# Arsitektur dan Organisasi Komputer:
## RISC (Reduced Instruction Set Computer)

Ir. Heru Nurwarsito, M.Kom
Barlian Henryranu P, ST, MT
Eko Saksi Pramukantoro, S.Kom, M.Kom



13

SELF-PROPAGATING ENTREPRENEURIAL EDUCATION DEVELOPMENT

## 1. PENDAHULUAN

### 1.1 Pengantar

Terdapat begitu banyak jenis produk yang menamakan dirinya komputer, mulai dari mikroprosesor berkeping-tunggal yang harganya beberapa dollar sampai supercomputer yang harganya puluhan juta dollar. Keragaman ini bukan saja ditunjukkan dari harganya, melainkan juga dalam ukuran, kinerja, dan aplikasinya. Walaupun perubahan di bidang komputer amat cepat dan bervariasi, beberapa konsep dasar tetap secara konsisten dapat diterapkan. Yang pasti, aplikasi dari konsep-konsep ini tergantung pada sasaran kinerja yang diinginkan perancang.

### 1.2 Tujuan

Tujuan dari modul ini adalah membahas beberapa kemajuan yang besar dalam bidang komputer dengan adanya inovasi RICS yang merupakan kemajuan yang sangat dramatis dalam fase sejarah arsitektur CPU, mengetahui karakteristik antara RISC DAN CICS, mengetahui perbedaan yang siginifikan antara pendekatan RISC dan CICS, mengetahui beberapa contoh prosesor dari RISC dan CICS, mengetahui beberapa kontroversi antara RISC dan CICS yang tiap tahun selalu berkembang karena semakin konvergensinya teknologi.

### 1.3 Latar Belakang

Salah satu penemuan penting di dalam organisasi dan arsitektur komputer saat ini adalah reduced set komputer (RISC). Arsitektur RISC merupakan perubahan dramatis pada kecenderungan arsitektur prosesor. Analisis pendekatan ini membawa membawa kita ke beberapa fokus masalah penting dalam organisasi dan arsitektur komputer. Pada modul bab RISC ini akan membahas tentang pendekatan RISC dan perbandingannya dengan pendekatan complex instruction set computer (CISC).

Ditinjau dari jenis set instruksinya, ada 2 jenis arsitektur komputer, yaitu:

1. Arsitektur komputer dengan kumpulan perintah yang sederhana (Reduced Instruction Set Computer = RISC)
2. Arsitektur komputer dengan kumpulan perintah yang rumit (Complex Instruction Set Computer = CISC)

## 2. Karakteristik Eksekusi Instruksi

### A. RISC

**Pengertian**

RISC singkatan dari Reduced Instruction Set Computer.Merupakan bagian dari arsitektur mikroprosessor, berbentuk kecil dan berfungsi untuk negeset istruksi dalam komunikasi diantara arsitektur yang lainnya.

**Karakteristik**

Arsitektur RISC memiliki beberapa karakteristik diantaranya :

1. Siklus mesin ditentukan oleh waktu yang digunakan untuk mengambil dua buah operand dari register, melakukan operasi ALU, dan menyimpan hasil operasinya kedalam register, dengan demikian instruksi mesin RISC tidak boleh lebih kompleks dan harus dapat mengeksekusi secepat mikroinstruksi pada mesin-mesin CISC. Dengan menggunakan instruksi sederhana atau nstruksi satu siklus hanya dibutuhkan satu mikrokode atau tidak sama sekali, instruksi mesin dapat dihardwired. Instruksi seperti itu akan dieksekusi lebih cepat dibanding yang sejenis pada yang lain karena tidak perlu mengakses penyimapanan kontrol mikroprogram saat eksekusi instruksi berlangsung.

2. Operasi berbentuk dari register-ke register yang hanya terdiri dari operasiload dan store yang mengakses memori . Fitur rancangan inimenyederhanakan set instruksi sehingga menyederhanakan pula unit control.Keuntungan lainnya memungkinkan optimasi pemakaian register sehinggaoperand yang sering diakses akan tetap ada di penyimpan

berkecepatantinggi. Penekanan pada operasi register ke register merupakan hal yang unikbagi perancangan RISC.

3. Penggunaan mode pengalamatan sederhana, hampir sama dengan instruksimenggunakan pengalamatan register,.Beberapa mode tambahan sepertipergeseran dan pe-relatif dapat dimasukkan selain itu banyak modekompleks dapat disintesis pada perangkat lunak dibanding yang sederhana,selain dapat menyederhanakan sel instruksi dan unit kontrol.

4. Penggunaan format-format instruksi sederhana, panjang instruksinya tetapdan disesuaikan dengan panjang word. Fitur ini memiliki beberapa kelebihankarena dengan menggunakan field yang tetap pendekodean opcode danpengaksesan operand register dapat dilakukan secara bersama-sama

**Ciri-Ciri**

1. Instruksi berukuran tunggal
2. Ukuran yang umum adalah 4 byte
3. Jumlah pengalamatan data sedikit, biasanya kurang dari 5 buah.
4. Tidak terdapat pengalamatan tak langsung yang mengharuskan melakukan sebuah akses memori agar memperoleh alamat operand lainnya dalam memori
5. Tidak terdapat operasi yang menggabungkan operasi load/store dengan operasi aritmatika, seperti penambahan ke memori dan penambahan dari memori.
6. Tidak terdapat lebih dari satu operand beralamat memori per instruksi7. Tidak mendukung perataan sembarang bagi data untuk operasi load/ store
7. Jumlah maksimum pemakaian memori manajemen bagi suatu alamat data adalah sebuah instruksi .
8. Jumlah bit bagi integer register spesifier sama dengan 5 atau lebih, artinya sedikitnya 32 buah register integer dapat direferensikan sekaligus secara eksplisit.
9. Jumlah bit floating point register spesifier sama dengan 4 atau lebih, artinya sedikitnya 16 register floating point dapat direferensikan sekaligus secara eksplisit.

Beberapa prosesor implementasi dari arsiteketur RISC adalah AMD29000, MIPS R2000, SPARC, MC 88000, HP PA, IBM RT/TC, IBM RS/6000,intel i860, Motorola 88000 (keluarga Motorola), PowerPC G5.

**Pendekatan**

Prosesor RISC hanya menggunakan instruksi-instruksi sederhana yang bisadieksekusi dalam satu siklus. Dengan demikian, instruksi 'MULT' sebagaimana dijelaskansebelumnya dibagi menjadi tiga instruksi yang berbeda, yaitu "LOAD", yang digunakanuntuk

memindahkan data dari memori ke dalam register, "PROD", yang digunakan untuk melakukan operasi produk (perkalian) dua operan yang berada di dalam register (bukan yangada di memori) dan "STORE", yang digunakan untuk memindahkan data dari register kembali ke memori. Berikut ini adalah urutan instruksi yang harus dieksekusi agar yangterjadi sama dengan instruksi "MULT" pada prosesor RISC (dalam 4 baris bahasa mesin):

```
LOAD   A,    2:3
LOAD   B,    5:2
PROD   A,    B
STORE  2:3,  A
```

Pada awalnya memang tidak kelihatan efisien.Hal ini dikarenakan semakin banyak barisinstruksi, semakin banyak lokasi RAM yang dibutuhkan untuk menyimpan instruksi-instruksitersebut.Kompailer juga harus melakukan konversi dari bahasa tingkat tinggi ke bentuk kodeinstruksi 4 baris tersebut.Strategi pada RISC ini memberikan beberapa kelebihan.Karena masing-masinginstruksi hanya membutuhkan satu siklus detak untuk eksekusi, maka seluruh program (yangsudah dijelaskan sebelumnya) dapat dikerjakan setara dengan kecepatan dari eksekusiinstruksi "MULT". Secara perangkat keras, prosesor RISC tidak terlalu banyak membutuhkan transistor dibandingkan dengan CISC, sehingga menyisakan ruangan untuk register-register serbaguna (general purpose registers). Selain itu, karena semua instruksidikerjakan dalam waktu yang sama (yaitu satu detak), maka dimungkinkan untuk melakukan pipelining. Memisahkan instruksi "LOAD" dan "STORE" sesungguhnya mengurangi kerja yangharus dilakukan oleh prosesor. Pada CISC, setelah instruksi "MULT" dieksekusi, prosesor akan secara otomatis menghapus isi register, jika ada operan yang dibutuhkan lagi untuk operasi berikutnya, maka prosesor harus menyimpan-ulang data tersebut dari memori keregister. Sedangkan pada RISC, operan tetap berada dalam register hingga ada data lain yangdisimpan ke dalam register yang bersangkutan.

## B. CISC

### Pengertian

CISC adalah singkatan dari *Complex Intruction Set Computer* dimana prosesor tersebut memiliki set instruksi yang kompleks dan lengkap. Sedangkan **RISC** adalah singkatan dari *Reduced Instruction Set Computer* yang artinya prosesor tersebut memiliki set instruksi program yang lebih sedikit. Karena perbedaan keduanya ada pada kata set instruksi yang kompleks atau sederhana (reduced).

**Karakteristik**

Arsitektur CISC memiliki beberapa karakteristik diantaranya :

1. Sarat informasi memberikan keuntungan di mana ukuran program-program yang dihasilkan akan menjadi relatif lebih kecil, dan penggunaan memory akan semakin berkurang. Karena CISC inilah biaya pembuatan komputer pada saat itu (tahun 1960) menjadi jauh lebih hemat
2. Dimaksudkan untuk meminimumkan jumlah perintah yang diperlukan untuk mengerjakan pekerjaan yang diberikan. (Jumlah perintah sedikit tetapi rumit) Konsep CISC menjadikan mesin mudah untuk diprogram dalam bahasa rakitan

**Ciri-Ciri**

Instruksi berukuran tunggal

1. Ukuran yang umum adalah 4 byte.
2. Jumlah mode pengalamatan data yang sedikit, biasanya kurang dari lima buah.
3. Tidak terdapat pengalamatan tak langsung.
4. Tidak terdapat operasi yang menggabungkan operasi load/store dengan operasi aritmetika (misalnya, penambahan dari memori, penambahan ke memori).

**Pendekatan**

Tujuan utama dari arsitektur CISC adalah melaksanakan suatu perintah cukup dengan beberapa baris bahasa mesin sedikit mungkin. Untuk tujuan contoh kita kali ini, sebuah prosesor CISC sudah dilengkapi dengan sebuah instruksi khusus, yang kita beri nama MULT.Saat dijalankan, instruksi akan membaca dua nilai dan menyimpannya ke 2 register yag berbeda, melakukan perkalian operan di unit eksekusi dan kemudian mengambalikan lagihasilnya ke register yang benar.

**MULT 2:3, 5:2**

MULT dalam hal ini lebih dikenal sebagai "complex instruction", atau instruksi yangkompleks. Bekerja secara langsung melalui memori komputer dan tidak memerlukaninstruksi lain seperti fungsi baca maupun menyimpan.Satu kelebihan dari sistem ini adalah kompailer hanya menerjemahkan instruksi-instruksi bahasa tingkat-tinggi ke dalam sebuah bahasa mesin. Karena panjang kode instruksi relatif pendek, hanya sedikit saja dari RAM yang digunakan untuk menyimpan instruksi-instruksi tersebut.

**Karakteristik dari beberapa Prosesor CISC, RISC, dan Superskalar**

| Karakteritik | CISC | | | RISC | | Superskalar | |
|---|---|---|---|---|---|---|---|
| | IBM 370/168 | VAX 11/780 | Intel 80486 | Motorola 88000 | MIPS R4000 | IBM RS/System 6000 | Intel 80960 |
| Tahun dibuat | 1973 | 1978 | 7989 | 1988 | 7991 | 1990 | 1989 |
| Jumlah instruksi | 208 | 303 | 235 | 51 | 94 | 184 | 62 |
| Instruksi (Bytes) | 2-6 | 2-57 | 1-11 | 4 | 32 | 4 | 4,8 |
| Mode Pengalamatan | 4 | 22 | 11 | 3 | 1 | 2 | 11 |
| Jumlah register general-purpose | 16 | 16 | 8 | 32 | 32 | 32 | 23-256 |
| Ukuran memori kontrol (Kbits) | 420 | 480 | 246 | - | - | - | - |
| Ukuran Cache (Kbytes) | 64 | 64 | 8 | 16 | 128 | 32-64 | 0,5 |

**Tabel Karakteristik dari beberapa Prosesor CISC, RISC, dan Superskalar**

Sementara itu, sejumlah penelitian telah dilakukan dalam beberapa tahun untuk menentukan karakteristik dan pola eksekusi instruksi mesin yang dihasilkan dari program – program HLL. Hasil penelitian ini telah memberikan inspirasi kepada para peneliti untuk mencari pendekatan secara keseluruhan, yakni untuk membuat arsitekturyang mendukung HLL secara lebih sederhana, bukannya lebih kompleks.

Dengan demikian, untuk memahami RISC, kita akan mengalaminya dengan tinjauan singkat tentang karakteristik eksekusi instruksi. Aspek – aspek komputasi yang dimaksud adalah sebagai berikut :

1. *Operasi-operasi yang dilakukan* : Hal ini menentukan fungsi – fungsi yang akan dilakukan oleh CPU dan interaksinya dengan memori.
2. *Operand-operand yang digunakan* : Jenis –jenis operand dan frekuensi pemakaiannya akan menentukan organisasi memori untuk menyimpannya dan mode pengalamatan untuk mengaksesnya.
3. *Pengurutan eksekusi* : hal ini akan menentukan kontrol dan organisasi pipeline.

Semua hasil penelitian tersebut didasarkan pada pengukuran dinamik. Dengan kata lain, pengukuran – pengukuran dikumpulkan oleh pengeksekusian program dan penghitungan jumlah waktu pada dijalankannya fitur – fitur tertentu atau sifat – sifat tertentu dengan benar. Sebaliknya, pengukuran statik melakukan perhitungan itu pada teks sumber suatu program. Pengukuran – pengukuran statik ini tidak memberikan informasi tentang kinerja

yang bermanfaat, karena pengukuran ini tidak diberi bobot relatif terhadap jumlah waktu pengeksekusian seluruh statement.

**Operasi dengan High Level Language (HLL)**

Hasil-hasil penelitian yang dilakukan untuk menganalisis tingkah laku program-program HLL menunjukkan:

- Assignment Statement sangat menonjol diikuti statement bersyarat (IF, LOOP) lihat tabel dibawah ini.

Hasil-hasil penelitian ini merupakan hal yang sangat penting bagi perancangan set instruksi mesin.

| | Kejadian Dinamik | | Instruksi-Mesin Berbobot | | Referensi Memori Berbobot | |
|---|---|---|---|---|---|---|
| | Pascal | C | Pascal | C | Pascal | C |
| ASSIGN | 45 | 38 | 13 | 13 | 14 | 15 |
| LOOP | 5 | 3 | 42 | 32 | 33 | 26 |
| CALL | 15 | 12 | 31 | 33 | 44 | 45 |
| IF | 29 | 43 | 11 | 21 | 7 | 13 |
| GOTO | - | 3 | - | - | - | - |
| OTHER | 6 | 1 | 3 | 1 | 2 | 1 |

**Tabel Frekuensi Dinamik Relatif Berbobot dari Operasi-operasi HLL**

**Operand**

Penelitian Paterson telah memperhatikan [PATT82a] frekuensi dinamik terjadinya kelas-kelas variabel. Hasil yang konsisten diantara program pascal dan C menunjukkan mayoritas referensi menunjuk ke variable scalar. Lebih lanjut, lebih dari 80% skalar bersifat variabel lokal (terhadap prosedur). Selain itu, referensi – referensi ke array / struktur memerlukan referensi sebelumnya ke indeks atau pointer – nya, yang juga biasanya merupakan suatu skalar lokal. Dengan demikian, terdapat referensi ke skalar dalam jumlah yang sangat banyak, dan referensi – referensi ini sangat terlokalisir.

Penelitian ini telah menguji tingkah laku dinamik program HLL yang tidak tergantung pada arsitektur tertentu. Penelitian [LUND77] menguji instruksi DEC-10 dan secara dinamik menemukan setiap instruksi rata-rata mereferensi 0,5 operand dalam memori dan rata-rata mereferensi 1,4 register. Hasil yang sama telah dilaporkan dalam [HUCK83] untuk program – program C, Pascal, dan FORTRAN pada S/370, PDP-11, dan VAX. Tentu saja angka ini tergantung pada arsitektur dan kompiler namun sudah cukup menjelaskan frekuensi pengaksesan operand.

| | Pascal | C | Rata-rata |
|---|---|---|---|
| Konstanta Integer | 16 | 23 | 20 |
| Variabel Skalar | 58 | 53 | 55 |
| Array/Struktur | 26 | 24 | 25 |

**Tabel Persentase Dinamik Operand-operand**

Penelitian – penelitian tersebut menyatakan pentingnya arsitektur yang berpengaruh pada kecepatan pengaksesan operand, karena operasi ini sering kali dilakukan. Penelitian Patterson menyatakan bahwa kandidat utama untuk optimisasi adalah mekanisme penyimpanan dan pengaksesan variabel – variabel skalar lokal.

**Procedure Calls**

Dalam HLL procedure call dan return merupakan aspek penting karena merupakan operasi yang membutuhkan banyak waktu dalam program yang dikompalasi sehingga banyak berguna untuk memperhatikan cara implementasi opperasi ini secara efisien. Adapun aspeknya yang penting adalah jumlah
parameter dan variabel yang berkaitan dengan prosedur dan kedalaman pensarangan (nesting).

Pada penelitian yang dilakukan Tanenbaum [TANE78], telah ditemukan bahwa 98% prosedur yang dipanggil secara dinamik dilewatkan dengan argument yang jumlahnya lebih sedikit dari enam buah, dan bahwa 92% prosedur yang dipanggil tersebut menggunakan variabel skalar lokal yang jumlahnya lebih sedikit dari enam buah. Hasil penelitian tersebut menunjukkan bahwa jumlah word yang diperlukan untuk peraktivasi prosedur tidak besar. Penelitian mengindikasikan bahwa pada kenyataannya referensi – referensi itu dibatasi oleh variabel – variabel yang jumlahnya relatif sedikit.

Kelompok Barkeley juga meneliti pola prosedur call dan return di dalam program – program HLL. Mereka menemukan bahwa rangkaian prosedur call yang tidak diinterupsi dalam waktu yang lama yang diikuti oleh rangkaian prosedur return sangatlah jarang terjadi. Melainkan, mereka menemukan bahwa program tetap terbatasi oleh window kecil kedalaman prosedur – invocation. Hasil penelitian ini memperkuat kesimpulan bahwa referensi – referensi operand sangat bersifat lokal.

**Implikasi**

Hasil-hasil penelitian secara umum dapat dinyatakan bahwa terdapat tiga buah elemen yang menentukan karakter arsitektur RISC.

- Pertama, penggunaan register dalam jumlah yang besar. Hal ini dimaksudkan untuk mengoptimalkan pereferensian operand.
- Kedua, diperlukan perhatian bagi perancangan pipeline instruksi. Karena tingginya proporsi instruksi pencabangan bersyarat dan prosedur call, pipeline instruksi yang bersifat langsung dan ringkas akan menjadi tidak efisien.
- Ketiga, terdapat set instruksi yang disederhanakan (dikurangi).

Keinginan untuk mengimplementasikan keseluruhan CPU dalam keping tunggal akan merupakan solusi Reduced Instruction Set.

### 3. Penggunaan File Register Besar

Terdapat statement assignment yang jumlahnya banyak dalam program-program HLL,dan banyak diantaranya berupa statement assignment sederhana seperti A = B. Di samping itu,terdapat pula akses operand per statement HLL dalam jumlah yang cukup besar. Apabila kitamenghubungkan kedua di atas dengan kenyataan bahwa sebagian besar akses adalah menuju keskalar-skalar lokal, maka sangat mungkin diperlukan penyimpanan register yang besar. Alasan diperlukannya penyimpanan register adalah register merupakan perangkat penyimpanan yang paling cepat, yang lebih cepat dibandingkan dengan memori utama dan memori cache. Dimungkinkan untuk menerapkan dua buah pendekatan dasar, yaitu berdasarkan perangkat lunak dan perangkat keras.

1. Pendekatan perangkat lunak mengandalkan kompiler untuk memaksimalkan pemakaian register. Pendakatan ini membutuhkan algoritma analisis program yang canggih.
2. Pendekatan perangkat keras dilakukan hanya dengan memperbanyak jumlah register sehingga akan lebih banyak variabel yang dapat ditampung di dalam register dalam periode waktu yang lebih lama.

**Register Windows**

**Register Variabel Lokal**

Dengan menggunakan pendekatan diatas, penggunaan register dalam jumlah yang besar akan mengurangi kebutuhan mengakses memori. Dalam hal ini tugas perancangan adalah mengatur register-register sedemikian rupa sehingga tujuan dapat tercapai.

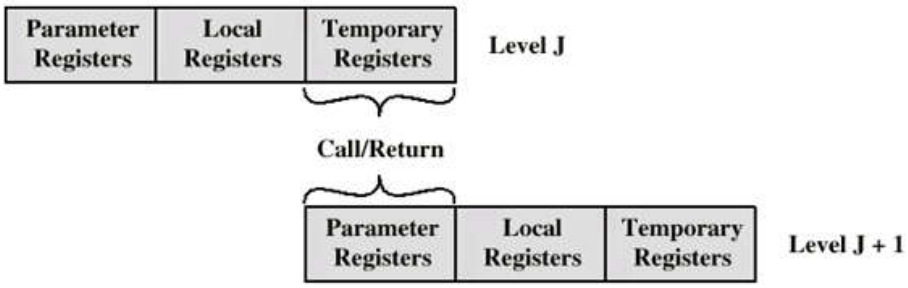
Karena sebagian referensi operand menuju ke skalar lokal, pendekatan yang harus dilakukan adalah dengan menyimpan referensi - referensi operand ini di dalam register, yang di sini mungkin beberapa register dicadangkan untuk variabel - variabel global. Masalah yang terjadi di sini adalah bahwa definisi lokal akan berubah setiap kali terjadi

prosedur call atau prosedur return, yang merupakan operasi - operasi yang sering kali terjadi. Pada setiap call variabel lokal harus disimpan dari register ke memori, sehingga register - register tersebut dapat dipakai kembali oleh program yang dipanggil. Di samping itu, parameter-parameter harus dilewatkan pula. Pada saat return, variabel – variabel parent program harus disimpan lagi (dimuatkan lagi ke dalam register) dan hasilnya harus dilewatkan lagi ke parent program.

Solusi ini didasarkan pada dua buah hasil lainnya yang telah dilaporkan pada Bagian 13.1. Pertama, prosedur tertentu hanya menggunakan beberapa parameter yang dilewatkan dan variabel lokal. Kedua, kedalaman aktivitas prosedur berfluktuasi dalam jangkauan yang cukup kecil. Untuk dapat memanfaatkan sifat – sifat tersebut, digunakan beberapa kelompok kecil register yang masing – masing kelompok tersebut di – assign ke prosedur – prosedur yang berlainan. Sebuah prosedur call secara otomatis mengalihkan CPU untuk menggunakan jendela register berukuran tetap yang lainnya, bukannya menyimpan register di dalam memori. Jendela – jendela prosedur yang berdampingan bertumapang tindih sehingga memungkinkan pelewatan parameter.

Pada suatu saat tertentu, hanya sebuah jendela register yang visibel dan dapat dialamati, seolah – olah jendela itu hanya satu – satunya kumpulan register. Jendela register dibagi menjadi tiga buah daerah yang berukuran tetap, yaitu :

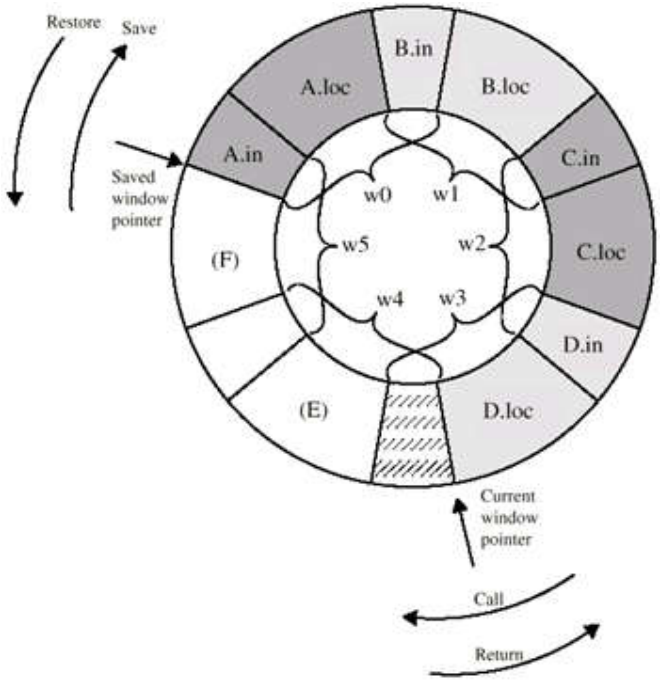
1. Register-register parameter
   Menampung parameter-parameter yang dilewatkan dari prosedur.
2. Register-register lokal
   Digunakan untuk variable lokal, setelah di-assign oleh kompiler.
3. Register-register temporer
   Digunakan untuk pertukaran parameter dan hasilnya dengan yang tingkatan yang lebih rendah berikutnya, jadi secara fisik register-register temporer yang berada dalam satu tingkatan sama seperti halnya register-register parameter pada tingkatan yang lebih rendah berikutnya. Overlap ini memungkinkan parameter-parameter dapat dilewatkan tanpa perpindahan aktual data.




**Overlapping Register Windows**

Untuk menangani bentuk call dan return yang mungkin terjadi , jumlah jendela register tidak boleh dibatasi. Melainkan jendela – jendela register dapat dipakai untuk menampung beberapa aktivasi prosedur yang terkini. Aktivasi – aktivasi yang lebih dulu harus disimpan di dalam memori dan kemudian digunakan kembali pada saat terjadi pengurangan kedalaman pensarangan. Dengan demikian, organisasi file register aktual merupakan buffer sirkuler jendela – jendela yang bertumpang tindih.

**Circular Buffer diagram**




**Gambar Circular Buffer diagram**

Organisasi tersebut ditunjukkan pada gambar, yang menggambarkan buffer sirkuler enam buah jendela. Buffer diisikan ke dalam 4 (A memanggil B, B memanggil C, C memanggil D), dengan prosedur D dalam keadaan aktif. Pointer jendela saat itu (Current Window Pointer - CWP) menunjuk ke jendela tempat prosedur yang sedang aktif. Referensi – referensi register oleh instruksi mesin di-offset-kan oleh pointer ini guna menentukan register fisik aktual. Pointer jendela yang disimpan akan mengidentifikasijendela yang paling akhir disimpan di

dalam memori. Apabila sekarang prosedur D memanggil prosedur E, argument – argument E akan disimpan di dalam register – register temporer D (tumpang tindih antara w3 dan w2) dan CWP dimajukan satu jendela.

Apabila kemudian prosedur E memanggil F, maka pemanggilan tidak dapat dilakukan dengan menggunakan status buffer saat itu. Hal ini disebabkan karena jendela prosedur F bertumpang tindih dengan jendela prosedur. Apabila F mulai memuatkan register – register temporernya, yaitu persiapan untuk melakukan sebuah call, maka register – register itu akan menindih register – register parameter A (A.In). Sehingga pada saat CWP dinaikkan (modulus 6) maka CWP akan sama dengan SWP, dan akan terjadi interrupt, dan jendela A akan disimpan. Hanya dua bagian pertama (A.In dan A.loc) yang perlu dusmpan. Kemudian SWP dinaikkan dan pemanggilan F dilakukan. Interrupt yang serupa dapat terjadi pada saat return. Misalnya, setelah aktivasi F dan pada saat B kembali ke A, CWP diturunkan dan akan sama dengan SWP. Hal ini akan menyebabkan terjadinya interrupt yang dapat mengakibatkan restorasi jendela A.

Dari keterangan di atas, diketahui bahwa file register N jendela hanya dapat menampung N – 1 buah aktivasi prosedur. Nilai N tidak perlu besar.

**Variabel-variabel Global**

Teknik Register Windows memberikan organisasi yang efisien untuk penyimpanan variable skalar lokal di dalam register. Akan tetapi teknik ini tidak dapat memenuhi kebutuhan penyimpanan variabel global, yang diakses oleh lebih dari sebuah prosedur (misalnya, variabel COMMON dalam FORTRAN).

Terdapat dua pilihan untuk memenuhi hal tersebut.

- Pertama,

Variabel-variabel yang dideklarasikan sebagai global pada HLL dapat disediakan lokasi-lokasi oleh kompiler.Namun, bagi yang sering mengakses variabel-variabel global, teknik tersebut tidaklah efisien.

- Alternatifnya adalah melibatkan kumpulan register global di dalam CPU. Register-register ini harus memiliki jumlah yang tetap dan dapat dipakai oleh semua prosedur.

## 4. Register vs Cache

| Large Register File | Cache |
|---|---|
| • Semua skalar lokal | Skalar lokal yang baru dipakai |
| • Variabel2 individual | Sekelompok memori |
| • Variabel2 global yg di-assign kompiler | Variabel global yang baru dipakai |
| • Save/restore tergantung prosedur | Save/restore tergantung algoritma penggantian cache |
| • Pengalamatan Regsiter | Pengalamatan Memori |

**Tabel Register vs Cache**

File register, yanh diorganisasikan menjadi dua jendela, berfungsi sebagai buffer kecil yang cepat untuk menampung subset seluruh variable yang memiliki kemungkinan besar akan banyak dipakai. Dari sudut pandang ini, file register berfungsi lebih menyerupai sebuah cache memori.

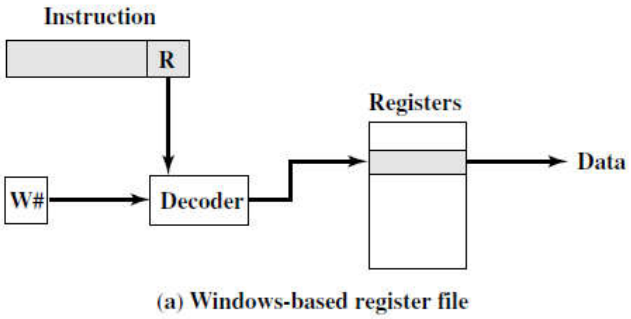
Gambar diatas membandingkan karakteristik kedua pendekatan tersebut. File register berbasis jendela menampung seluruh variabel skalar local (kecuali terjadi overflow jendela, yang jarang terjadi) *N* – 1 buah aktivasi prosedur yang terkini. File register dapat menghemat waktu, karena semua variabel skalar local dipertahankan. Sebaliknya, cache dapat melakukan pemakaian ruang yang lebih efisien, karena cache bereaksi terhadap situasi secara dinamis. Selain itu, umumnya cache memperlakukan seluruh referensi memori secara sama, termasuk instruksi dan data jenis lainnya. Sehingga penghematan dalam bidang lainnya dapat dimungkinkan apabila menggunakan cache dan bukan file register.

File register dapat tidak efisien dalam menggunakan ruang, karena tidak semua prosedur akan memerlukan ruang jendela sepenuhnya yang telah diberikan. Sebaliknya cache memiliki ketidakefisienan lainnya : Data akan dibaca ke dalam cache dalam bentuk blok – blok. Sementara file register hanya berisi variabel – variabel yang sedang digunakan, cache membaca suatu blok data, yang mungkin sebagian darinya tidak akan digunakan.

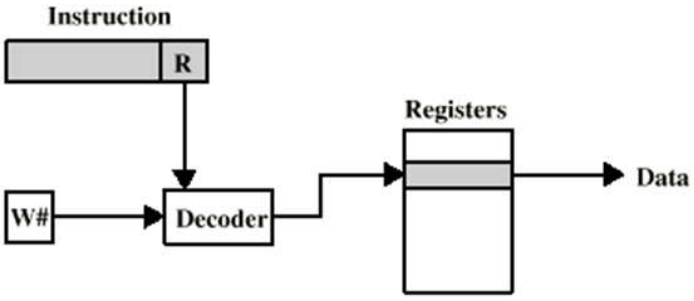
Cache memiliki kemampuan untuk menangani variabel global dan juga variabel lokal. Umumnya terdapat banyak skalar – skalar global, namun hanya beberapa skalar saja yang akan banyak digunakan [KATE83]. Sebuah cache akan secara dinamis akan menemukan variabel – variabel ini dan kemudian akan menyimpannya. Apabila file register berbasis jendela ditambahkan dengan register – register global, maka cache akan dapat juga menampung sejumlah skalar global. Namun compiler akan mengalami kesulitan untuk menentukan skalar global yang akan banyak dipakai.

Dengan menggunakan file register, pergerakan data antara register – register dengan memori ditentukan oleh kedalaman pensarangan prosedur. Karena biasanya kedalaman tersebut berfluktuasi dalam jangkauan yang kecil, maka penggunaan memori relatif jarang. Sebagian besar memori – memori cache bersifat set asosiatif dengan ukuran set yang kecil. Dengan demikian terdapat bahaya dengan data atau instruksi lainnya akan menindih (*overwrite*) variabel – variabel yang sering digunakan.

Berdasarkan pembahasan sebelumnya, sulit untuk memilih antara file register berbasis jendela dan cache. Namun terdapat sebuah karakteristik yaitu adanya kelebihan dengan pendekatan register dan adanya pendapat bahwa sistem berbasis cache akan lebih lambat. Perbedaan ini terlihat dengan membandingkan jumlah overhead pengalamatan kedua pendekatan tersebut.




(a) Windows-based register file




**Gambar Pereferensian Skalar Lokal**

Gambar di atas menjelaskan perbedaan kedua pendekatan tersebut. untuk mereferensi skalar lokal pada file register berbasis window, digunakan nomor register " virtual " dan nomor jendela (window). Untuk memilih salah satu register fisik, nomor ini dapat dilewatkan melalui decoder yang relatif sederhana. Untuk mereferensi lokasi memori di dalam cache, harus dibuat alamat memori yang memiliki lebar maksimal. Kompleksitas operasi tersebut bergantung pada mode pengalamatan. Pada set cache asosiatif, sejumlah alamat digunakan untuk membaca jumlah word dan tag yang sama dengan ukuran set. Alamat – alamat lainnya dibandingkan dengan tag – tag, dan akan diambil sebuah word yang telah dibaca. Maka jelas

walaupun cache memiliki kecepatan yang sama dengan file register, waktu akses akan lebih lama. Dengan demikian, dari sudut pandang kinerja, file register berbasis window memiliki kelebihan pada skalar lokal. Penambahan kinerja lebih lanjut dapat diperoleh dengan cara menambahkan cache untuk instruksi saja.

### 5. Optimasi Register Berbasis Kompiler

Sekarang kita asumsikan bahwa pada mesin RISC, target hanya tersedia register dalam jumlah yang sedikit (16 – 32 buah). Di sini, penggunaan register yang telah dioptimalkan tersebut merupakan tanggung jawab kompiler. Tentu saja sebuah program yang ditulis dalam bahasa tingkat tinggi tidak memiliki referensi eksplisit ke register. Kuantitas – kuantitas program diacu secara simbolis. Fungsi kompiler adalah untuk menjaga operand bagi komputasi sebanyak mungkin di dalam register, dan bukannya di dalam memori utama. Hal itu ditujukan untuk meminimalkan operasi load dan store.

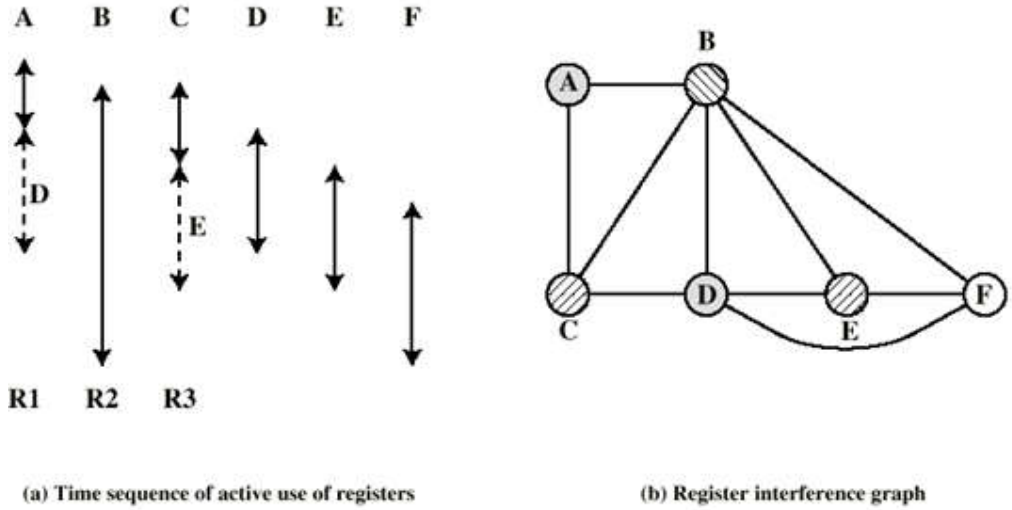
Setiap kuantitas program yang merupakan kandidat yang akan disimpan dalam suatu register di - assign ke register simbolis atau virtual. Kemudian kompiler memetakan register simbolik dalam jumlah tak terbatas ke dalam register real yang jumlahnya tetap. Register simbolik yang pemakaiannya tidak bertumpang tindih dapat memakai sebuah register real secara bersama – sama. Apabila, dalam bagian program tertentu, terdapat lebih banyak jumlah kuantitas daripada jumlah register real, maka sebagian kuantitas di - assign ke lokasi - lokasi memori. Instruksi load dan store digunakan kuantitas posisi sementara di dalam register untuk keperluan operasi komputasi.

Inti tugas optimasi adalah untuk menentukan kuantitas yang akan di – assign ke register pada sembarang posisi di dalam program. Teknik yang sangat umum digunakan pada kompiler RISC dikenal sebagai graph coloring, yang merupakan sebuah teknik yang diambil dari ilmu topologi [CHAI82, CHOW86, COUT86, CHOW90].

#### Graph Coloring

Masalah yang ditemukan dalam teknik graph coloring adalah apabila diketahui sebuah graph yang terdiri dari simpul dan lengan, berikan warna - warna ke sejumlah simpul sehingga simpul - simpul yang berdampingan memiliki warna yang berlainan, dan lakukan hal tersebut sedemikian rupa sehingga digunakan warna sesedikit mungkin. Masalah tersebut digunakan pada masalah kompiler seperti berikut. Pertama, program dianalisis untuk membentuk graph interferensi register. Simpul-simpul graph merupakan register simbolik. Apabila dua buah register simbolik ‘hidup’ selama potongan program tertentu yang sama, maka untuk menggambarkan interferensinya kedua register itu digabungkan dengan sebuah lengan. Kemudian dilakukan pewarnaan terhadap graph dengan menggunakan n buah warna, untuk n

adalah jumlah register. Apabila proses ini tidak sepenuhnya berhasil, maka simpul-simpul yang tidak dapat diwarnai harus ditempatkan di dalam memori, dan operasi load dan store harus dilakukan untuk membuat ruang bagi kuantitas-kuantitas yang dipengaruhi apabila diperlukan.




(a) Time sequence of active use of registers (b) Register interference graph

**Gambar Graph Coloring Approach**

Gambar di atas merupakan contoh proses sederhana proses tersebut. Anggap bahwa sebuah program yang memiliki enam buah register simbolik akan dikompilasi menjadi tiga buah register aktual. Gambar 13.4a menjelaskan urutan waktu penggunaan secara aktif masing – masing register simbolik, dan Gambar 13.4b menjelaskan grafik interferensi register. Diberikan juga pewarnaan dengan menggunakan tiga buah warna. Sebuah register simbolik, F, dibiarkan tidak diwarnai dan harus dilakukan dengan menggunakan load dan store.

Secara umum, terdapat untung rugi antara penggunaan register dalam jumlah yang banyak dengan optimisasi register berbasis kompiler. Misalnya, [BRAD91a] melaporkan tentang penelitian yang memodelkan arsitektur RISC dengan fitur yang sama dengan arsitektur Motorola 88000 dan MIPS R2000. Arsitektur-arsitektur di atas memiliki jumlah register yang berbeda yang berkisar antara 16 hingga 128 buah, serta semuanya menggunakan general purpose register dan memisahkan antara register yang menggunakan integer dengan register yang menggunakan floating point. Studi tersebut menunjukkan bahwa walaupun dengan menggunakan optimisasi sederhana, hanya terdapat sedikit keuntungan dalam menggunakan register yang jumlah lebih dari 64 buah. Dengan teknik optimisasi register yang canggih, hanya terjadi peningkatan kinerja yang marginal bila menggunakan register yang jumlahnya lebih dari 32 buah. Terakhir, penelitian tersebut menyatakan bahwa dengan register yang berjumlah sedikit (misalnya, 16 buah), sebuah mesin yang menggunakan organisasi register yang dapat dipakai bersama akan lebih cepat dibandingkan dengan mesin yang menggunakan organisasi pemisahan register. Kesimpulan yang sama dapat diperoleh dari [HUGU91], yang melaporkan tentang penelitian yang sangat

menekankan pada penggunaan register yang berjumlah sedikit, bukannya membandingkan arsitektur – arsitektur yang menggunakan register dalam jumlah banyak dari usaha optimasinya

## 6. Reduced Instruction Set Arsitecture

**Mengapa CISC?**

Dua alasan utama yang menjadi motivasi kecenderungan ini : adanya keinginan untuk menyederhanakan kompiler dan keinginan untuk meningkatkan kinerja. Sebelum kedua alasan tersebut terdapat keinginan untuk beralih ke bahasa tingkat tinggi (HLL) pada sebagian pemrograman, dan arsitek – arsitek yang berkeinginan untuk merancang mesin yang mendukung HLL lebih baik.

Alasan pertama, yaitu penyederhanaan compiler, telah cukup jelas. Tugas pembuat compiler adalah menghasilkan rangkaian instruksi mesin bagi semua pernyataan HLL. Apabila terdapat instruksi mesin yang menyerupai HLL, maka tugas ini akan disederhanakan. Alasan tersebut untuk menggunakan complex instruction set adalah bahwa eksekusi instruksi akan lebih cepat. Alasan pertama, yaitu penyederhanaan kompiler telah cukup jelas. Tugas pembuat kompiler adalah menghasilkan rangkaian instruksi mesin bagi semua pernyataan HLL. Apabila terdapat instruksi mesin yang menyerupai pernyataan HLL, maka tugas ini akan disederhanakan. Alasan tersebut pernah diragukan oleh para peneliti RISC ([HENN82], [RADI83], [PATT82b]). Mereka berdapat bahwa instruksi mesin yang kompleks sering kali sulit digunakan karena kompiler harus menemukan kasus-kasus yang sesuai dengan konsepnya. Pekerjaan mengoptimalkan kode yang dihasilkan untuk meminimalkan ukuran kode, mengurangi hitungan eksekusi instruksi, dan meningkatkan pipelining jauh lebih sulit apabila menggunakan complex instruction set. Penelitian – penelitian sebelumnya menyatakan bahwa sebagian besar instruksi dalam program yang telah dikompilasi merupakan kode yang relatif sederhana.

Alasan penting lainnya adalah harapan bahwa CISC akan menghasilkan program yang lebih kecil dan lebih cepat. Terdapat dua keuntungan yang dapat diperoleh dari program berukuran kecil. Pertama, karena program membutuhkan memori yang lebih sedikit, maka akan dapat menghemat sumber daya dan dengan semakin murahnya harga memori saat ini. Program yang lebih kecil akan meningkatkan kinerja, dan peningkatan kinerja. Yaitu instruksi yang lebih sedikit dapat diartikan sebagai lebih sedikitnya byte - byte instruksi yang harus diambil. Dan kedua, pada lingkungan paging, program yang berukuran lebih kecil akan mengurangi terjadinya page fault.

Umumnya, program CISC yang diekspresikan dalam bahasa mesin simbolik, akan lebih pendek (yaitu, instruksinya lebih sedikit), namun jumlah bit yang menempati tidaklah kecil. Tabel 13.6 menjelaskan hasil yang diperoleh dari tiga penelitian yang membandingkan ukuran program bahasa C yang dikompilasi pada bermacam – macam mesin termasuk RISC I, yang memiliki arsitektur reduced instruction set. Penggunaan CISC di sini hanya menghemat sedikit atau sama sekali tidak menghemat penggunaan memori dibandingkan dengan bila menggunakan RISC.

Telah diketahui bahwa kompiler – kompiler pada mesin CISC cenderung menggunakan instruksi – instruksi yang lebih sederhana, karena itu keringkasan instruksi kompleks jarang sekali memegang peranan. Selain itu, karena pada CISC terdapat instruksi yang lebih banyak, maka diperlukan opcode yang lebih panjang, yang menghasilkan instruksi yang lebih panjang pula. Sedangkan RISC cenderung lebih menekankan pada referensi register dibandingkan pada referensi memori, dan referensi register memerlukan bit yang jumlahnya lebih sedikit.

Dengan demikian, harapan bahwa CISC akan menghasilkan program – program yang berukuran lebih kecil tidak dapat terealisasi. Faktor motivasi kedua untuk menggunakan complex instruction set yaitu karena suatu operasi HLL yang kompleks akan mengeksekusi lebih cepat karena menyerupai sebuah instruksi mesin dibandingkan denganmengeksekusi sejumlah instruksi-instruksi yang lebih primitif. Akan tetapi karena adanya bias ke arah penggunaan instruksi yang lebih sederhana, maka eksekusi yang lebih cepat tersebut sulit terjadi juga. Seluruh unit control harus dibuat lebih kompleks, dan/atau penyimpanan kontol mikroprogram harus dibuat lebih besar, untuk mengakomodasi set instruksi yang lebih kaya. Salah satu faktor tersebut akan meningkatkan waktu eksekusi instruksi – instruksi sederhana.

Pada kenyataannya, beberapa penelitian telah menemukan bahwa percepatan terhadap eksekusi fungsi – fungsi kompleks tidak berkaitan banyak dengan dengan kekuatan instruksi mesin yang kompleks seperti halnya kekuatannya pada penyimpanan kontol berkecepatan tingggi [RADI83]. Akibatnya, penyimpanan kontol berlaku sebagai suatu cache instruksi. Dengan demikian, arsitektur perangkat keras berada dalam posisi mencoba untuk menentukan subroutine atau fungsi mana yang akan sering digunakan dan menga-assign-nya ke penyimpanan control dengan cara mengimplementasikannya di dalam mikrokode.

Dengan demikian dapat disimpulkan bahwa tidaklah jelas bahwa kecenderungan peningkatan pemakaian complex instruction set akan terjadi. Hal ini menyebabkan kegiatan sejumlah kelompok untuk melanjutkan pemikiran lainnya.

**Karakteristik CISC versus RISC**

Setelah terjadi antusiasme untuk menggunakan mesin-mesin RISC, terjadi kejadian – kejadian sepert :

1. Rancangan RISC dapat memperoleh keuntungan dengan mengambil sejumlah feature CISC.
2. Rancangan CISC dapat memperoleh keuntungan dengan mengambil sejumlah feature RISC.

Hasilnya adalah bahwa rancangan RISC terbaru, yang dikenal sebagai PowerPC, tidak lagi "murni" RISC dan rancangan CISC yang terbaru, yang dikenal sebagai Pentium, memiliki beberapa karakteristik RISC.

Ciri-ciri RISC :

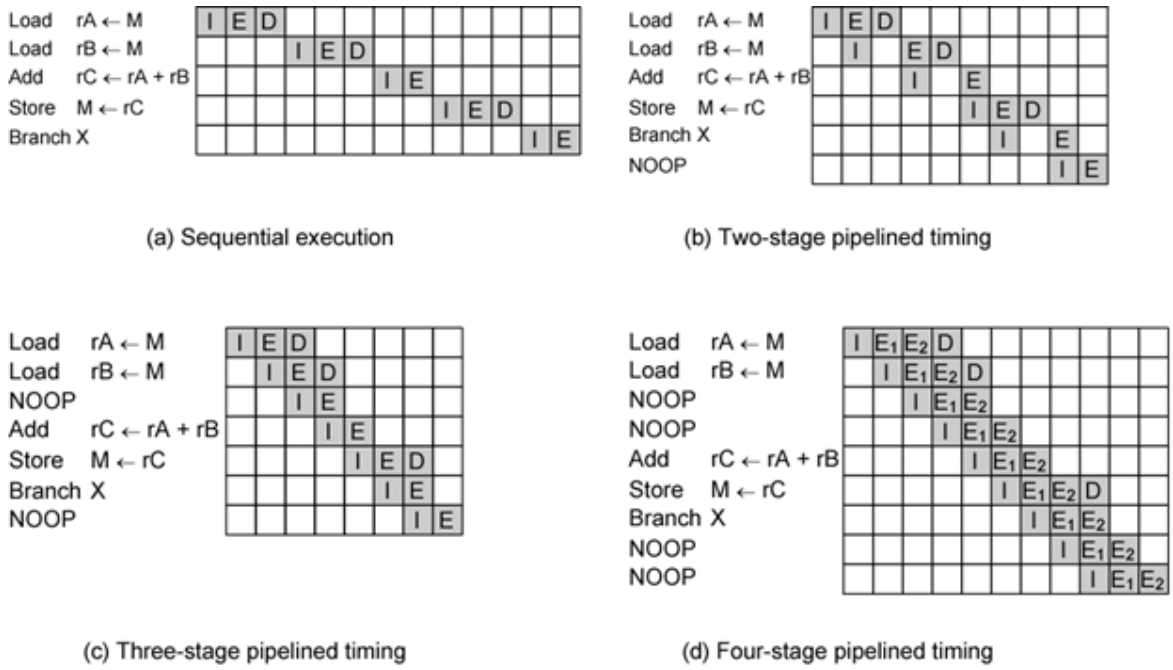
1. Instruksi berukuran tunggal.
2. Ukurannya yang umum adalah 4 byte.
3. Jumlah mode pengalamatan data yang sedikit, biasanya kurang dari lima buah.
4. Tidak terdapat pengalamatan tak langsung yang mengharuskan anda melakukan sebuahakses memori agar memperoleh alamat operand lainnya di dalam memori.
5. Tidak terdapat operasi yang menggabungkan operasi load/store dengan operasiaritmetika. (misalnya, penambahan dari memori, penambahan ke dalam memori).
6. Tidak terdapat lebih dari satu operand beralamat memori per instruksi.
7. Tidak mendukung perataan sembarang, bagi data untuk operasi load/store.
8. Jumlah maksimum pemakaian memori management unit (MMU) bagi suatu alamat dataadalah sebuah instruksi floating
9. Jumlah bit bagi floating point register specifier sama dengan lima atau lebih.

Jumlah bit floating point register specifier sama dengan empat atau lebih

**7. Pipelining RISC**

Terdapat berbagai macam instruksi pada register to register

- Siklus Instruksi memiliki 2 Fase :
  1. I : Instruction Fetch (Pengambilan Instruksi)
  2. E : Execute (Melakukan operasiALU dengan register input dan output
- Operasi Load dan Store memiliki 3 Fase :
  1. I : Instruction Fetch
  2. E : Execute (Menghitung alamat memori)
  3. D : Memory (Operasi register ke memori atau memori ke register




**Gambar Efek dari Pipelining**

Gambar di atas menggambarkan pewaktuan rangkaian instruksi tanpa menggunakan pipelining. Jelas proses ini tidak efisien. Bahkan pipelining yang sangat sederhana pun dapat meningkatkan kinerja.Gambar menjelaskan teknik pipelining dua arah, dengan fase I dan fase E dua buah instruksi yang berlainan dapat dilakukan secara bersamaan.Teknik ini dapat menghasilkan sampai dua kali waktu kecepatan eksekusi teknik serial.Timbul dua masalah yang menghambat tercapainya percepatan maksimum.Pertama, asumsikan bahwa digunakan sebuah memori single – port dan hanya sebuah akses memori yang diizinkan pada setiap fasenya.Hal ini mengharuskan penambahan status menunggu di dalam sejumlah instruksi.Kedua, instruksi pencabangan menginterupsi aliran rangkaian instruksi.Untuk melakukan hal tersebut dengan memakai sirkuit yang sederhana, maka sebuah instruksi NOOP dapat ditambahkan ke dalam aliran instruksi oleh compiler atau assembler.

Pipelining dapat ditingkatkan lebih lanjut dengan cara mengizinkan dua buah akses memori tiap fasenya. Cara ini menghasilkan rangkaian seperti yang digambarkan pada Gambar 13.5.c. Maka sekarang dapat ditumpukkan hingga tiga buah instruksi dan peningkatannya akan mencapai tiga kali. Demikian juga halnya, instruksi pencabangan dapat menghambat percepatan sehingga kecepatan yang terjadi akan kurang dari kecepatan maksimum. Disamping itu, perlu dicatat bahwa dependensi data memiliki efek – efek tertentu.Bila sebuah instruksi membutuhkan sebuah operand yang telah diubah oleh instruksi sebelumnya, maka diperlukan waktu delay.

Pipelining yang telah dibahas sejauh ini akan dapat bekerja dengan baik apabila ketiga fasenya memiliki durasi yang hampir sama. Karena biasanya fase E melibatkan operasi ALU, maka fase ini akan lebih lama. Dalam hal ini, dapat dibagi menjadi dua buah subfase :

- E1 : membaca file register
- E2 : menulis operasi ALU san register

Karena adanya kesederhanaan dan keteraturan set instruksi, rancangan pembuatan fase menjadi tiga atau empat fase cukup mudah dilakukan. Gambar 13.5.d menunjukkan hasil yang diperoleh dengan menggunakan pipeline empat arah.Hingga empat buah instruksi dapat berjalam sekaligus, dan percepatan maksimum yang dihasilkan adalah empat kali. Juga perlu diingat tentang penggunaan NOOP untuk mengakomodasi delay data dan pencabangan.

**Optimalisasi Pipeline**

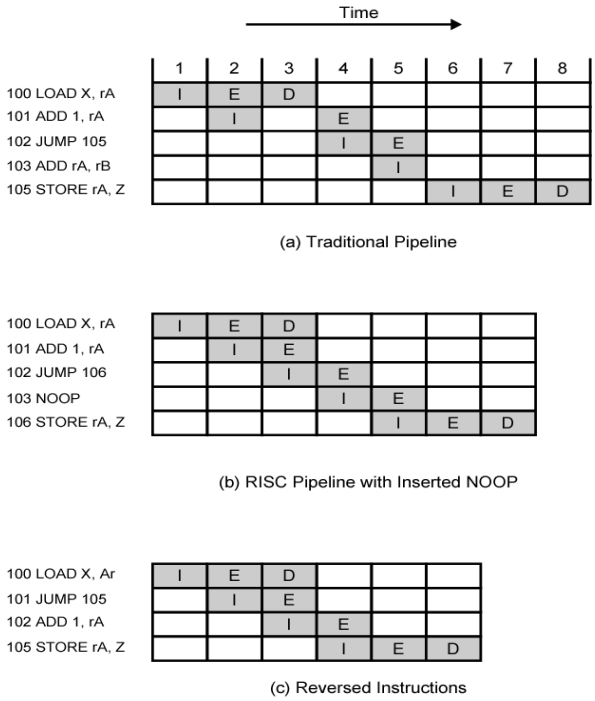
```
do i=2, n-1
                    a[i] = a[i] + a[i-1] * a[i+l]
end do
Becomes
do i=2, n-2, 2
                    a[i] = a[i] + a[i-1] * a[i+i]
                    a[i+l] = a[i+l] + a[i] *
a[i+2]
end do
if (mod(n-2,2) = i) then
                    a[n-1] = a[n-1] + a[n-2] *
a[n]
end if
```

Karena sederhananya instruksi - instruksi RISC, maka pipelining dapat digunakan dengan efisien. Terdapat beberapa macam durasi eksekusi instruksi, dan pipeline dapat digunakan untuk menggambarkan perbedaan tersebut. Namun seperti kita ketahui bahwa dependensi data dan percabangan dapat mengurangi kecepatan eksekusi keseluruhan.

| Address | Normal Branch | | Delayed Branch | | Optimized Delayed Branch | |
|---|---|---|---|---|---|---|
| 100 | LOAD | X, rA | LOAD | X, rA | LOAD | X, rA |
| 101 | ADD | 1, rA | ADD | 1, rA | JUMP | 105 |
| 102 | JUMP | 105 | JUMP | 106 | ADD | 1, rA |
| 103 | ADD | rA, rB | NOOP | | ADD | rA, rB |
| 104 | SUB | rC, rB | ADD | rA, rB | SUB | rC, rB |
| 105 | STORE | rA, Z | SUB | rC, rB | STORE | rA, Z |
| 106 | | | STORE | rA, Z | | |

**Gambar Normal dan Delayed Branch**

Untuk mengkompensasi dependensi - dependensi, telah dibuat teknik pengorganisasian kembali kode. Pertama, ambil instruksi percabangan. *Delayed Branch*, yang merupakan cara untuk meningkatkan efisiensi pipelining, memanfaatkan percabangan yang tidak akan dilakukan sampai dengan instruksi berikutnya dieksekusi. Prosedur yang agak ganjil ini dijelaskan dalam Tabel 13.6. Pada kolom pertama, kita melihat adanya program bahasa mesin instruksi simbolik yang normal. Setelah 102 dieksekusi, instruksi berikutnya yang akan dieksekusi adalah 105. Agar dapat meregulasasikan pipeline, maka setelah pencabangan ini disisipkan sebuah NOOP. Namun peningkatan kinerja akan tercapai apabila instruksi pada 101 dan 102 dipertukarkan. Gambar menjelaskan hasil yang diperoleh. Instruksi JUMP diambil sebelum instruksi ADD. Namun perlu diingat bahwa instruksi ADD diambil sebelum eksekusi instruksi JUMP memiliki kesempatan untuk mengubah program counter. Sehingga semantik orisinil program dapat dijaga.




**Gambar Pemakaian Pencabangan Ber - Delay**

Instruksi pertukaran ini akan berfungsi dengan baik bagi percabangan tak bersyarat, call, dan return. Bagi percabangan bersyarat, prosedur ini tidak dapat diterapkan begitu saja. Apabila syarat percabangan yang diuji dapat diubah oleh instruksi sebelumnya, maka kompiler harus menahan dahulu agar tidak melakukan pertukaran, namun menyisipkan sebuah NOOP.

Taktik yang sama, yang disebut delayed load, dapat digunakan terhadap instruksi LOAD. Pada instruksi LO register tersebut, yaitu posisi yang CPU akan idle sampai pemuatan selesai. Apabila kompiler dapat mengatur kembali instruksi – instruksi sehingga tugas – tugas yang bermanfaat dapat dilakukan pada saat pemuatan berada di dalam pipeline, maka efisiensi akan bertambah.

## REFERENSI

Stalling, W. 2010. Computer Organization and Architecture design dan Performance eighth edition. Prentice Hall

## PROPAGASI

**A. Latihan dan Diskusi (Propagasi vertical dan Horizontal)**

**13.1** What are some typical distinguishing characteristics of RISC organization?

**13.2** Briefly explain the two basic approaches used to minimize register-memory operations on RISC machines.

**13.3** If a circular register buffer is used to handle local variables for nested procedures, describe two approaches for handling global variables.

**13.4** What are some typical characteristics of a RISC instruction set architecture?

**13.5** What is a delayed branch?

**B. Pertanyaan** (Evaluasi mandiri)

**13.1** Considering the call-return pattern in Figure 4.21, how many overflows and underflows (each of which causes a register save/restore) will occur with a window size of

**a.** 5?

**b.** 8?

**c.** 16?

**13.2** In the discussion of Figure 13.2, it was stated that only the first two portions of a window are saved or restored.Why is it not necessary to save the temporary registers?

**13.3** We wish to determine the execution time for a given program using the various pipelining schemes discussed in Section 13.5. Let For the simple sequential scheme (Figure 13.6a), the execution time is stages. Derive formulas for two-stage, three-stage, and four-stage pipelining.

**13.4** Reorganize the code sequence in Figure 13.6d to reduce the number of NOOPs.

**C. QUIZ** -mutiple choice (Evaluasi)

**D. PROYEK** (Eksplorasi entrepreneurship, penerapan topic bahasan pada dunia nyata)