]={16'b0,M[R[rs]+SignExtImm](15:0)}	(2)	25_{hex}
Load Linked	ll	I	R[rt] = M[R[rs]+SignExtImm]	(2,7)	30_{hex}
Load Upper Imm.	lui	I	R[rt] = {imm, 16'b0}		f_{hex}
Load Word	lw	I	R[rt] = M[R[rs]+SignExtImm]	(2)	23_{hex}
Nor	nor	R	R[rd] = ~ (R[rs] | R[rt])		0 / 27_{hex}
Or	or	R	R[rd] = R[rs] | R[rt]		0 / 25_{hex}
Or Immediate	ori	I	R[rt] = R[rs] | ZeroExtImm	(3)	d_{hex}
Set Less Than	slt	R	R[rd] = (R[rs] < R[rt]) ? 1 : 0		0 / $2a_{hex}$
Set Less Than Imm.	slti	I	R[rt] = (R[rs] < SignExtImm)? 1 : 0	(2)	a_{hex}
Set Less Than Imm. Unsigned	sltiu	I	R[rt] = (R[rs] < SignExtImm) ? 1 : 0	(2,6)	b_{hex}
Set Less Than Unsig.	sltu	R	R[rd] = (R[rs] < R[rt]) ? 1 : 0	(6)	0 / $2b_{hex}$
Shift Left Logical	sll	R	R[rd] = R[rt] << shamt		0 / 00_{hex}
Shift Right Logical	srl	R	R[rd] = R[rt] >>> shamt		0 / 02_{hex}
Store Byte	sb	I	M[R[rs]+SignExtImm](7:0) = R[rt](7:0)	(2)	28_{hex}
Store Conditional	sc	I	M[R[rs]+SignExtImm] = R[rt]; R[rt] = (*atomic*) ? 1 : 0	(2,7)	38_{hex}
Store Halfword	sh	I	M[R[rs]+SignExtImm](15:0) = R[rt](15:0)	(2)	29_{hex}
Store Word	sw	I	M[R[rs]+SignExtImm] = R[rt]	(2)	$2b_{hex}$
Subtract	sub	R	R[rd] = R[rs] - R[rt]	(1)	0 / 22_{hex}
Subtract Unsigned	subu	R	R[rd] = R[rs] - R[rt]		0 / 23_{hex}

(1) May cause overflow exception
(2) SignExtImm = { 16{immediate[15]}, immediate }
(3) ZeroExtImm = { 16{1b'0}, immediate }
(4) BranchAddr = { 14{immediate[15]}, immediate, 2'b0 }
(5) JumpAddr = { PC+4[31:28], address, 2'b0 }
(6) Operands considered unsigned numbers (vs. 2's comp.)
(7) Atomic test&set pair; R[rt] = 1 if pair atomic, 0 if not atomic

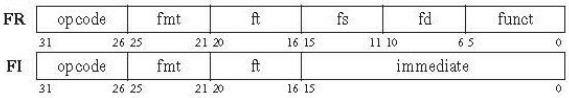
BASIC INSTRUCTION FORMATS



ARITHMETIC CORE INSTRUCTION SET

NAME, MNEMONIC		FOR-MAT	OPERATION		OPCODE / FMT /FT / FUNCT (Hex)
Branch On FP True	bc1t	FI	if(FPcond)PC=PC+4+BranchAddr	(4)	11/8/1/--
Branch On FP False	bc1f	FI	if(!FPcond)PC=PC+4+BranchAddr	(4)	11/8/0/--
Divide	div	R	Lo=R[rs]/R[rt]; Hi=R[rs]%R[rt]		0/--/--/1a
Divide Unsigned	divu	R	Lo=R[rs]/R[rt]; Hi=R[rs]%R[rt]	(6)	0/--/--/1b
FP Add Single	add.s	FR	F[fd]= F[fs] + F[ft]		11/10/--/0
FP Add Double	add.d	FR	{F[fd],F[fd+1]} = {F[fs],F[fs+1]} + {F[ft],F[ft+1]}		11/11/--/0
FP Compare Single	c.x.s*	FR	FPcond = (F[fs] *op* F[ft]) ? 1 : 0		11/10/--/*y*
FP Compare Double	c.x.d*	FR	FPcond = ({F[fs],F[fs+1]} *op* {F[ft],F[ft+1]}) ? 1 : 0		11/11/--/*y*
* (*x* is `eq`, `lt`, or `le`) (*op* is ==, <, or <=) (*y* is 32, 3c, or 3e)					
FP Divide Single	div.s	FR	F[fd] = F[fs] / F[ft]		11/10/--/3
FP Divide Double	div.d	FR	{F[fd],F[fd+1]} = {F[fs],F[fs+1]} / {F[ft],F[ft+1]}		11/11/--/3
FP Multiply Single	mul.s	FR	F[fd] = F[fs] * F[ft]		11/10/--/2
FP Multiply Double	mul.d	FR	{F[fd],F[fd+1]} = {F[fs],F[fs+1]} * {F[ft],F[ft+1]}		11/11/--/2
FP Subtract Single	sub.s	FR	F[fd]=F[fs] - F[ft]		11/10/--/1
FP Subtract Double	sub.d	FR	{F[fd],F[fd+1]} = {F[fs],F[fs+1]} - {F[ft],F[ft+1]}		11/11/--/1
Load FP Single	lwc1	I	F[rt]=M[R[rs]+SignExtImm]	(2)	31/--/--/--
Load FP Double	ldc1	I	F[rt]=M[R[rs]+SignExtImm]; F[rt+1]=M[R[rs]+SignExtImm+4]	(2)	35/--/--/--
Move From Hi	mfhi	R	R[rd] = Hi		0 /--/--/10
Move From Lo	mflo	R	R[rd] = Lo		0 /--/--/12
Move From Control	mfc0	R	R[rd] = CR[rs]		10 /0/--/0
Multiply	mult	R	{Hi,Lo} = R[rs] * R[rt]		0/--/--/18
Multiply Unsigned	multu	R	{Hi,Lo} = R[rs] * R[rt]	(6)	0/--/--/19
Shift Right Arith.	sra	R	R[rd] = R[rt] >> shamt		0/--/--/3
Store FP Single	swc1	I	M[R[rs]+SignExtImm] = F[rt]	(2)	39/--/--/--
Store FP Double	sdc1	I	M[R[rs]+SignExtImm] = F[rt]; M[R[rs]+SignExtImm+4] = F[rt+1]	(2)	3d/--/--/--

FLOATING-POINT INSTRUCTION FORMATS



PSEUDOINSTRUCTION SET

NAME	MNEMONIC	OPERATION
Branch Less Than	blt	if(R[rs]<R[rt]) PC = Label
Branch Greater Than	bgt	if(R[rs]>R[rt]) PC = Label
Branch Less Than or Equal	ble	if(R[rs]<=R[rt]) PC = Label
Branch Greater Than or Equal	bge	if(R[rs]>=R[rt]) PC = Label
Load Immediate	li	R[rd] = immediate
Move	move	R[rd] = R[rs]

REGISTER NAME, NUMBER, USE, CALL CONVENTION

NAME	NUMBER	USE	PRESERVED ACROSS A CALL?
$zero	0	The Constant Value 0	N.A.
$at	1	Assembler Temporary	No
$v0-$v1	2-3	Values for Function Results and Expression Evaluation	No
$a0-$a3	4-7	Arguments	No
$t0-$t7	8-15	Temporaries	No
$s0-$s7	16-23	Saved Temporaries	Yes
$t8-$t9	24-25	Temporaries	No
$k0-$k1	26-27	Reserved for OS Kernel	No
$gp	28	Global Pointer	Yes
$sp	29	Stack Pointer	Yes
$fp	30	Frame Pointer	Yes
$ra	31	Return Address	Yes

OPCODES, BASE CONVERSION, ASCII SYMBOLS

MIPS opcode (31:26)	(1) MIPS funct (5:0)	(2) MIPS funct (5:0)	Binary	Decimal	Hexadecimal	ASCII Character	Decimal	Hexadecimal	ASCII Character
(1)	sll	add.*f*	00 0000	0	0	NUL	64	40	@
		sub.*f*	00 0001	1	1	SOH	65	41	A
j	srl	mul.*f*	00 0010	2	2	STX	66	42	B
jal	sra	div.*f*	00 0011	3	3	ETX	67	43	C
beq	sllv	sqrt.*f*	00 0100	4	4	EOT	68	44	D
bne		abs.*f*	00 0101	5	5	ENQ	69	45	E
blez	srlv	mov.*f*	00 0110	6	6	ACK	70	46	F
bgtz	srav	neg.*f*	00 0111	7	7	BEL	71	47	G
addi	jr		00 1000	8	8	BS	72	48	H
addiu	jalr		00 1001	9	9	HT	73	49	I
slti	movz		00 1010	10	a	LF	74	4a	J
sltiu	movn		00 1011	11	b	VT	75	4b	K
andi	syscall	round.w.*f*	00 1100	12	c	FF	76	4c	L
ori	break	trunc.w.*f*	00 1101	13	d	CR	77	4d	M
xori		ceil.w.*f*	00 1110	14	e	SO	78	4e	N
lui	sync	floor.w.*f*	00 1111	15	f	SI	79	4f	O
	mfhi		01 0000	16	10	DLE	80	50	P
(2)	mthi		01 0001	17	11	DC1	81	51	Q
	mflo	movz.*f*	01 0010	18	12	DC2	82	52	R
	mtlo	movn.*f*	01 0011	19	13	DC3	83	53	S
			01 0100	20	14	DC4	84	54	T
			01 0101	21	15	NAK	85	55	U
			01 0110	22	16	SYN	86	56	V
			01 0111	23	17	ETB	87	57	W
	mult		01 1000	24	18	CAN	88	58	X
	multu		01 1001	25	19	EM	89	59	Y
	div		01 1010	26	1a	SUB	90	5a	Z
	divu		01 1011	27	1b	ESC	91	5b	[
			01 1100	28	1c	FS	92	5c	\
			01 1101	29	1d	GS	93	5d	]
			01 1110	30	1e	RS	94	5e	^
			01 1111	31	1f	US	95	5f	_
lb	add	cvt.s.*f*	10 0000	32	20	Space	96	60	`
lh	addu	cvt.d.*f*	10 0001	33	21	!	97	61	a
lwl	sub		10 0010	34	22	"	98	62	b
lw	subu		10 0011	35	23	#	99	63	c
lbu	and	cvt.w.*f*	10 0100	36	24	$	100	64	d
lhu	or		10 0101	37	25	%	101	65	e
lwr	xor		10 0110	38	26	&	102	66	f
	nor		10 0111	39	27	'	103	67	g
sb			10 1000	40	28	(	104	68	h
sh			10 1001	41	29	)	105	69	i
swl	slt		10 1010	42	2a	*	106	6a	j
sw	sltu		10 1011	43	2b	+	107	6b	k
			10 1100	44	2c	,	108	6c	l
			10 1101	45	2d	-	109	6d	m
swr			10 1110	46	2e	.	110	6e	n
cache			10 1111	47	2f	/	111	6f	o
ll	tge	c.*f*.*f*	11 0000	48	30	0	112	70	p
lwc1	tgeu	c.un.*f*	11 0001	49	31	1	113	71	q
lwc2	tlt	c.eq.*f*	11 0010	50	32	2	114	72	r
pref	tltu	c.ueq.*f*	11 0011	51	33	3	115	73	s
	teq	c.olt.*f*	11 0100	52	34	4	116	74	t
ldc1		c.ult.*f*	11 0101	53	35	5	117	75	u
ldc2	tne	c.ole.*f*	11 0110	54	36	6	118	76	v
		c.ule.*f*	11 0111	55	37	7	119	77	w
sc		c.sf.*f*	11 1000	56	38	8	120	78	x
swc1		c.ngle.*f*	11 1001	57	39	9	121	79	y
swc2		c.seq.*f*	11 1010	58	3a	:	122	7a	z
		c.ngl.*f*	11 1011	59	3b	;	123	7b	{
		c.lt.*f*	11 1100	60	3c	<	124	7c	|
sdc1		c.nge.*f*	11 1101	61	3d	=	125	7d	}
sdc2		c.le.*f*	11 1110	62	3e	>	126	7e	~
		c.ngt.*f*	11 1111	63	3f	?	127	7f	DEL

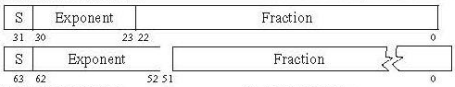
(1) opcode(31:26) = 0
(2) opcode(31:26) = 17_{ten} (11_{hex}); if fmt(25:21)=16_{ten} (10_{hex}) *f* = s (single); if fmt(25:21)=17_{ten} (11_{hex}) *f* = d (double)

IEEE 754 FLOATING-POINT STANDARD

$$(-1)^S \times (1 + \text{Fraction}) \times 2^{(\text{Exponent} - \text{Bias})}$$

where Single Precision Bias = 127, Double Precision Bias = 1023.

IEEE Single Precision and Double Precision Formats:

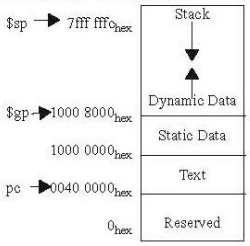


IEEE 754 Symbols

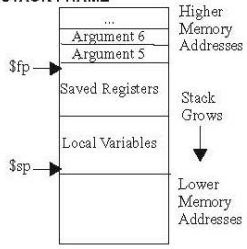
Exponent	Fraction	Object
0	0	± 0
0	≠0	± Denorm
1 to MAX - 1	anything	± Fl. Pt. Num.
MAX	0	±∞
MAX	≠0	NaN

S.P. MAX = 255, D.P. MAX = 2047

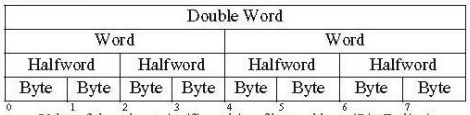
MEMORY ALLOCATION



STACK FRAME

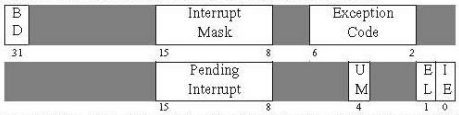


DATA ALIGNMENT



Value of three least significant bits of byte address (Big Endian)

EXCEPTION CONTROL REGISTERS: CAUSE AND STATUS



BD = Branch Delay, UM = User Mode, EL = Exception Level, IE =Interrupt Enable

EXCEPTION CODES

Number	Name	Cause of Exception	Number	Name	Cause of Exception
0	Int	Interrupt (hardware)	9	Bp	Breakpoint Exception
4	AdEL	Address Error Exception (load or instruction fetch)	10	RI	Reserved Instruction Exception
5	AdES	Address Error Exception (store)	11	CpU	Coprocessor Unimplemented
6	IBE	Bus Error on Instruction Fetch	12	Ov	Arithmetic Overflow Exception
7	DBE	Bus Error on Load or Store	13	Tr	Trap
8	Sys	Syscall Exception	15	FPE	Floating Point Exception

SIZE PREFIXES (10^x for Disk, Communication; 2^x for Memory)

SIZE	PREFIX	SIZE	PREFIX	SIZE	PREFIX	SIZE	PREFIX
10^3, 2^{10}	Kilo-	10^{15}, 2^{50}	Peta-	10^{-3}	milli-	10^{-15}	femto-
10^6, 2^{20}	Mega-	10^{18}, 2^{60}	Exa-	10^{-6}	micro-	10^{-18}	atto-
10^9, 2^{30}	Giga-	10^{21}, 2^{70}	Zetta-	10^{-9}	nano-	10^{-21}	zepto-
10^{12}, 2^{40}	Tera-	10^{24}, 2^{80}	Yotta-	10^{-12}	pico-	10^{-24}	yocto-

The symbol for each prefix is just its first letter, except μ is used for micro.

LAMPIRAN IV : ALTERA® MEGAFUNCTION ALTSYNCRAM

Altera® MegaFunction ALTSYNCRAM merupakan MegaFunction untuk mengimplementasikan RAM *dual-port* berdasarkan parameter-parameter tertentu. Altera® MegaFunction ALTSYNCRAM tersedia untuk perangkat FPGA Cyclone™, Stratix™, dan Stratix™ GX. Altera® merekomendasikan untuk menggunakan MegaWizard® dalam melakukan instansiasi ALTSYNCRAM. Deklarasi komponen dalam VHDL untuk ALTSYNCRAM diberikan sebagai berikut.

```
COMPONENT altsyncram
  GENERIC
    (
    OPERATION_MODE                         :  STRING  := "SINGLE_PORT";
    WIDTH_A                                :  INTEGER := 8;     -- 1;
    WIDTHAD_A                              :  INTEGER := 2;     -- 1;
    NUMWORDS_A                             :  INTEGER := 4;     -- 1;
    OUTDATA_REG_A                          :  STRING  := "UNREGISTERED";
    ADDRESS_ACLR_A                         :  STRING  := "NONE";
    OUTDATA_ACLR_A                         :  STRING  := "NONE";
    INDATA_ACLR_A                          :  STRING  := "NONE";
    WRCONTROL_ACLR_A                       :  STRING  := "NONE";
    BYTEENA_ACLR_A                         :  STRING  := "NONE";
    WIDTH_BYTEENA_A                        :  INTEGER := 1;
    WIDTH_B                                :  INTEGER := 8;     -- 1;
    WIDTHAD_B                              :  INTEGER := 4;     -- 1;
    NUMWORDS_B                             :  INTEGER := 4;     -- 1;
    RDCONTROL_REG_B                        :  STRING  := "CLOCK1";
    ADDRESS_REG_B                          :  STRING  := "CLOCK1";
    INDATA_REG_B                           :  STRING  := "CLOCK1";
    WRCONTROL_WRADDRESS_REG_B              :  STRING  := "CLOCK1";
    BYTEENA_REG_B                          :  STRING  := "CLOCK1";
    OUTDATA_REG_B                          :  STRING  := "UNREGISTERED";
    OUTDATA_ACLR_B                         :  STRING  := "NONE";
    RDCONTROL_ACLR_B                       :  STRING  := "NONE";
    INDATA_ACLR_B                          :  STRING  := "NONE";
    WRCONTROL_ACLR_B                       :  STRING  := "NONE";
    ADDRESS_ACLR_B                         :  STRING  := "NONE";
    BYTEENA_ACLR_B                         :  STRING  := "NONE";
    WIDTH_BYTEENA_B                        :  INTEGER := 1;
    BYTE_SIZE                              :  INTEGER := 8;
    READ_DURING_WRITE_MODE_MIXED_PORTS     :  STRING  := "DONT_CARE";
    RAM_BLOCK_TYPE                         :  STRING  := "AUTO";
    INIT_FILE                              :  STRING  := "UNUSED";
    INIT_FILE_LAYOUT                       :  STRING  := "PORT_A";
    MAXIMUM_DEPTH                          :  INTEGER := 0;
    INTENDED_DEVICE_FAMILY                 :  STRING  := "STRATIX";
    LPM_HINT                               :  STRING  := "BOGUS"
  );
  PORT
    (
    wren_a, wren_b, aclr0, aclr1               : IN STD_LOGIC := '0';
    rden_b, clock0, clock1, clocken0, locken1  : IN STD_LOGIC := '1';
    data_a     : IN STD_LOGIC_VECTOR(WIDTH_A-1 DOWNTO 0):= (OTHERS => '0');
    data_b     : IN STD_LOGIC_VECTOR(WIDTH_B-1 DOWNTO 0):= (OTHERS => '0');
    address_a  : IN STD_LOGIC_VECTOR(WIDTHAD_A-1 DOWNTO 0):=(OTHERS => '0');
    address_b  : IN STD_LOGIC_VECTOR(WIDTHAD_B-1 DOWNTO 0):=(OTHERS => '0');
    byteena_a  : IN STD_LOGIC_VECTOR((WIDTH_BYTEENA_A-1) DOWNTO 0):=(OTHERS => '1');
    byteena_b  : IN STD_LOGIC_VECTOR((WIDTH_BYTEENA_B-1) DOWNTO 0):=(OTHERS => '1');
    q_a        : OUT STD_LOGIC_VECTOR(WIDTH_A - 1 DOWNTO 0);
    q_b        : OUT STD_LOGIC_VECTOR(WIDTH_B - 1 DOWNTO 0)
    );
END COMPONENT;
```

Untuk menggunakan komponen ini, kita juga dapat memanggil *library* ALTSYNCRAM sebagai berikut.

```
LIBRARY altera_mf;
USE altera_mf.altera_mf_components.all;
```

Definisi *port* untuk ALTSYNCRAM diberikan sebagai berikut.

Port *Input*

Port Name	Required	Description	Comments
wren_a	No	Write enable input.	The wren_a port is not available when the OPERATION_MODE parameter is set to "ROM" mode.
wren_b	No	Write enable input.	The wren_b input port is available only when the OPERATION_MODE parameter is set to"BIDIR_DUAL_PORT".
rden_b	No	Read enable input port.	The rden_b input port is available only when the OPERATION_MODE parameter is set to "DUAL_PORT"and when the RAM_BLOCK_TYPE parameter is not set to "MEGARAM".
data_a[]	No	Data input port to the memory.	Input port WIDTH_A-1..0 wide.
data_b[]	No	Data input port to the memory.	Input port WIDTH_B-1..0 wide.
address_a[]	Yes	Address input to the memory.	Input port WIDTHAD_A-1..0 wide.
address_b[]	Yes	Address input to the memory.	Input port WIDTHAD_B-1..0 wide.
clock0	Yes	Clock input port for the RAM.	
clock1	No	Clock input port for the RAM.	
clocken0	No	Clock enable for clock0.	
clocken1	No	Clock enable for clock1.	
aclr0	No	The first asynchronous clear input.	
aclr1	No	The second asynchronous clear input.	
byteena_a[]	No	Byte enable input port.	Input port WIDTH_BYTEENA_A-1..0 wide. The byteena_a enable input port can be used only when the data_a port is at least two bytes wide.
byteena_b[]	No	Byte enable input port.	Input port WIDTH_BYTEENA_B-1..0 wide. Thebyteena_b enable input port can be used only when the data_b port is at least two bytes wide.

Port *Output*

Port Name	Required	Description	Comments
q_a[]	Yes	Data output port from the memory.	Output port WIDTH_A-1..0 wide. The q_a[] port is legal only when the OPERATION_MODE parameter is set to "SINGLE_PORT", "BIDIR_DUAL_PORT", or"ROM".
q_b[]	Yes	Data output port from the memory.	Output port WIDTH_B-1..0 wide. The q_b[] port is legal only when the OPERATION_MODE parameter is set to "DUAL_PORT" or "BIDIR_DUAL_PORT".

Deskripsi Parameter

Parameter	Type	Required	Comments
OPERATION_MODE	String	Yes	Specifies the operation of the RAM. Values are "SINGLE_PORT", "DUAL_PORT","BIDIR_DUAL_PORT", or "ROM". If omitted, the default is "BIDIR_DUAL_PORT".
WIDTH_A	Integer	Yes	Specifies the width of the data_a[] input port. If omitted, the default is "1".
WIDTHAD_A	Integer	Yes	Specifies the width of the address_a[] input port. When the OPERATION_MODEparameter is set to "BIDIR_DUAL" mode, the WIDTH_A parameter is required. If omitted, the default is "1".
NUMWORDS_A	Integer	No	Number of words stored in memory. If omitted, the default is 2 ^ WIDTHAD_A.
OUTDATA_REG_A	String	No	Specifies the clock for the q_a[] port. Values are "CLOCK0", "CLOCK1","UNREGISTERED", or "UNUSED". If omitted, the default is "UNREGISTERED".
ADDRESS_ACLR_A	String	No	Specifies the asynchronous clear for the address_a[] port. Values are"CLEAR0", "NONE", or "UNUSED". If omitted, the default is "NONE".
OUTDATA_ACLR_A	String	No	Specifies the asynchronous clear for the q_a[] output port. Values are "CLEAR0", "CLEAR1", "NONE", or "UNUSED". If omitted, the default is "NONE".
INDATA_ACLR_A	String	No	Specifies the asynchronous clear for the data_a[]input port. Values are"CLEAR0", "NONE", or "UNUSED". If omitted, the default is "NONE".
WRCONTROL_ACLR_A	String	No	Specifies the asynchronous clear for the wren_ainput port. Values are "CLEAR0","NONE", or "UNUSED". If omitted, the default is "NONE".
BYTEENA_ACLR_A	String	No	Specifies asynchronous clear for the byteena_a input port. Values are"CLEAR0", "CLEAR1", or "NONE". If omitted, the default is "NONE".
WIDTH_BYTEENA_A	Integer	No	Specifies the width of the byteena_a input port. The WIDTH_BYTEENA_Aparameter

			value must be equal to WIDTH_A / BYTE_SIZE. The WIDTH_BYTEENA_Aparameter is required if the byteena_a port is specified.
WIDTH_B	Integer	No	Specifies the width of the data_b[] input port. When the OPERATION_MODEparameter is set to "DUAL_PORT" mode, the WIDTH_B parameter is required. If omitted, the default is "1".
WIDTHAD_B	Integer	No	Specifies the width of the address_b[] input port. If omitted, the default is "1".
NUMWORDS_B	Integer	No	Number of words stored in memory. If omitted, the default is 2 ^ WIDTHAD_B.
RDCONTROL_REG_B	String	No	Specifies the clock for the rden_b port during read mode. Values are "CLOCK0","CLOCK1", or "UNUSED". If omitted, the default is "CLOCK1".
ADDRESS_REG_B	String	No	Specifies the clock for the address_b[] port. Values are "CLOCK0", "CLOCK1", or "UNUSED". If omitted, the default is "CLOCK1".
INDATA_REG_B	String	No	Specifies the clock for the data_b[] port. Values are "CLOCK0", "CLOCK1", or"UNUSED". If omitted, the default is "CLOCK1".
WRCONTROL_WRADDRESS REG_B	String	No	Specifies the clock for the wren_b and address_b[] port during write mode. Values are "CLOCK0", "CLOCK1", or "UNUSED". If omitted, the default is"CLOCK1".
BYTEENA_REG_B	String	No	Specifies the clock for the byteena_b[] port. Values are "CLOCK0", "CLOCK1", or "UNUSED". If omitted, the default is "CLOCK1".
OUTDATA_REG_B	String	No	Specifies the clock for the q_b[] port. Values are "CLOCK0", "CLOCK1","UNREGISTERED", or "UNUSED". If omitted, the default is "UNREGISTERED".
OUTDATA_ACLR_B	String	No	Specifies the asynchronous clear for the q_b[] output port. Values are"CLEAR0", "CLEAR1", "NONE", or "UNUSED". If omitted, the default is "NONE".
RDCONTROL_ACLR_B	String	No	Specifies the clear source for the port B read enable control register. Values are"CLEAR0", "CLEAR1", "NONE", or "UNUSED". The default value is "NONE".
INDATA_ACLR_B	String	No	Specifies the asynchronous clear for the data_b[]input port. Values are"CLEAR0", "NONE", or "UNUSED". If omitted, the default is "NONE".
WRCONTROL_ACLR_B	String	No	Specifies the asynchronous clear for the wren_binput port. Values are "CLEAR0","NONE", or "UNUSED". If omitted, the default is "NONE".
ADDRESS_ACLR_B	String	No	Specifies the asynchronous clear for the address_b[] port. Values are"CLEAR0", "NONE", or "UNUSED". If omitted, the default is "NONE".
BYTEENA_ACLR_B	String	No	Specifies asynchronous clear for the byteena_b input port. Values are"CLEAR0", "CLEAR1", "NONE", or "UNUSED". If omitted, the default is "NONE".
WIDTH_BYTEENA_B	Integer	No	Specifies the width of the byteena_b input port. The WIDTH_BYTEENA_Bparameter value must be equal to WIDTH_B / BYTE_SIZE. The WIDTH_BYTEENA_Bparameter is required if the byteena_b port is specified.
BYTE_SIZE	Integer	No	Specifies the byte enable size. Values are "8" or "9". If omitted, the default is"8".
READ_DURING_WRITE_M ODE_MIXED_PORTS	String	No	Specifies the behavior when the read and write operations occur at different ports on the same RAM address. Values are "OLD_DATA" or "DONT_CARE". The default value is "DONT_CARE". When the OPERATION_MODE parameter is set to"MEGARAM", the READ_DURING_WRITE_MODE_MIXED_PORTS parameter must be set to "DONT_CARE".
RAM_BLOCK_TYPE	String	No	Specifies the RAM block type. Values are "M512", "M4K", "MEGARAM", or "AUTO". If omitted, the default is "AUTO".
INIT_FILE	String	No	Name of the Memory Initialization File (**.mif**) or Hexadecimal (Intel-Format) Output File (**.hexout**) containing RAM initialization data ("<*file name*>"), or"UNUSED". The default is "UNUSED". The INIT_FILE parameter is unavailable when the RAM_BLOCK_TYPE parameter is set to MEGARAM. When the OPERATION_MODE parameter is set to "DUAL_PORT", the Compiler uses only theWIDTH_B parameters to read the initialization file.
INIT_FILE_LAYOUT	String	No	Specifies the layout port used with the initialization file. Values are "PORT_A" or"PORT_B". If the OPERATION_MODE is set to "DUAL_PORT" mode, the default value is "PORT_B". If the OPERATION_MODE is set to other modes, the the default value is "PORT_A".
MAXIMUM_DEPTH	Integer	No	Specifies the maximum segmented value of the RAM. The MAXIMUM_DEPTHparameter value depends on the RAM_BLOCK_TYPE parameter. If omitted, the default is "0".
INTENDED_DEVICE_FAM ILY	String	No	This parameter is used for modeling and behavioral simulation purposes. Create the PLL with the **MegaWizard Plug-in Manager** to calculate the value for this parameter.
LPM_HINT	String	No	Allows you to assign Altera-specific parameters in VHDL Design Files (**.vhd**). If omitted, the default is "UNUSED".

Sumber :
http://www.pldworld.com/_altera/html/_sw/q2help/source/mega/mega_file_altsynch_ram.htm
diakses 24 September 2014.