BAB 2

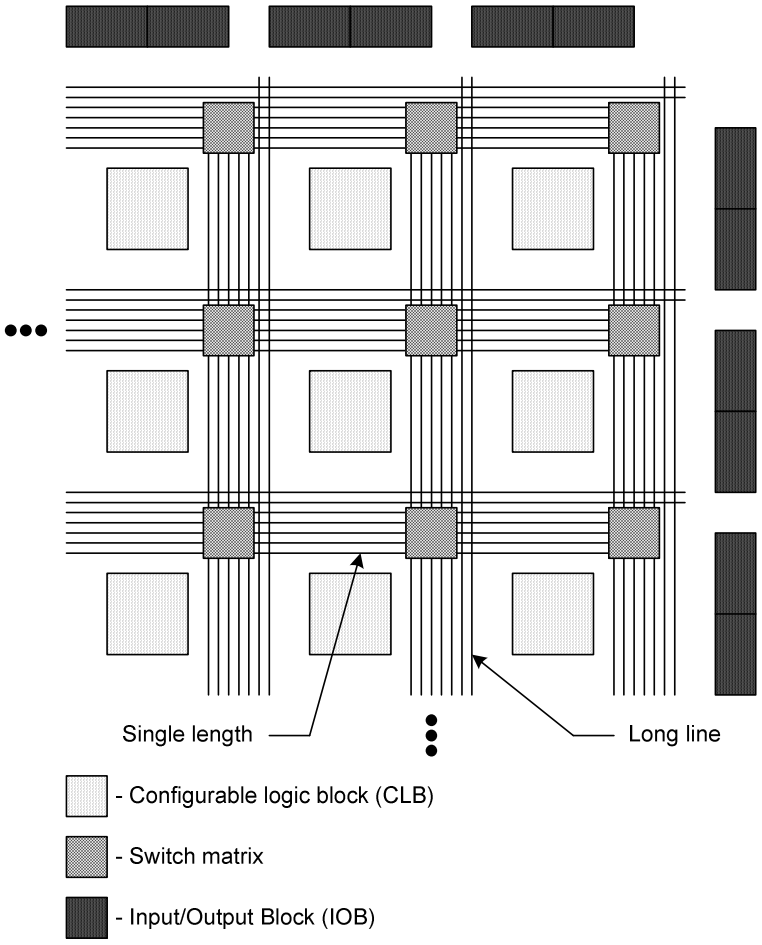
LANDASAN TEORI

2.1 Teori-teori Dasar/Umum

Teori dasar/umum yang akan dibahas di sini adalah pertama mengenai karakteristik FPGA (*Field-Programmable Gate Array*) yang merupakan salah satu jenis PLD (*Programmable Logic Device*) yang nantinya akan digunakan pada saat implementasi. Kedua akan membahas mengenai bahasa pemrograman VHDL (VHSIC (*Very High Speed Integrated Circuit*) *Hardware Decription Language*) yang digunakan untuk melakukan perancangan. Ketiga mengenai *software* Xilinx ISE WebPACK yang merupakan *integrated design environment* untuk melakukan sintesis, implementasi, hingga menghasilkan file bit yang nantinya digunakan untuk mengkonfigurasi FPGA. *Software* Xilinx ISE WebPACK dapat didownload secara gratis dari internet pada situs resmi Xilinx (www.xilinx.com). Modul FPGA dan *software* ISE WebPACK yang digunakan diproduksi oleh perusahaan Xilinx yang merupakan perusahan besar pertama yang memproduksi FPGA. (http://www.fpga-faq.com/FAQ_Pages/0007_Device_type_comparisons.htm, 2003)

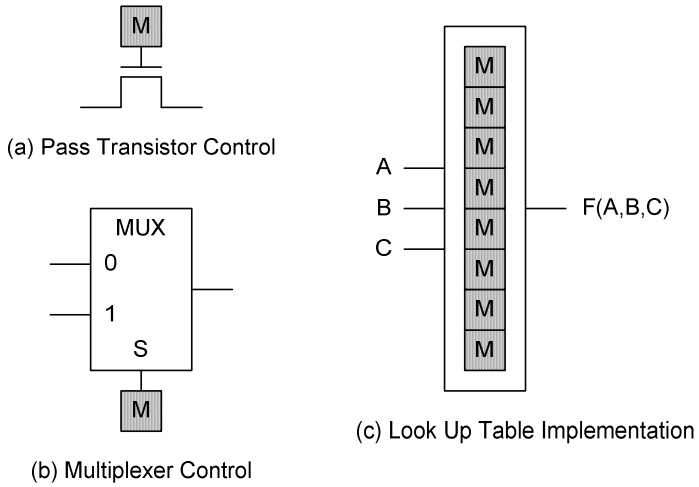
2.1.1 FPGA (*Field-Programmable Gate Array*)

FPGA terdiri dari tiga komponen utama (Gambar 2.1), yaitu *Configurable Logic Block* (CLB), *Switch Matrix*, dan *Input*/*Output Block* (IOB). (Mano dan Kime, 2001, pp328-329)



Gambar 2.1 Tiga Komponen Utama FPGA

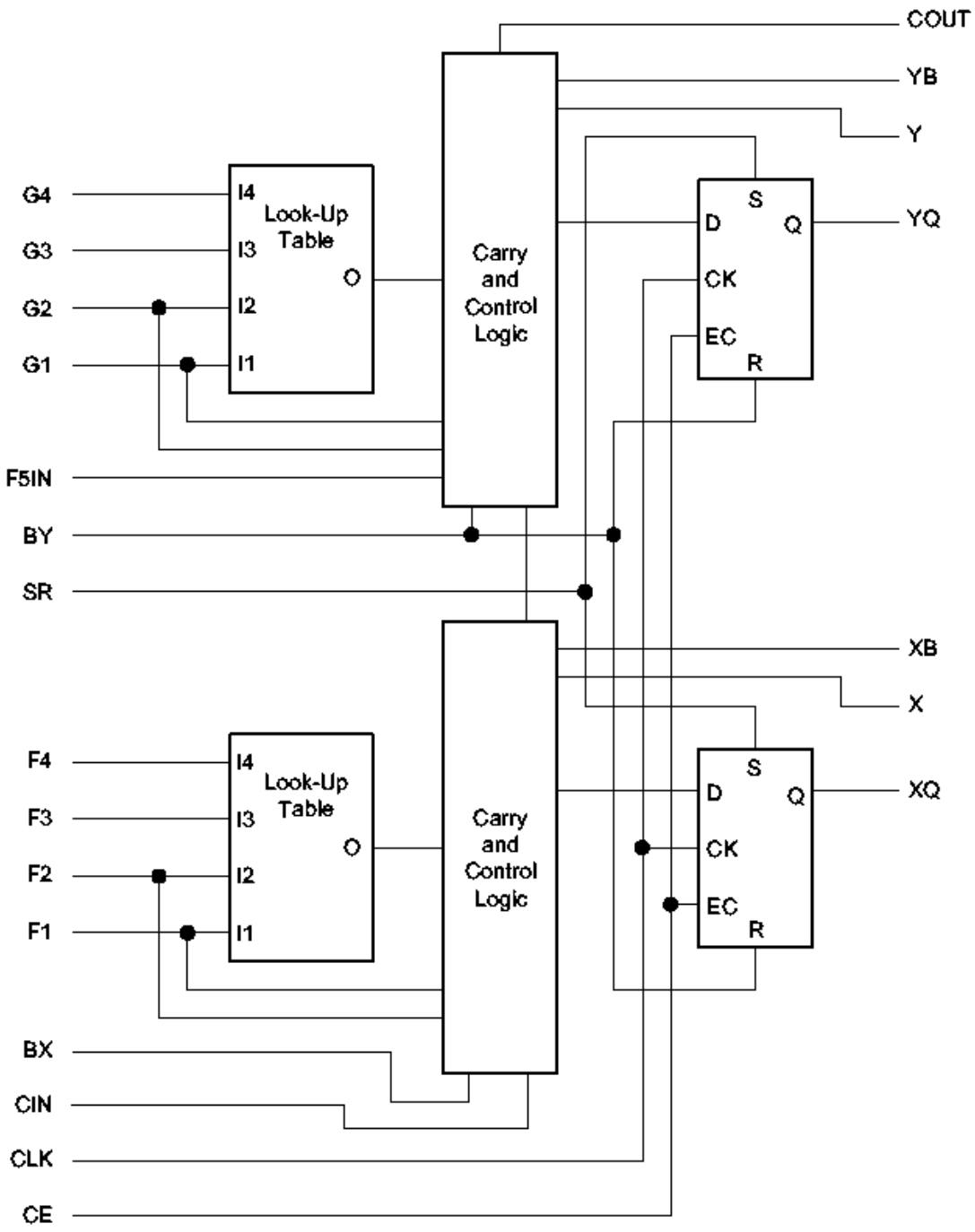
Informasi program untuk mengontrol elemen logika yang dapat dikonfigurasi dan interkoneksi antara sumber-sumber, disimpan menggunakan teknologi SRAM (*Static Random Access Memory*). Implementasi logika dari pengontrolan oleh bit-bit SRAM menggunakan tiga teknik, yaitu *Pass Transistor Control* (Gambar 2.2a), *Multiplexer Control* (Gambar 2.2b), dan *Look Up Table* (LUT) (Gambar 2.2c). (Mano dan Kime, 2001, pp329)



Gambar 2.2 Tiga Teknik Implementasi Pengontrolan Logika pada FPGA

CLB merupakan *array* dari blok-blok atau elemen untuk mengkonfigurasi atau membangun logika. Blok-blok dasar yang membangun sebuah CLB disebut *logic cell* (LC). Setiap LC mengandung *4-input function generator, carry logic*, dan *storage element*. Setiap CLB terdiri dari empat buah LC yang dikelompokan ke dalam dua *slice* yang serupa (Gambar 2.3). (Xilinx Inc., 2003, pp3)

4-*input function generator* diimplementasikan sebagai 4-*input look-up table* (LUT). Setiap LUT dapat menyediakan 16x1-*bit synchronous RAM*. Sehingga dua LUT dalam sebuah *slice* dapat disatukan untuk membentuk 16x2-*bit* atau 32x1-*bit synchronous RAM* atau 16x1-*bit dual-port synchronous RAM*. LUT juga dapat menyediakan 16-*bit shift register* yang cocok untuk menangkap data kecepatan tinggi atau *burst-mode*. (Xilinx Inc., 2003, p3)



Gambar 2.3 Skematik Sebuah *Slice*

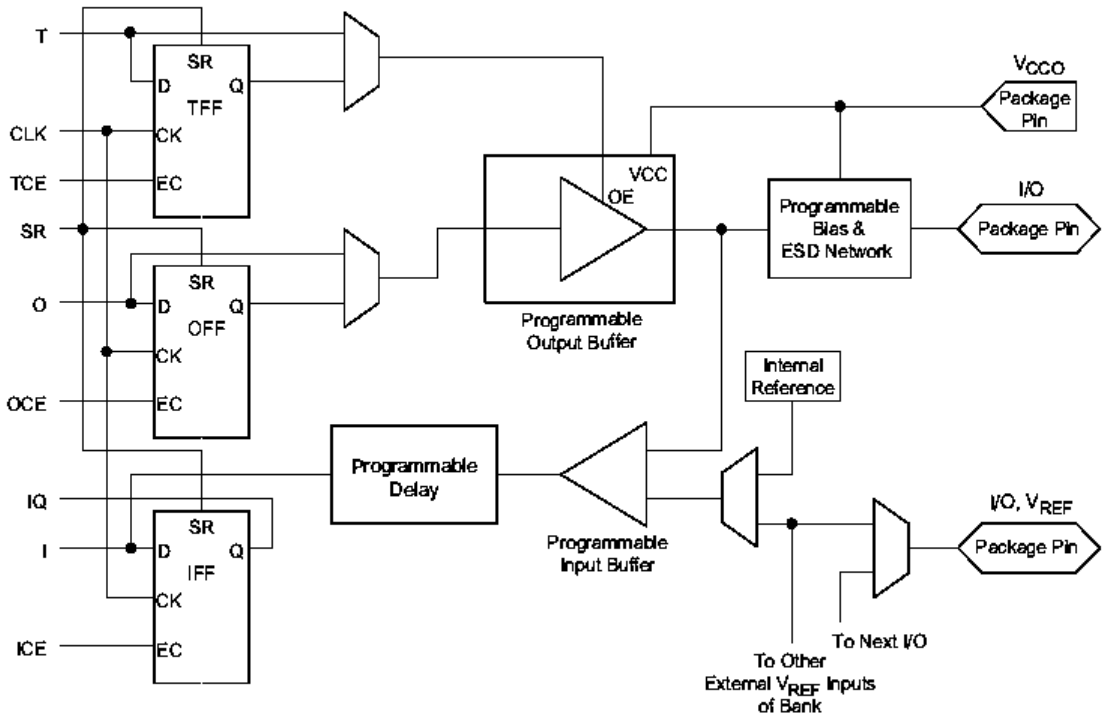
Storage element pada *slice* dapat dikonfigurasi sebagai *edge-triggered D-type flip-flop* atau sebagai *level-sensitive latch*. Masukkan pada D dapat melalui *function generator* di dalam *slice* maupun langsung dari masukan *slice* (tidak melalui *function generator*). (Xilinx Inc., 2003, pp3)

Switch Matrix menghubungkan ke dan dari CLB dan IOB yang dapat diprogram, yaitu dengan *Pass Transistor Control*. *Long-line* merupakan jenis persambungan jarak jauh, sedangkan *Single-length* menghubungkan antara CLB atau IOB yang bersebelahan. (Mano dan Kime, 2001, pp330-331)

IOB menyediakan hubungan antara *pin* dari paket (I/O *pin*) dengan logika internal. Keistimewaannya, *input* dan *output* dapat mendukung banyak standar sinyalisasi. Standar sinyal untuk *low-voltage* bergantung pada V_{REF} sedangkan untuk *high-voltage* bergantung pada V_{CCO}. (Xilinx Inc., 2003, p1-3)

Rangkaian *weak-keeper* terhubung dengan setiap *output*. Tegangan pada V_{REF} harus menyediakan standar sinyal yang diinginkan agar rangkaian ini dapat memeriksa tegangan pada *pad* dan mengedalikan *pin High* atau *Low* sesuai dengan sinyal *input* yang didapat melalui IOB *input buffer*. (Xilinx Inc., 2003, p2)

IOB terdiri dari *register* dan *three-state buffer*. *Register* pada IOB berfungsi sebagai *edge-triggered D-type flip-flop* atau sebagai *level-sensitive latch* (Gambar 2.4). *Three-state buffer* memungkinkan I/O *pin* digunakan sebagai masukan, keluaran, atau masukan/keluaran. (Xilinx Inc., 2003, p2)



Gambar 2.4 Skematik IOB

Rangkaian *Delay-locked Loop* (DLL) memungkinkan *zero propagation delay*, *low clock skew* dari sinyal *clock* keluaran yang disebarkan ke setiap *device*, dan *clock domain control* yang mutakhir. (Xilinx Inc., 2003, p20)

2.1.2 VHDL (VHSIC (*Very High Speed Integrated Circuit*) *Hardware Decription Language*)

VHDL adalah singkatan dari VHSIC *Hardware Description Language*, yang mana VHSIC adalah singkatan dari *Very High Speed Integrated Circuit*. VHDL merupakan standar yang dikembangkan oleh IEEE (*Institute of Electrical and Electronics Engineers*). Standar yang digunakan

secara luas adalah VHDL 1076-1987. Sedangkan versi revisinya, VHDL 1076-1993 masih dalam proses untuk menggantikan versi yang lama.

VHDL dapat digunakan sebagai dokumentasi, pembuktian, dan sintesa pada perancangan digital berukuran besar. VHDL menggunakan tiga pendekatan untuk mendiskripsikan *hardware*. Ketiga pendekatan itu adalah metode *structural*, *data flow*, dan *behavioral*. (http://www.gmvhdl.com/VHDL.html, 2003)

Metode *structural* membagi rancangan ke dalam beberapa blok agar mudah dimengerti dan diatur. Blok-blok tersebut kemudian dihubungkan hingga membentuk rancangan yang utuh. Setiap blok pada VHDL dapat disamakan dengan sebuah bagian yang berdiri sendiri yang disebut *entity*. *Entity* juga menggambarkan antarmuka rancangan. *Component* menggambarkan antarmuka dari *entity* yang nantinya akan digunakan sebagai sebuah *instance* (sub blok). *Component instance* adalah salinan lain dari sebuah *component* yang akan dihubungkan ke bagian (*part*) dan sinyal lain.

Pada metode *data flow*, jalur digambarkan dengan menyatakan bagaimana *input* dan *output* dalam komponen primitip (seperti gerbang AND) terhubung. Bagian arsitektur mengambarkan operasi internal dari sebuah rancangan dan metode ini menentukan bagaimana aliran data dari *input* hingga *output*.

Pendekatan dengan metode *behavioral* berbeda dengan dua metode sebelumnya, ia tidak benar-benar menggambarkan bagaimana rancangan diimplementasikan. Dasarnya adalah pendekatan kotak hitam (*black box*) dalam melakukan pemodelan, tidak perduli apa isi kotak hitam tersebut dan

bagaimana cara kerjanya. Penjabaran behavioral didukung oleh *process statement* yang muncul dalam badan *architecture declaration* seperti pada saat menyatakan *signal assignment*. Isi dari *process statement* dapat diurutkan penulisannya seperti pada *sequential statement* yang ditemukan dalam bahasa pemrograman.

Gambar pada skematik dapat langsung menyampaikan struktur rancangan, tetapi karena formatnya yang spesifik menyebabkan skematik tidak *portable*. VHDL lebih *portable* dan mudah dimodifikasi. Banyak *development software* yang hanya mendukung representasi textual dari sebuah rancangan (seperti VHDL, Verilog, Abel, dan HDL yang lain). Ada beberapa *tool* yang memungkinkan perubahan format dari representasi textual yang satu ke yang lainnya.

2.1.3 ISE (*Integrated Software Environment*)

ISE merupakan *software* keluaran perusahaan Xilinx yang digunakan untuk melakukan perancangan, sintesis, implementasi, dan menghasilkan *file* program untuk mengkonfigurasi FPGA. Dimana versi yang digunakan adalah Xilinx ISE WebPack seri 5.2i yang bisa di *download* secara gratis melalui situs resmi Xilinx.

Beberapa dukungan yang disediakan oleh Xilinx WebPACK *ISE* diantaranya *design entry* (*HDL Editor*, *StateCAD State Machine Editor*, *Schematic Editor – Engineering Capture System* (ECS)), Sintesis (XST - *Xilinx Synthesis Technology*), Simulasi (*HDL Bencher Testbench Generator*, integrasi dengan *ModelSim Simulator* dari *Model Technology, Inc.*),

Implementasi (*Translate*, *MAP*, *Place and Route* (PAR), *Floorplanner*, *Timing Analyzer*, *Xpower*), *Device Download* dan *Program File Formatting* (*BitGen* dan *iMPACT*). (Xilinx Inc., 2003a)

2.2 Prosesor RISC

Di sini akan dijabarkan karakteristik sistem RISC sebagai patokan dasar dalam perancangan, ciri-ciri sistem RISC, beberapa pengertian dan langkah-langkah yang nantinya akan digunakan dalam perancangan.

2.2.1 Elemen Sistem RISC

Karakteristik eksekusi dari rentetan instruksi mesin yang dihasilkan oleh program HLL menunjukkan aspek-aspek yang menarik, yaitu frekuensi distribusi eksekusi instruksi mesin untuk perpindahan data (*move*) 33%, percabangan bersyarat (*conditional branch*) 20%, aritmatika/logika 16%, dan yang lain berkisar antara 0,1% dan 10%. Untuk mode pengalamatan menunjukkan bahwa mayoritas instruksi menggunakan mode pengalamatan yang sederhana dimana alamat dapat dikalkulasi dalam satu siklus (*register*, *register indirect*, *displacement*), mode pengalamatan yang rumit (*memory indirect*, *indexed+indirect*, *displacement+indirect*, *stack*) hanya digunakan mendekati 18% dari instruksi. 74 hingga 80% dari operand adalah skalar (*integer*, *real*, *character*, dll) yang dapat disimpan di *register*, sisanya (20 sampai 26%) adalah *array/structure*. 90% dari *array/structure* adalah variabel global dan 80% dari skalar adalah variabel lokal. Hanya 1,25% dari pemanggilan prosesur memiliki lebih dari enam parameter. Hanya 6,7% dari

pemanggilan prosedur memiliki lebih dari enam lokal variabel. Rangkaian persarangan pemanggilan prosedur umumnya pendek dan sangat jarang lebih panjang dari enam. (http://www.ida.liu.se/~TDTS51/lectures/lectures5-6.pdf, 2003)

Rata-rata pengambilan percabangan untuk prosesor MIPS adalah 47% percabangan tidak diambil dan 53% percabangan diambil. (http://www.cs.ualberta.ca/~amaral/courses/429/webslides/Topic3-Pipelining/sld042 .htm, 2003)

Hasil penelitian atas karakteristik eksekusi dari rentetan instruksi mesin di atas menjadi titik permulaan dari arsitektur RISC. Ada tiga buah elemen yang menentukan karakteristik RISC diantaranya *register* dalam jumlah yang besar, penggunaan *pipeline*, dan kumpulan instruksi yang sederhana. (http://cse.stanford.edu/class/sophomore-college/projects-00/risc/whatis/index.html, 2003)

2.2.1.1 *Register* dalam Jumlah yang Besar

Konsep desain RISC umumnya memasukkan jumlah *register* yang banyak untuk memperkecil jumlah interaksi dengan memori. Kenyataan bahwa penyimpanan *register* merupakan perangkat penyimpan yang cepat, lebih cepat dibandingkan dengan memori utama dan *cache memory*. Kenyataan lain adalah bahwa sebagaian besar akses adalah menuju ke skalar-skalar lokal. Karena itu dimungkinkan dilakukannya dua pendekatan dasar yaitu pendekatan perangkat lunak (menggunakan kompiler untuk memaksimalkan pemakaian *register*) dan

pendekatan perangkat keras (memperbanyak jumlah *register*), sehingga akan banyak variabel yang dapat ditampung di dalam *register* untuk periode waktu yang lebih lama. (Stallings, 1997, pp126-127)

2.2.1.2 Perancangan *Pipeline* Instruksi

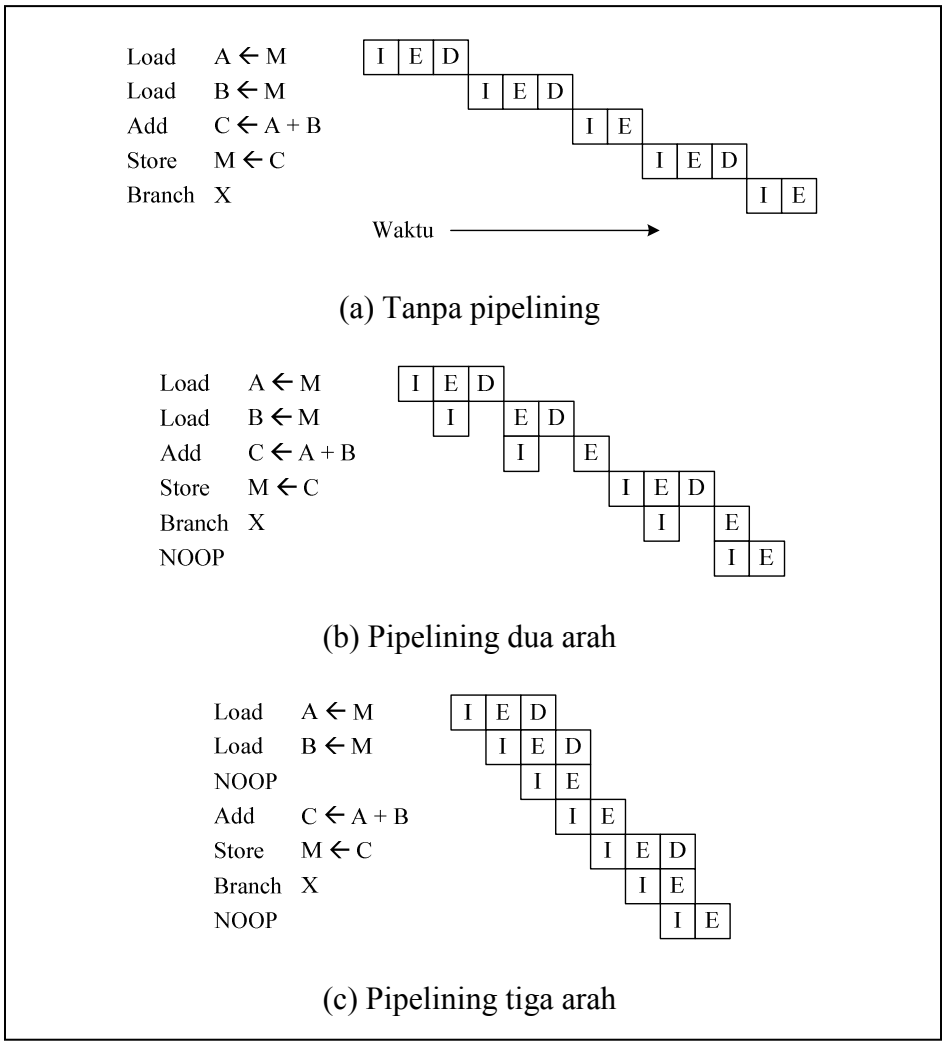
Instruksi *pipeline* merupakan teknik yang memungkinkan eksekusi simultan dari suatu bagian atau tingkatan dari instruksi untuk menambah ekfektifitas proses instruksi.

Sebagian besar instruksi merupakan operasi dari *register to register* dan sebuah siklus instruksi memiliki dua buah fase yaitu pengambilan instruksi (*Instruction Fetch* atau I) dan melakukan operasi ALU dengan input *register* dan output *register* (*Execute* atau E). Bagi operasi-operasi *load* dan *store*, diperlukan tiga buah fase, yaitu pengambilan instruksi (I), menghitung alamat memori (*Eksekusi* atau E) dan operasi *register* ke memori atau memori ke *register* (*Memory* atau D). (Stallings, 1997, p141)

Gambar 2.5a menunjukkan eksekusi instruksi tanpa menggunakan *pipelining*. Teknik *pipelining* dua arah dengan fase I dan fase E memungkinkan dua buah instruksi yang berlainan dapat dilakukan bersamaan (Gambar 2.5b). Penggunaan sebuah memori *single-port* dimana hanya sebuah akses memori yang diizinkan pada setiap fasanya seperti pada Gambar 2.5b dapat ditingkatkan dengan mengizinkan dua buah akses memori tiap fasenya (menggunakan asumsi *read-after-write*).

Cara ini menghasilkan rangkaian seperti yang digambarkan pada Gambar 2.5c dan disebut pipelining tiga arah. (Stallings, 1997, pp141-142)

Idealnya *pipeline* dengan n tahap harus n kali lebih cepat dibandingkan prosesor tanpa *pipeline* (atau dibandingkan *pipeline* satu tahap) (http://cse.stanford.edu/class/sophomore-college/projects-00/risc/pipelining/index.html, 2003; Stallings, 1997, p143). Proses *pipeline* akan dapat bekerja baik apabila setiap fasenya memiliki durasi yang sama atau hampir sama.



Gambar 2.5 Jenis-jenis *Pipeline*

2.2.1.3 Waktu Eksekusi dalam Satu Siklus dengan Set Instruksi yang Sederhana

Arsitektur RISC mengharuskan bahwa sebuah instruksi dijalankan dalam satu siklus mesin atau disebut satu CPI (C*lock Per Instruction*). Tujuannya adalah untuk optimasi setiap instruksi pada CPU. Siklus mesin ditentukan oleh waktu yang digunakan untuk mengambil dua buah operand dari *register*, melakukan operasi ALU, dan menyimpan hasil operasinya ke dalam *register*. Dengan menggunakan instruksi sederhana atau instruksi satu siklus, maka dibutuhkan sedikit *microcode* atau tidak sama sekali. Oleh karena itu instruksi mesin dapat dilaksanakan secara *hardwired*. (Stallings, 1997, p136)

Kesederhanaan instuksi pada arsitektur RISC meliputi kesederhaan *address mode* sehingga dapat menyederhanakan kumpulan instruksi dan unit kontrol. Format instruksi yang digunakan juga sederhana (hanya sebuah atau beberapa *format* saja yang digunakan). Yang diikuti dengan panjang instruksi tetap dan disamakan dengan panjang *word*. Lokasi *field*, khususnya *opcode* juga tetap sehingga pengkodean *opcode* dan pengaksesan *operand register* dapat dilakukan secara bersamaan. (Stallings, 1997, p137)

2.2.2 Ciri-ciri Sistem RISC

Secara umum sebuah prosesor dengan arsitektur RISC mempunyai ciri-ciri sebagai berikut (Stallings, 1997, p139):

- Instruksi berukuran tunggal.

- Ukuran yang umum adalah 4 byte.
- Jumlah mode pengalamatan sedikit, biasanya kurang dari lima buah.
- Tidak terdapat pengalamatan tidak langsung (*indirect addressing*) yang mengharuskan pengguna melakukan sebuah akses memori agar memperolah alamat *operand* lainnya di dalam memori.
- Tidak terdapat operasi yang menggabungkan operasi *load/store* dengan operasi aritmatika.
- Tidak terdapat lebih dari satu *operand* beralamat memori per instruksi.
- Tidak mendukung perataan sembarang, bagi data untuk operasi *load/store*.
- Jumlah maksimum pemakaian *Memory Management Unit* (MMU) bagi suatu alamat data adalah sebuah instruksi.
- Jumlah bit bagi *integer register specifier* sama dengan lima atau lebih. Ini berarti bahwa sedikitnya 32 buah *register integer* dapat direferensikan sekaligus secara ekplisit.
- Jumlah bit *floating point register* specifier sama dengan empat atau lebih. Ini berarti bahwa sedikitnya 16 buah *register floating point* dapat direferensikan secara ekplisit sekaligus.
- Kompleksitas *hardware* dilimpahkan pada *software* (*compiler*).(http:// teachweb.cis.uoguelph.ca/cs405/RISC.htm, 2003)

2.3 Rancangan Set Instruksi

Kumpulan instruksi (*instruction set*) merupakan alat bagi pemrogram untuk mengontrol CPU. Kumpulan instruksi menentukan banyak fungsi yang akan dilakukan oleh CPU dan karena itu memiliki efek yang sangat menentukan pada implementasi pembuatan CPU. Masalah rancangan fundamental yang paling penting dalam merancang kumpulan instruksi meliputi (Stallings,1997, p9):

1. *Operation Repertoire* : Berapa banyak dan operasi-operasi apa yang harus disediakan, dan sekompleks apakah operasi itu seharusnya. Jenis-jenis instruksi dapat dikelompokan menjadi (Stallings, 1997, p6):
   - Pengolahan Data (*Data Processing*) : Instruksi-instruksi aritmatika dan logika.
   - Penyimpanan Data (*Data Storege*) : Instruksi-instruksi memori.
   - Pemindahan Data (*Data Movement*) : Instruksi I/O.
   - Kontrol (*Control*) : Instruksi pemeriksaan dan percabangan.
2. Jenis Data (*Data Types*) : Berbagai jenis data pada saat operasi dijalankan.
3. Format Instruksi (*Instruction Format*) : Panjang instruksi (dalam *bit*), jumlah alamat, ukuran *field*, dsb. Instruksi dibagi menjadi beberapa *field-field*, *field-field* ini berkaitan dengan elemen-elemen yang akan mengisi instruksi. Layout instruksi ini dikenal sebagai format instruksi (*instruction format*). (Stallings, 1997, p4)
4. *Register* : Jumlah *register* CPU yang dapat direferensikan oleh instruksi, dan fungsinya.

5. Pengalamatan (*Addressing*) : Mode untuk menspesifikasikan alamat suatu operand.

2.4 Format Instruksi dan Mode Pengalamatan

Format-format instruksi MIPS / *Microprocessor without Interlocking Pipeline Stages* (merupakan prosesor dengan arsitektur RISC) (Waldron, 1999, pp59-60):

- *Register Type* : Instruksi aritmatika, logika, dan shift yang mengambil data dari *register* menggunakan format ini.
- *Immediate Type* : LOAD, STORE, percabangan bersyarat, PC-*relative addressing*, instruksi aritmatika, logika, serta shift yang mengambil nilai langsung menggunakan format ini.
- *Jump Type* : Percabangan tidak bersyarat menggunakan format ini.

Mode-mode pengalamatan pada MIPS (Waldron, 1999, pp60-64):

- *Register* : Menggunakan format *register type*. Melakukan operasi *register to register*. Merupakan mode pengalamatan yang paling sederhana dan eksekusi yang paling cepat karena menghindari penundaan yang berhubungan dengan akses memori.
- *Base* : Menggunakan format *immediate type*. Berguna pada pemrograman struktur data (*Data Structure*). Struktur data yang sangat penting pada pemrograman komputer adalah *record* yang disebut *structure* pada bahasa pemrograman C. *Record* merupakan kumpulan dari variabel yang diperlakukan sebagai sebuah kesatuan (*unit*).

Pada '*base register addressing*', *base* adalah *register* dan *offset* adalah bagian dari instruksi. Untuk menaruh *base* pada *register* digunakan *register* sebagai *pointer* ke struktur. Karena itu dibutuhkan perintah *load* dari *pointer* (alamat) ke dalam *register*.

Base addressing : `la R1,kata`

`lw R2,(R1)`

Untuk mengakses elemen *array* yang posisinya pada urutan tertentu yang ditunjuk oleh indexnya, maka base yang biasanya merupakan nilai awal dari array dijumlahkan dengan nilai indexnya. Pengalamatan ini disebut juga pengalamatan index (*Indexed Addressing*).

- *Immediate* : Menggunakan format *immediate type*. Berfungsi pada operasi yang membutuhkan nilai konstan.
- PC-*relative* : Menggunakan format *immediate type*. Berguna untuk percabangan bersyarat.

2.5 *Pipelining Hazard*

Hazard merupakan masalah pewaktuan yang timbul karena eksekusi sebuah instruksi dengan *pipeline* tertunda selama satu atau lebih siklus *clock* dari waktu saat instruksi yang mengandung operasi diambil. (Mano dan Kime, 2001, p551)

Beberapa masalah yang timbul dengan adanya *pipelining*, yaitu *Data Depedensi* terjadi ketika instruksi bergantung pada hasil dari instruksi sebelumnya (http://cse.stanford.edu/class/sophomore-college/projects-00/risc/pipelining/index.html, 2003). Masalah ini dapat diatasi dengan menggunakan

instruksi *No-operation* (NOP) baik secara *software* maupun *hardware*, menyelipkan instruksi lain diantaranya, atau dengan *register forwarding*/*register file bypass* (Mano dan Kime, 2001, pp552-558).

Masalah lain yang timbul adalah *control hazard*. *Control hazard* terjadi pada saat penggunaan instruksi percabangan, dimana instruksi setelah instruksi percabangan (yang berada pada *delay slot*) akan dieksekusi sebelum kontrol melewati target percabangan, konsep ini disebut juga dengan *Delay Branch* (Hewlett-Packard Company, 1994, p4-7). *Control hazard* dapat diatasi dengan penambahan instruksi NOP secara *software* (oleh programmer atau kompiler) dan secara *hardware*, maupun dengan metoda *branch prediction* (Mano dan Kime, 2001, pp558-562). Dapat juga dilakukan dengan metoda *delayed branch* yang teroptimasi (menaruh instruksi yang mendahului instruksi percabangan pada *delay slot* tetapi tidak mempengaruhi hasil dari percabangan) (Stallings, 1997, p144).

2.6 *Little/Big-Endian*

Little/Big-Endian berkaitan dengan bagaimana byte-byte dalam sebuah *word* dan bit-bit dalam suatu byte direferensikan dan direpresentasikan. *Litle-endian* mengurutkan dari bit terbesar di sebelah kiri menuju bit terkecil sebelah kanan, sehingga bit ke nol berada di sebelah kanan. *Big-endian* mengurutkan bit terkecil di sebelah kiri menuju bit terbesar di sebelah kanan, sehingga bit ke nol berada disebelah kiri. (Stallings, 1997, p47)

2.7 *Stack*

Operasi stack menyimpan data dari *register* ke dalam memori data secara LIFO (*Last In First Out*) atau dikenal juga sebagai *pushdown list* (Stallings, 1997, p42). *Stack* digunakan dalam operasi *rekursif* (fungsi yang memanggil dirinya sendiri) dan digunakan dalam menyelesaikan ekspresi aritmatika. *Stack* disimpan di memori data pada alamat yang ditunjuk oleh *stack pointer*. Instruksi PUSH digunakan untuk menempatkan *item* baru ke puncak *stack*, instruksi POP memindahkan satu *item* dari *stack* (Mano dan Kime, 2001, p485). *Stack* bekerja dengan cara melakukan penumpukkan, sehingga *stack pointer* harus dikurang pada instruksi PUSH dan ditambah pada instruksi POP.

2.8 Prosedur

Prosedur atau fungsi atau subrutin merupakan program komputer yang merupakan bagian dari program yang lebih besar dan melakukan tugas tertentu. Prosedure/subrutin merupakan teknik yang sangat penting untuk strukturisasi program (Waldron, 1999, p91). Karena subrutin dipanggil dari sembarang tempat, maka CPU harus menyimpan alamat kembalinya agar pengembalian dapat berlangsung dengan benar. Terdapat tiga tempat yang umum digunakan untuk menyimpan alamat kembali (Stallings, 1997, p26):

- *Register*

  Nilai alamat kembali disipan pada *register*, kemudian nilai *program counter* (PC) diisi oleh alamat tujuan. Setelah prosedur dikerjakan maka nilai PC diisi oleh alamat dari *register* kembali.

Reg ← PC + 1

PC ← Target

- Awal Subrutin (*Start of Subroutine*)

  Nilai alamat kembali disimpan pada awal subrutin.

  Target ← PC + 1

  PC ← Target + 1

- Puncak Stack (*Top of Stack*)

  Hanya cara ini yang memungkinkan dilakukannya *reentrant* (pembukaan beberapa CALL dalam waktu bersamaan). Penggunaan stack juga memungkinkan dilakukannya pelewatan parameter (*passing parameter*) dan pengembalian nilai (*return value*).

Ketika prosedur menggunakan lokal variabel, yang terbaik adalah menyimpan sebanyak mungkin variabel tersebut ke dalam *register*, karena akses ke *register* jauh lebih cepat dibandingkan memori. Masalahnya adalah ketika sebuah prosedur memanggil prosedur lain, *register* harus disimpan agar nilai yang dikandungnya tidak akan hilang. Ada dua strategi dasar untuk menyimpan isi *register* pada saat pemanggilan prosedur (Waldron, 1999, p93):

1. *Caller saves* : berarti kode yang membuat pemanggilan akan menyimpan nilai *register* pada stack. Keuntungannya adalah pemanggil tahu *register* mana yang akan digunakannya, jadi pemanggil tidak menyimpan nilai yang tidak digunakan saat kembali (*return*).

2. *Callee saves* : berarti prosedur yang dipanggil akan melakukan penyimpanan. Keuntungannya adalah yang dipanggil mengetahui *register* mana yang dibutuhkan dan hanya menyimpan *register* tersebut.

2.9 *Interrupt*

Program *interrupt* digunakan untuk menangani berbagai situasi yang dibutuhkan berangkat dari urutan program yang biasa. Sebuah program *interrupt* mengirim kendali dari program yang sedang berjalan ke layanan program lain sebagai hasil dari permintaan yang dibangkitkan dari luar maupun dalam. (Mano dan Kime, 2001, p500)

Tiga jenis *interrupt* utama yang menyebabkan perubahan pada eksekusi normal dari sebuah program adalah sebagai berikut (Mano dan Kime, 2001, pp502-504):

1. *External interrupt*

   Merupakan *interrupt* yang datang dari peralatan *input* atau *output* (I/O), dari peralatan pewaktuan, dari rangkaian yang memonitor *power supply*, atau dari sumber eksternal lainnya.

   Mikroinstruksi khusus yang melaksanakan *interrupt* adalah sebagai berikut:

SP ← SP - 1	Mengurangi *Stack Pointer*
M[SP] ← PC	Simpan *return address* pada stack
SP ← SP - 1	Mengurangi *Stack Pointer*
M[SP] ← PSR	Simpan *processor status word* pada stack
EI ← 0	*Reset enable-interrupt flip-flop*

INTACK ← 1 *Enable interrupt acknowledge*

PC ← IVAD Transfer *interrupt vector address* ke PC

Saat kembali dari program *interrupt*, maka yang dilakukan adalah POP *return address* ke PC dan mengembalikan nilai PSR (*processor status word*).

2. *Internal interrupt*

   Merupakan *interrupt* yang dibangkitkan dari penggunaan data atau instruksi yang ilegal atau salah. *Internal interrupt* disebut juga *traps*.

3. *Software interrupt*

   Merupakan *interrupt* yang dibangkitkan oleh eksekusi sebuah instruksi. *Software interrupt* adalah instruksi yang dipanggil secara khusus yang berlaku seperti sebuah *interrupt* dibandingkan sebuah pemanggilan prosedur.

2.10 Perhitungan *Speed Up* untuk *Pipelining*

Perhitungan peningkatan kecepatan (*speed up*) dengan metoda *Pipelining* dilakukan dengan rumus berikut (http://www.cs.berkeley.edu/~culler/cs252-s02/slides/lec01.pdf, 2004):

$$\text{Speed Up} = \frac{\text{Ideal CPI} \times \text{Pipeline Depth}}{\text{Ideal CPI} + \text{Pipeline Stall CPI}} \times \frac{\text{Cycle Time}_{\text{unpipelined}}}{\text{Cycle Time}_{\text{pipelined}}}$$

dimana CPI (*cycle per instruction*) untuk *pipeline* adalah:

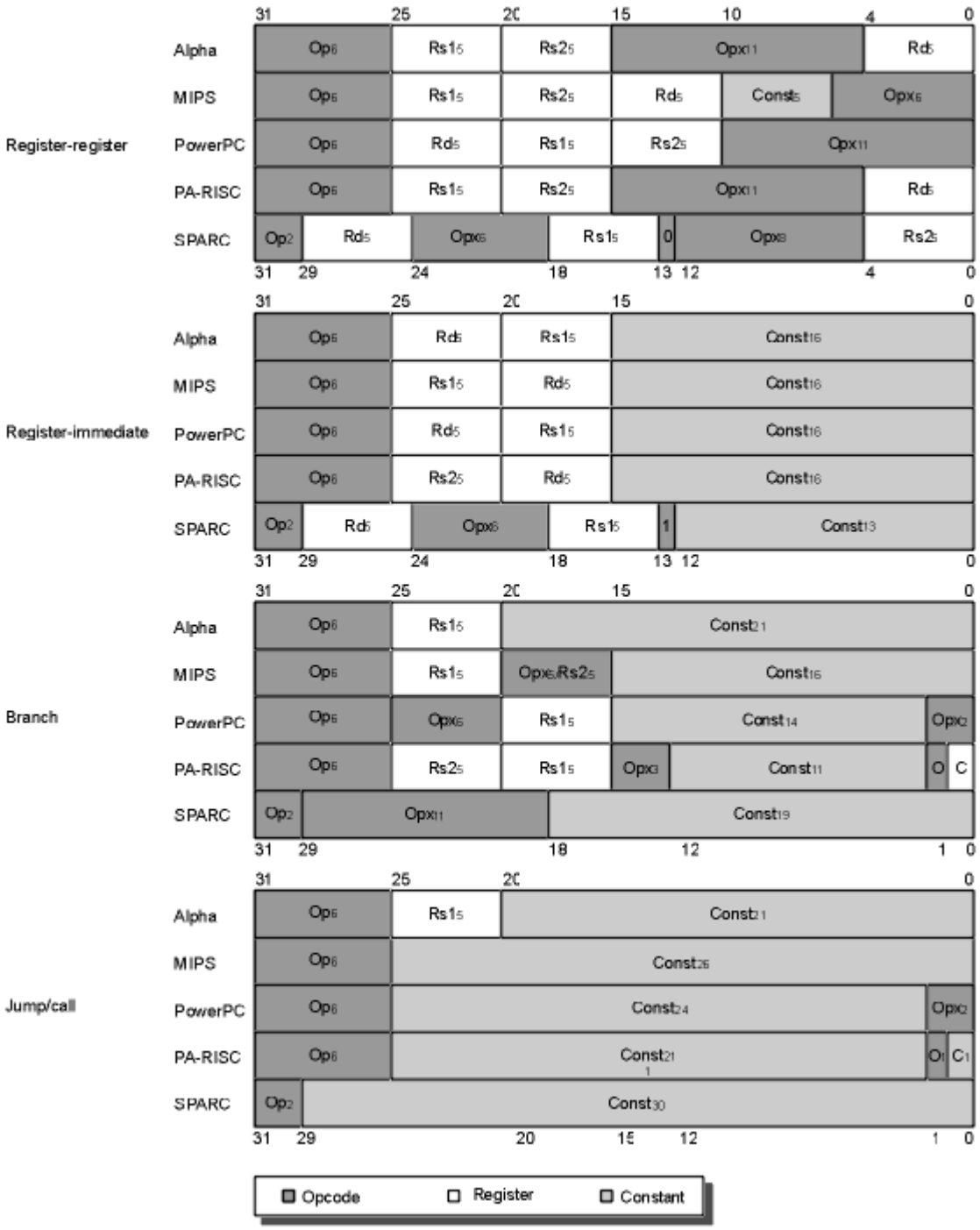
$$\text{CPI}_{\text{pipelined}} = \text{Ideal CPI} + \text{Average Pipeline Stall CPI}$$

2.11 Survei Arsitektur RISC

Survei arsitektur RISC berdasarkan beberapa prosesor RISC komersial (Tabel 2.1 sampai Tabel 2.7 dan Gambar 2.6). (ftp://ftp.mkp.com/COD2e /Web_ Extensions/survey.htm, 2003)

Tabel 2.1 Ringkasan dari Lima Prosesor dengan Arsitektur RISC

	Alpha	**MIPS I**	**PA-RISC 1.1**	**PowerPC**	**SPARC V8**
Date announced	1992	1986	1986	1993	1987
Instruction size (bits)	32	32	32	32	32
Address space (size, model)	64 bits, flat	32 bits, flat	48 bits, segmented	32 bits, flat	32 bits, flat
Data alignment	Aligned	Aligned	Aligned	Unaligned	Aligned
Data addressing modes	1	1	5	4	2
Protection	Page	Page	Page	Page	Page
Minimum page size	8 KB	4 KB	4 KB	4 KB	8 KB
Integer registers (number, model, size)	31 GPR x 64 bits	31 GPR x 32 bits	31 GPR x 32 bits	32 GPR x 32 bits	31 GPR x 32 bits
Separate floating-point registers	31 x 32 or 31 x 64 bits	16 x 32 or 16 x 64 bits	56 x 32 or 28 x 64 bits	32 x 32 or 32 x 64 bits	32 x 32 or 32 x 64 bits
Floating-point format	IEEE 754 single, double	IEEE 754 single, double	IEEE 754 single, double	IEEE 754 single, double	IEEE 754 single, double



Gambar 2.6 Format Instruksi dari Lima Prosesor dengan Arsitektur RISC

Tabel 2.2 Ringkasan Mode Pengalamatan Data dari Lima Prosesor dengan Arsitektur RISC

Addressing mode	**Alpha**	**MIPS V**	**PA-RISC 2.0**	**PowerPC**	**SPARC V9**
Register + offset (displacement or based)	X	X	X	X	X
Register + register (indexed)	---	X (FP)	X (Loads)	X	X
Register + scaled register (scaled)	---	---	X	---	---
Register + offset and update register	---	---	X	X	---
Register + register and update register	---	---	X	X	---

Tabel 2.3 Ringkasan Ketetapan Eksekusi dari Lima Prosesor dengan Arsitektur RISC

Format: instruction category	**Alpha**	**MIPS V**	**PA-RISC 2.0**	**PowerPC**	**SPARC V9**
Branch: all	Sign	Sign	Sign	Sign	Sign
Jump/call: all	Sign	---	Sign	Sign	Sign
Register-immediate: data transfer	Sign	Sign	Sign	Sign	Sign
Register-immediate: arithmetic	Zero	Sign	Sign	Sign	Sign
Register-immediate: logical	Zero	Zero	---	Zero	Sign

Tabel 2.4 Ringkasan Instruksi Perpindahan Data dari Lima Prosesor dengan Arsitektur RISC

Data transfer (instruction formats)	**R-I**	**R-I**	**R-I, R-R**	**R-I, R-R**	**R-I, R-R**
Instruction name	**Alpha**	**MIPS V**	**PA-RISC 2.0**	**PowerPC**	**SPARC V9**
Load byte signed	LDBU; SEXTB	LB	LDB; EXTRW,S 31,8	LBZ; EXTSB	LDSB
Load byte unsigned	LDBU	LBU	LDB, LDBX, LDBS	LBZ	LDUB
Load halfword signed	LDWU; SEXTW	LH	LDH; EXTRW,S 31,16	LHA	LDSH
Load halfword unsigned	LDWU	LHU	LDH, LDHX, LDHS	LHZ	LDUH
Load word	LDLS	LW	LDW, LDWX, LDWS	LW	LD
Load SP float	LDS*	LWC1	FLDWX, FLDWS	LFS	LDF
Load DP float	LDT	LDC1	FLDDX, FLDDS	LFD	LDDF
Store byte	STB	SB	STB, STBX, STBS	STB	STB
Store halfword	STW	SH	STH, STHX, STHS	STH	STH
Store word	STL	SW	STW, STWX, STWS	STW	ST
Store SP float	STS	SWC1	FSTWX, FSTWS	STFS	STF
Store DP float	STT	SDC1	FSTDX, FSTDS	STFD	STDF
Read, write special registers	MF_, MT_	MF, MT_	MFCTL, MTCTL	MFSPR, MF_, MTSPR, MT_	RD, WR, RDPR, WRPR, LDXFSR, STXFSR
Move integer to FP register	ITOFS	MFC1/ DMFC1	STW; FLDWX	STW; LDFS	ST; LDF
Move FP to integer register	FTTOIS	MTC1/ DMTC1	FSTWX; LDW	STFS; LW	STF; LD

Tabel 2.5 Ringkasan Instruksi Aritmatika dan Logika dari Lima Prosesor dengan Arsitektur RISC

Arithmetic, logical (instruction formats)	**R-R, R-I**	**R-R, R-I**	**R-R, R-I**	**R-R, R-I**	**R-R, R-I**
Instruction name	**Alpha**	**MIPS V**	**PA-RISC 2.0**	**PowerPC**	**SPARC V9**
Add	ADDL	ADDU, ADDU	ADDL, LD0, ADDI, UADDCM	ADD, ADDI	ADD
Add (trap if overflow)	ADDLV	ADD, ADDI	ADDO, ADDIO	ADDO; MCRXR; BC	ADDcc; TVS
Sub	SUBL	SUBU	SUB, SUBI	SUBF	SUB
Sub (trap if overflow)	SUBLV	SUB	SUBTO, SUBIO	SUBF/oe	SUBcc; TVS
Multiply	MULL	MULT, MULTU	SHiADD; ...; (i=1,2,3)	MULLW, MULLI	MULX
Multiply (trap if overflow)	MULLV	---	SHiADDO; ...;	---	---
Divide	---	DIV, DIVU	DS; ...; DS	DIVW	DIVX
Divide (trap if overflow)	---	---	---	---	---
And	AND	AND,ANDI	AND	AND, ANDI	AND
Or	BIS	OR, ORI	OR	OR, ORI	OR
Xor	XOR	XOR,XORI	XOR	XOR, XORI	XOR
Load high part register	LDAH	LUI	LDIL	ADDIS	SETHI (Bfmt.)
Shift left logical	SLL	SLLV, SLL	DEPW, Z 31-i,32-i	RLWINM	SLL
Shift right logical	SRL	SRLV, SRL	EXTRW, U 31, 32-i	RLWINM 32-i	SRL
Shift right arithmetic	SRA	SRAV, SRA	EXTRW, S 31, 32-i	SRAW	SRA
Compare	CMPEQ, CMPLT, CMPLE	SLT/U, SLTI/U	COMB	CMP(I)CLR	SUBcc r0,...

Tabel 2.6 Ringkasan Instruksi Kontrol dari Lima Prosesor dengan Arsitektur RISC

Control (instruction formats)	**B, J/C**	**B, J/C**	**B, J/C**	**B, J/C**	**B, J/C**
Instruction name	**Alpha**	**MIPS V**	**PA-RISC 2.0**	**PowerPC**	**SPARC V9**
Branch on integer compare	B_ (<, >, <=, >=, =, not=)	BEQ, BNE, B_Z (<, >, <=, >=)	COMB, COMIB	BC	BR_Z, BPcc (<, >, <=, >=, =, not=)
Branch on floating-point compare	FB_(<, >, <=, >=, =, not=)	BC1T, BC1F	FSTWX f0; LDW t; BB t	BC	FBPfcc (<, >, <=, >=, =, ...)
Jump, jump register	BR, JMP	J, JR	BL r0, BLR r0	B, BCLR, BCCTR	BA, JMPL r0,...
Call, call register	BSR	JAL, JALR	BL, BLE	BL, BLA, BCLRL, BCCTRL	CALL, JMPL
Trap	CALL_PAL GENTRAP	BREAK	BREAK	TW, TWI	Ticc, SIR
Return from interrupt	CALL_PAL REI	JR; ERET	RFI, RFIR	RFI	DONE, RETRY, RETURN

Tabel 2.7 Kaidah yang Berlaku dari Lima Prosesor dengan Arsitektur RISC

Conventions	**Alpha**	**MIPS V**	**PA-RISC 2.0**	**PowerPC**	**SPARC V9**
Register with value 0	r31 (source)	r0	r0	r0 (addressing)	r0
Return address register	(any)	r31	r2, r31	link (special)	r31
No-op	LDQ_U r31, ...	SLL r0, r0, r0	OR r0, r0, r0	ORI r0, r0, #0	SETHI r0,0

Tabel 2.7 Kaidah yang Berlaku dari Lima Prosesor dengan Arsitektur RISC (lanjutan)

Conventions	**Alpha**	**MIPS V**	**PA-RISC 2.0**	**PowerPC**	**SPARC V9**
Move R-R integer	`BIS ..., r31, ...`	`ADD ..., r0, ...`	`OR ..., r0, ...`	`OR rx, ry, ry`	`OR ..., r0, ...`
Operand order	`OP Rs1, Rs2, Rd`	`OP Rd, Rs1, Rs2`	`OP Rs1, Rs2, Rd`	`OP Rd, Rs1, Rs2`	`OP Rs1, Rs2, Rd`