BAB

8

Pemrograman Mikro

8.1 UNIT KENDALI LOGIKA

Unit kendali logika (*CLU* atau *Control logic unit*) mengatur seluruh aktifitas perangkat keras di dalam komputer. CLU menyebabkan suatu instruksi di-fetch dari memori, memberi kode pada instruksi tersebut untuk menentukan operasi yang akan dilaksanakan, menentukan sumber dan tujuan data, dan menyebabkan perpindahan data dan eksekusi operasi yang diperlukan. CLU mengulangi seluruh proses sampai sebuah operasi HALT secara tiba-tiba masuk ke dalam program dan dieksekusi.

Kode instruksi, bersama-sama dengan data, tersimpan di dalam memori. Seperti telah kita ketahui, sebuah instruksi merupakan suatu entitas yang kompleks yang pelaksanaannya tidak dapat diselesaikan dalam sebuah pulsa waktu tunggal. Karena itu, setelah menginterpretasikan kode biner suatu instruksi, CLU menghasilkan serangkaian perintah kendali, yang disebut sebagai **instruksi-mikro** (*microinstruction*), yang menjalankan instruksi tersebut. Kita telah melihat contoh-contoh instruksi sema-

cam ini pada Bab 4 dan 5. Untuk membedakan antara sebuah instruksi dan sebuah instruksi-mikro, seringkali instruksi-instruksi disebut sebagai **instruksi-makro** (*macroinstruction*).

Durasi siklus fetch/eksekusi tergantung pada jenis operasi yang akan dikerjakan, mode pengalamatan yang digunakan dan jumlah operand yang diperlukan. Yang dikerjakan CLU adalah membagi setiap siklus instruksi menjadi serangkaian **keadaan** (*state*). Semua state mempunyai panjang yang sama dan durasi setiap state sama dengan periode clock komputer. Karena itu, suatu state berhubungan dengan unit terkecil dari aktifitas pemrosesan dan menyatakan eksekusi satu atau lebih instruksi-mikro secara bersamaan.

Instruksi-mikro merupakan operasi primitif tingkat rendah yang bertindak secara langsung pada sirkuit logika suatu komputer. Mereka memerinci fungsi-fungsi (sinyal-sinyal) seperti berikut:

1. Membuka/menutup suatu gerbang (*gate*) dari sebuah register ke sebuah bus.
2. Mentransfer data sepanjang sebuah bus.
3. Memberi inisial sinyal-sinyal kendali seperti READ, WRITE, SHIFT, CLEAR dan SET.
4. Mengirimkan sinyal-sinyal waktu.
5. Menunggu sejumlah periode waktu tertentu.
6. Menguji bit-bit tertentu dalam sebuah register.

Ada dua pendekatan pokok bagi perancangan sebuah CLU yaitu: rancangan *hard-wired* (atau *logika random*) dan rancangan *microprogrammed*. Pada pendekatan *hard-wired*, sejumlah gerbang (*gate*), counter dan register saling dihubungkan untuk menghasilkan sinyal-sinyal kendali. Setiap rancangan memerlukan sekelompok piranti logika dan hubungan yang berbeda-beda. Pada pendekatan *microprogrammed*, dibentuk serangkaian instruksi-mikro, disebut sebagai sebuah **program-mikro**, untuk setiap instruksi-mikro dan disimpan dalam sebuah memori kendali (biasanya sebuah ROM) dalam CLU. Kemudian waktu yang diperlukan dan sinyal kendali dihasilkan dengan menjalankan suatu program-mikro untuk masing-masing instruksi-makro.

8.2 KENDALI HARD-WIRED

Sewaktu sebuah instruksi ditempatkan dalam **register instruksi** (*IR* atau *instruction register*), CLU men-decode instruksi itu dan menghasilkan serangkaian

instruksi-mikro. Ambillah contoh suatu komputer yang mempunyai 16 instruksi sehingga setiap instruksi dapat diberi kode dengan sebuah opcode 4-bit yang unik. (Tentu saja, sisa word instruksi berisi informasi pengalamatan yang penting seperti register-register yang terlibat, address-address memori dan offset). Gambar 8-1 menunjukkan suatu kemungkinan implementasi decoder untuk opcode tersebut. **Mnemonic**-nya diperlihatkan berikut ini:

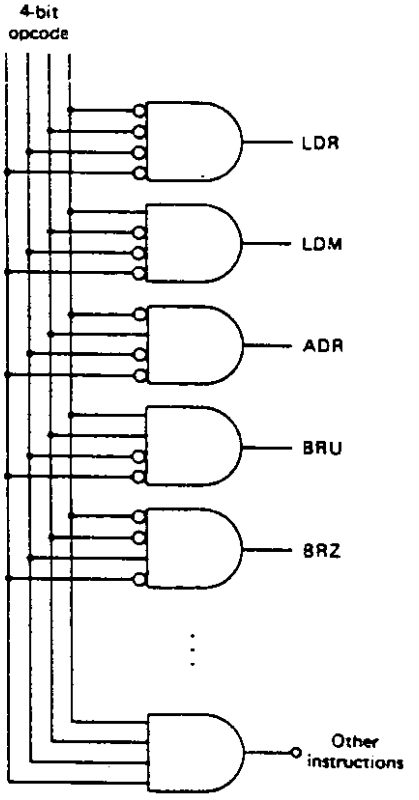
LDR (Load suatu register dari memori)
LDM (Load memori dari suatu register)
ADR (Add ke register)
BRU (Branch/percabangan tidak kondisional)
BRZ (Branch/percabangan pada nol)

Setiap output decoder dapat digunakan untuk menghasilkan serangkaian instruksi-mikro. Instruksi-mikro ini dikendalikan oleh sinyal waktu dan beberapa kondisi status di dalam sistem.

Alamat semua operand yang diperlukan oleh opcode harus dipasok ke *memory address register* (MAR) pada waktu yang tepat. Unit memori itu men-decode address yang dipersembahkan oleh MAR dan mengeluarkan word yang tersimpan dalam lokasi yang dialamati melalui *memory buffer register* (MBR). Jika informasi pengalamatan dalam instruksi tersebut bukan address operand yang sesungguhnya maka CLU harus menerapkan langkah-langkah yang diperlukan untuk pengalamatan tidak langsung, berindeks atau mode pengalamatan lainnya (lihat Bagian 5.3).

Setiap rangkaian instruksi-mikro dijalankan dalam suatu periode waktu yang disebut sebagai suatu **siklus mesin** (*machine cycle*). CLU membagi-bagi siklus mesin menjadi serangkaian keadaan (state), t_0, t_1, ..., sedemikian sehingga durasi tiap-tiap state sama dengan periode clock komputer. Kita dapat menghasilkan sinyal waktu ini dengan menggunakan sebuah counter ring atau, seperti ditunjukkan pada Gambar 8-2(a), sebuah kombinasi counter-decoder. Sirkuit counter-decoder ini menghasilkan suatu rangkaian berurutan atas empat state tersebut yang mengulanginya setiap empat pulsa waktu. Diagram waktu untuk sirkuit ini ditunjukkan pada Gambar 8-2(b).

Untuk mengilustrasikan bagaimana sirkuit counter-decoder pada Gambar 8-2 menghasilkan sinyal waktu yang diperlukan untuk menjalankan sebuah instruksi komputer, perhatikan suatu instruksi ADD pada sebuah komputer pengalamatan tunggal (lihat Bab 5). Untuk mem-fetch dan menjalankan instruksi ini, CLU harus melakukan langkah-langkah berikut ini:



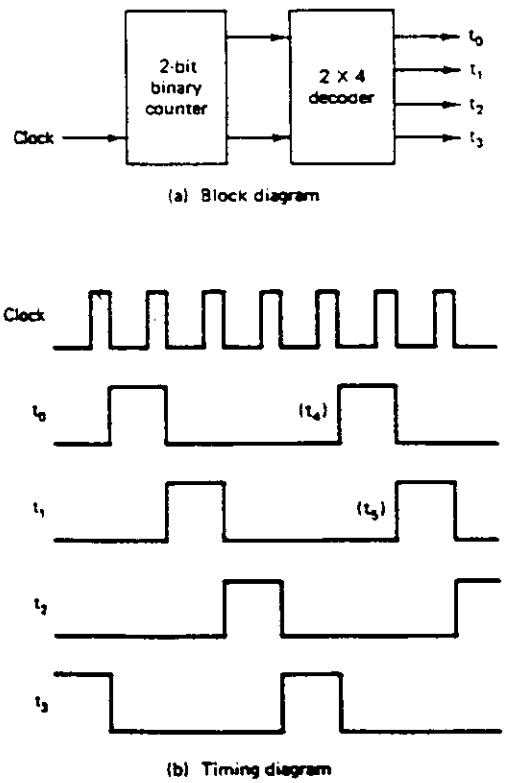
Gambar 8-1 Decoder opcode 4-bit

t_0: MAR ← (PC)	Isi PC dikirim ke MAR.
t_1: MBR ← M[MAR]	Isi lokasi memori yang dialamati diletakkan pada MBR dan PC bertambah 1.
t_2: IR ← (MBR)	Isi MBR dikirim ke IR.
t_3: MAR ← (IR|address operand)	Field address operand instruksi tersebut dikirim dari IR ke MAR.
t_4: MBR ← M[MAR]	Operand diletakkan pada MBR.
t_5: ACC ← (ACC) + (MBR)	Isi MBR dan akumulator ditambahkan dan hasilnya diletakkan pada akumulator

Sinyal waktu t_4 dan t_5 ditunjukkan dalam tanda kurung pada Gambar 8-2(b).

Seperti yang kita lihat, pelaksanaan instruksi ADD terdiri atas serangkaian enam keadaan (pulsa waktu). Dalam menghasilkan rangkaian ini, kita telah mengasumsikan bahwa waktu akses memori sama rata dengan bagian-bagian komputer lainnya. Namun jika memori tersebut lebih lambat maka state t_1 dan t_4 akan mengkonsumsi waktu lebih banyak dan pelaksanaan instruksi itu akan memerlukan lebih dari enam keadaan (*state*).

Pada pokoknya, sebuah CLU yang *hard-wired* bisa sinkron dan bisa pula tidak sinkron. Pada **CLU yang sinkron** (*synchronous CLU*), setiap operasi dikendalikan oleh clock. Frekuensi clock harus sedemekian rupa sehingga waktu antara dua pulsa waktu cukup untuk memungkinkan penyelesaian instruksi-mikro yang



Gambar 8-2 Pembuatan sinyal waktu

paling lambat. Pada **CLU yang tidak sinkron** (*asynchronous CLU*), penyelesaian suatu operasi memicu operasi berikutnya dan karena itu tidak diperlukan clock. Meskipun rancangan CLU yang tidak sinkron lebih kompleks namun ia dapat dibuat agar berfungsi lebih cepat dibandingkan dengan CLU yang sinkron. Namun demikian, dengan pendekatan apapun, implementasi *hard-wired* pada CLU menghasilkan respon yang cepat secara keseluruhan. Walaupun demikian jenis ini telah terdesak oleh kendali *microprogrammed*, untuk alasan-alasan tertentu yang akan dibahas pada bagian berikut.

8.3 KENDALI MICROPROGRAMMED

Istilah *program-mikro* pertama kali diungkapkan oleh M.V Wilkes pada awal tahun 1950-an ketika dia mengajukan suatu pendekatan baru untuk mengendalikan perancangan unit. Ide ini menarik perhatian banyak ahli dan insinyur komputer pada saat itu, walaupun hal itu tampak tidak realistis karena adanya persyaratan untuk memori kendali yang sangat cepat dan relatif tidak mahal. Situasi ini berubah secara dramatis dengan adanya pengumuman keluarga komputer IBM System/360 pada bulan April 1964. Seluruh model terbesar menyertakan memori kontrol yang cepat dan tidak mahal dan merupakan *microprogrammed*. Sejak itu, pemrograman mikro menjadi hal yang umum sejalan dengan peningkatan kecepatan dan penurunan harga memori kontrol.

Suatu unit logika kendali *hard-wired* memerlukan perancangan ulang perangkat keras secara ekstensif jika serangkaian instruksi harus dikembangkan atau jika fungsi sebuah instruksi harus diubah. Sebaliknya, dalam **CLU microprogrammed**, serangkaian instruksi mikro (program-mikro) yang berhubungan dengan masing-masing instruksi dalam kelompok instruksi tersimpan di dalam memori hanya-baca (*ROM* atau *read-only memory*) yang disebut sebagai **memori kendali**. Oleh karena itu, arti sebuah instruksi dapat diubah dengan mengubah program-mikro yang bersesuaian dengan instruksi tersebut dan kelompok instruksi dapat dikembangkan hanya dengan menyertakan ROM tambahan yang berisi program-mikro yang bersesuaian. Sebagai hasilnya, perubahan perangkat keras yang diperlukan dalam CLU dapat diusahakan sampai batas minimal.

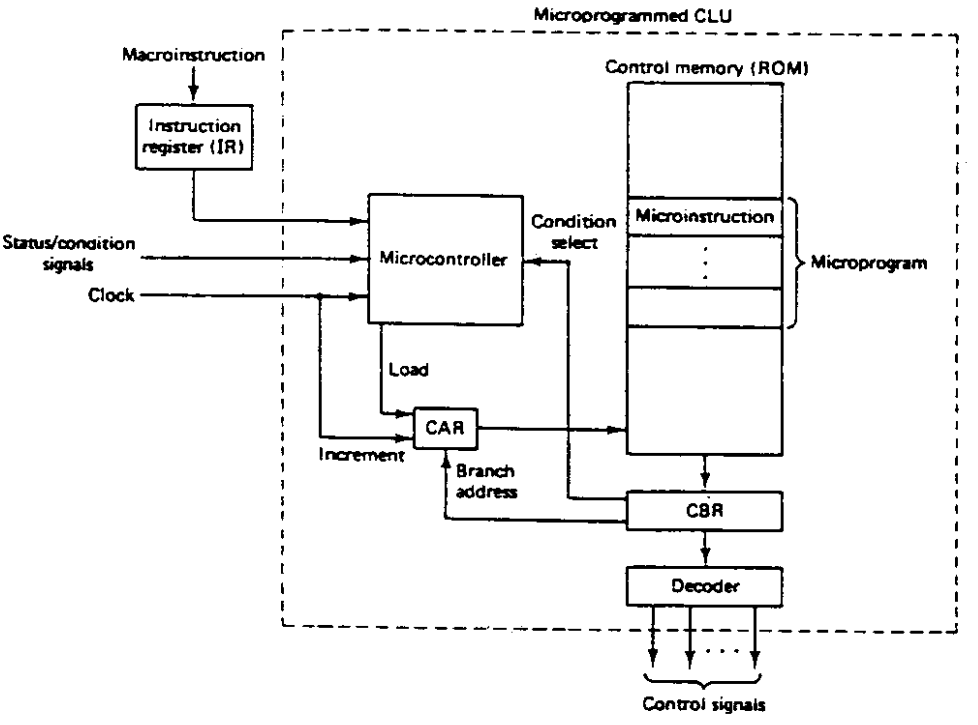
Telah disebutkan sebelumnya, teknik pemrograman mikro pertama kali diperkenalkan pada awal tahun 1950-an namun tidak layak secara ekonomis sampai suatu saat tersedia ROM yang cepat dan harganya wajar. Tingkat di mana program-mikro dapat diubah oleh pemakai, bervariasi dari suatu komputer ke kom-

puter lainnya. Beberapa komputer tidak memungkinkan adanya perubahan, sedangkan yang lainnya memungkinkan sebagian perubahan dan yang lainnya lagi tidak memiliki kelompok instruksi yang tetap namun sebagai pengganti memungkinkan pemakai untuk melakukan pemrograman-mikro atas kelompok instruksi yang dibentuk sesuai dengan peruntukkan aplikasi komputer tersebut. Konsekuensinya, suatu CLU *microprogrammed* disebut sebagai *microprogrammable* (dapat diprogram secara mikro) jika memori kontrol dapat dimodifikasi oleh pemakai untuk menghasilkan instruksi-makro yang dibentuk sesuai keinginannya. Jika tidak, kelompok instruksi tersebut tetap, mirip dengan kelompok instruksi dalam CLU *hard-wired.*

Organisasi CLU Microprogrammed

Organisasi pokok sebuah CLU *microprogrammed* ditunjukkan pada Gambar 8-3. Instruksi-makro disimpan dalam memori utama dan diakses melalui *memory address register* (MAR) dan *memory buffer register* (MBR). Instruksi di-fetch ke dalam register instruksi (*IR* atau *instruction register*) dan **pengendali-mikro** (*microcontroller* atau *sequencer*) menjalankan program-mikro yang bersesuaian. Address awal program-mikro di-load ke dalam **register address kendali** (*CAR* atau *control address register*) dan kemudian memori kontrol mentransfer instruksi-mikro pertama ke dalam **register buffer kendali** (*CBR* atau *control buffer register*). Dengan mem-fetch sebuah instruksi-mikro dari memori kontrol berarti kita menyatakan sebuah **siklus-mikro**, yaitu waktu di mana instruksi-mikro di-decode untuk menghasilkan sinyal kendali yang diperlukan untuk menjalankannya. CAR secara normal bertambah 1 pada tiap-tiap pulsa waktu sehingga ia dapat mengalamati instruksi-mikro berikutnya secara berurutan. Namun, perhatikan bahwa rangkaian tersebut dapat diubah oleh kondisi-kondisi yang terjadi di dalam atau di luar CLU, yang mungkin menyebabkan pengendali-mikro (*microcontroller*) meningkatkan CAR dengan lebih dari 1. Jika ada operand yang diperlukan untuk suatu instruksi tertentu maka informasi address dalam IR di-decode untuk melengkapi lokasi operand.

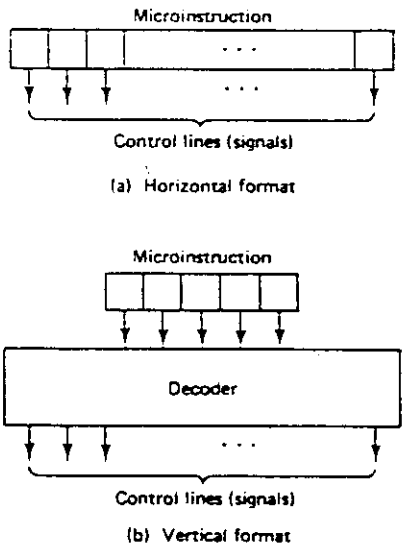
Seperti kita lihat, ada kemiripan fungsional antara pasangan register MAR dan MBR dan pasangan CAR dan CBR. Pasangan pertama digunakan untuk mengakses instruksi-makro, sedangkan pasangan yang terakhir digunakan untuk mengakses instruksi-mikro. Perhatikan juga bahwa fungsi sirkuit perangkai instruksi-mikro mirip dengan fungsi sirkuit perangkai instruksi-makro, yang menggunakan sebuah program counter (PC).



Gambar 8-3 Organisasi sebuah CLU *microprogrammed*

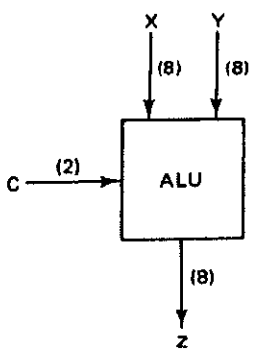
Format Instruksi-mikro

Pada dasarnya ada dua jenis format instruksi-mikro: horisontal dan vertikal. Pada format instruksi-mikro **horisontal**, satu bit diberikan untuk setiap sinyal logika yang dapat dihasilkan oleh instruksi-mikro, seperti ditunjukkan pada Gambar 8-4(a). Dengan demikian, jika dibutuhkan sejumlah K sinyal kendali yang berbeda maka dibutuhkan instruksi-mikro dengan word sepanjang K bit. Untuk menghasilkan suatu sinyal tertentu, bit yang bersesuaian dalam instruksi-mikro di-set menjadi 1; kehadiran suatu sinyal kendali diindikasikan dengan menempatkan sebuah nol pada posisi bit yang semestinya. Pendekatan ini mempunyai keuntungan bahwa kita dapat menghasilkan sebanyak mungkin sinyal kendali yang diperlukan secara beruntun, yang memungkinkan suatu operasi yang sangat cepat.



Gambar 8-4 Format instruksi-mikro

Namun demikian, kebanyakan operasi-mikro adalah mutual ekslusif dan tidak pernah dipanggil secara bersamaan. Karena itu, kita dapat membagi mereka ke dalam kelompok-kelompok dan menggunakan sejumlah bit (field) untuk memberi kode sekumpulan instruksi-mikro yang mutual eksklusif. Kemudian digunakan suatu decoder untuk memilih operasi mikro tertentu yang akan dipanggil. Jika



Gambar 8-5 Sebuah ALU

terbawa ke dalam ekstrem (hanya satu field) maka proses coding dan decoding menghasilkan suatu format instruksi-mikro **vertikal**, dimana hanya satu operasi-mikro yang dipanggil pada suatu waktu, seperti ditunjukkan pada Gambar 8-4(b). Karena itu format sebuah instruksi-mikro vertikal menyerupai format sebuah instruksi-makro dan terdiri atas suatu kode operasi tunggal, disebut sebagai **op-code mikro**, satu operand atau lebih, dan beberapa field lain (untuk percabangan kondisional, misalnya).

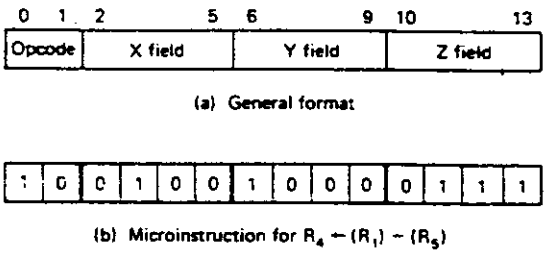
Untuk menggambarkan perbedaan antara format instruksi-mikro horisontal dan vertikal, perhatikan Gambar 8-5, yang memperlihatkan sebuah unit aritmatika dan logika (ALU) dengan dua input paralel 8-bit X dan Y dan sebuah output paralel tunggal 8-bit Z. Dua bit input yang diberi nama C digunakan untuk mengendalikan operasi ALU. Data input dapat diperoleh dari salah satu dari enam register 8-bit R_0 hingga R_5 (tidak diperlihatkan Gambar 8-5) dan data output dapat ditulis ke dalam salah satu register ini. Kita berasumsi bahwa input dan output ALU tetap tidak berubah sampai data baru tersimpan ke dalamnya. Untuk mengakses register tersebut, masing-masing memiliki sebuah address 4-bit yang di-encode, misalnya, dengan kode di atas 3. Dengan demikian kita meng-encode R_0 dengan pola bit 0011, R_1 dengan 0100, dan seterusnya, meng-encode R_5 dengan 1000.

Agar lebih sederhana, mari kita asumsikan bahwa ALU hanya dapat menjalankan tiga fungsi yang berbeda: tidak ada operasi (NOP atau no operation), tambahkan X dan Y ($X + Y$), dan kurangkan Y dari X ($X - Y$). Untuk memilih salah satu operasi ini, ALU menggunakan dua baris kendali (C) yang diperlihatkan pada Gambar 8-5. Asumsikan bahwa NOP di-encode sebagai 00, ($X + Y$) sebagai 01, dan ($X - Y$) sebagai 10.

Rangkaian operasi-mikro yang berhubungan dengan ALU tersebut terdiri atas langkah-langkah berikut ini:

$X \leftarrow$ (Register)	Transfer isi sebuah register ke input X pada ALU.
$Y \leftarrow$ (Register)	Transfer isi sebuah register ke input Y pada ALU.
$Z \leftarrow f(X,Y)$	Memilih sebuah operasi ALU.
Register $\leftarrow$ (Z)	Menuliskan Z ke sebuah register.

Jika kita menggunakan suatu format instruksi-mikro horisontal untuk menjalankan operasi-mikro ini maka instruksi-mikro terdiri atas 14 bit, seperti ditunjukkan pada Gambar 8-6(a). Dua bit dicadangkan untuk opcode, yang menunjukkan operasi ALU yang dipilih. Field X 4-bit yang menunjukkan address register tersebut memasok data ke input X, sedangkan field Y 4-bit menunjukkan register yang



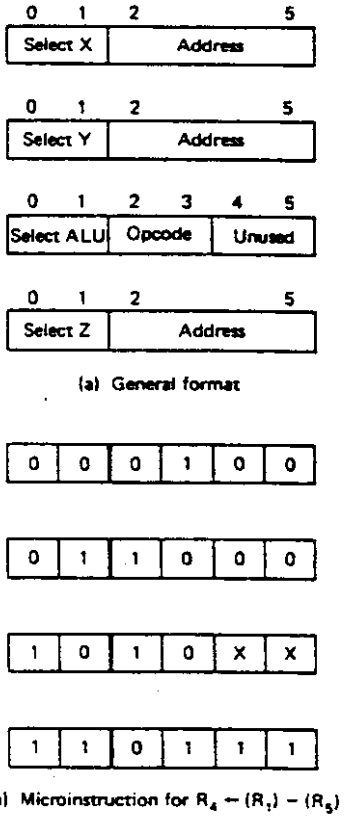
Gambar 8-6 Format horisontal untuk ALU pada Gambar 8-5

berhubungan dengan input *Y* dan field *Z* 4-bit menunjukkan register tempat penulisan hasil operasi ALU. Misalnya, Gambar 8-6(b) memperlihatkan instruksi-mikro horisontal yang telah dikodekan, yang diperlukan untuk menjalankan $R_4 \leftarrow (R_1) - (R_5)$.

Sebaliknya, suatu format vertikal memerlukan empat instruksi-mikro untuk menjalankan operasi-mikro, seperti ditunjukkan pada Gambar 8-7(a). Dua bit dicadangkan untuk meng-encode masing-masing keempat langkah tersebut dalam rangkaian operasi-mikro yang berhubungan dengan ALU. Seperti kita lihat, format vertikal ini memerlukan 24 bit memori kendali. Hanya 2 bit yang tidak digunakan, yaitu dalam instruksi-mikro berhubungan dengan langkah "pilih sebuah operasi ALU". Jika kita meng-encode empat operasi-mikro "pilih *X*", "pilih *Y*", "pilih ALU", " pilih *Z*", masing-masing sebagai 00, 01, 10 dan 11, Gambar 8-7(b) memperlihatkan pemberian kode yang diperlukan untuk menjalankan $R_4 \leftarrow (R_1) - (R_5)$.

Dalam contoh ini terlihat bahwa format horisontal menggunakan memori kendali lebih efisien daripada format vertikal. Hal ini disebabkan karena, dalam contoh tersebut, seluruh field tersebut digunakan dalam format horisontal. Tetapi, hal ini tidak terjadi dalam setiap kasus program-mikro. Sebagai contoh, perhatikan rangkaian operasi berikut ini:

$$R_4 \leftarrow (R_1) - (R_5)$$
$$R_0 \leftarrow (R_4)$$
$$R_3 \leftarrow (R_1) + (R_5)$$
$$R_2 \leftarrow (R_3)$$

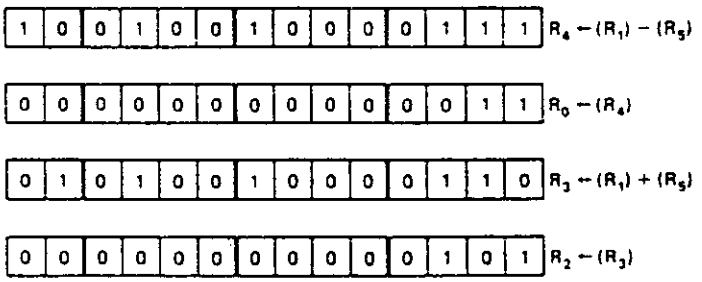


Gambar 8-7 Format vertikal untuk ALU pada Gambar 8-5

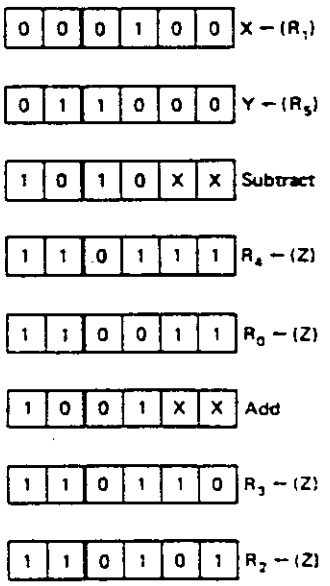
Perhatikan bahwa karena (R_1) - (R_5) ditransfer ke R_4 dari Z, $R_0 \leftarrow (R_4)$ dengan demikian berarti bahwa Z juga disalin ke dalam R_0. Suatu situasi yang serupa terjadi pula pada operasi $R_2 \leftarrow (R_3)$.

Pada Gambar 8-8(a) kita memperlihatkan keempat instruksi-mikro horisontal yang diperlukan untuk menjalankan rangkaian tersebut. Program mikro itu menggunakan 56 bit (4 x 14) memori kendali tetapi seperti sia-sia karena ketiga field dalam instruksi-mikro ke-dua dan ke-empat tidak digunakan. Sebaliknya, delapan instruksi-mikro vertikal yang diperlihatkan pada Gambar 8-8(b) hanya memerlukan 48 bit (8 x 6) memori kendali dengan hanya 4 bit yang sia-sia.

Karena format dan operasi instruksi-mikro vertikal menyerupai instruksi-makro maka lebih mudah bagi kita untuk menuliskan program-mikro vertikal daripada pasangan horisontal-nya.



(a) Horizontal microinstruction sequence



(b) Vertical microinstruction sequence

Gambar 8-8 Contoh program-mikro

Pendekatan vertikal memerlukan panjang word yang relatif pendek tetapi implementasinya memerlukan sirkuit yang lebih kompleks dan tidak mengambil seluruh keuntungan paralelisme dalam arsitektur-mikro. Kebanyakan prosesor memakai kombinasi dari kedua pendekatan itu. Format instruksi-mikro dapat sangat bervariasi dari rancangan yang satu ke lainnya dan tergantung pada faktor-faktor

seperti kelompok operasi mikro yang dipilih, pertukaran (*trade-off*) antara panjang word instruksi-mikro dan kecepatan operasi, dan tingkat kompleksitas sirkuit decoding-instruksimikro yang diterapkan.

Perangkaian Instruksi-mikro

Meskipun terkadang kita cukup hanya mem-fetch instruksi-mikro berikutnya secara berurutan, kita memerlukan beberapa mekanisme yang memungkinkan lompatan kondisional (*conditional jump*) dalam program-mikro yang memungkinkannya untuk membuat suatu keputusan. Lompatan kondisional memungkinkan instruksi-mikro mempunyai dua pendahulu (*successor*) dan meningkatkan kecepatan eksekusinya, ketimbang men-setup beberapa kondisi dalam satu instruksi-mikro dan kemudian mengujinya pada instruksi-mikro berikutnya. Untuk mencapai hal tersebut, disiapkan dua field untuk setiap instruksi-mikro yaitu: sebuah field ADDR, yang berisi address pendahulu (*successor*) yang potensial bagi instruksi-mikro saat ini, dan sebuah field COND, yang menentukan apakah instruksi-mikro berikutnya di-fetch dari (CAR) + 1 atau dari ADDR.

Pilihan dari instruksi-mikro berikutnya ditentukan oleh sirkuit logika perangkai-mikro (*microsequencing logic circuit*). Output sirkuit kendali ini mengendalikan suatu multiplexer, yang mengarahkan dari (CAR) + 1 atau ADDR ke CAR karena address instruksi-mikro berikutnya berada dalam CAR. Karena hanya ada empat pilihan mengenai instruksi-mikro berikutnya maka lebar field COND dapat sebesar 2 bit dan keempat pilihan tersebut dapat diindikasikan dengan mengatur COND sebagai berikut:

COND = 00	Jangan lompat (jump); instruksi-mikro berikutnya diambil dari (CAR) + 1.
COND = 01	Lompat ke ADDR jika C_1.
COND = 10	Lompat ke ADDR jika C_2.
COND = 11	Lompat ke ADDR tanpa kondisional.

Di sini, C_1 dan C_2 merupakan dua bit status yang mewakili kondisi untuk suatu lompatan (jump). Sirkuit logika perangkai-mikro mengkombinasikan C_1 dan C_2 dan dua bit COND untuk menghasilkan suatu output. Dengan demikian sinyal kendali adalah 1 (rute ADDR ke CAR) jika COND = 01 dan C_1 = 1, jika COND = 10 dan C_2 = 1, atau jika COND = 11. Jika tidak maka sinyal kendali bernilai 0 dan instruksi-mikro berikutnya dalam rangkaian di-fetch.

8.4 EMULASI

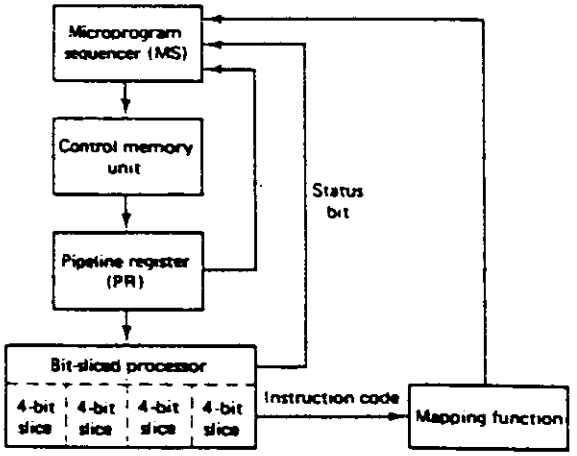
Salah satu fungsi pokok dari kendali *microprogrammed* adalah untuk menyediakan suatu alat untuk pengendalian komputer yang relatif sederhana, fleksibel dan tidak mahal. Fleksibilitas kendali *microprogrammed* dalam menangani sumber daya suatu komputer memungkinkan kita untuk menerapkan kelas instruksi yang berbeda-beda. Dalam suatu komputer dengan sebuah kumpulan instruksi yang tetap, memori kendali dapat berupa sebuah ROM. Namun demikian, jika kita menggunakan memori baca/tulis (atau ROM yang dapat diprogram) untuk memori kendali, kita dapat mengubah kumpulan instruksi tersebut dengan menulis program-mikro yang baru. Hal ini menuntun kita kepada konsep emulasi.

Dalam **emulasi** (*emulation*), suatu komputer diprogram secara mikro (*microprogrammed*) untuk mempunyai kumpulan instruksi yang benar-benar sama dengan komputer lainnya dan dapat mempunyai tingkah laku yang sama pula. Oleh karena itu, program-program yang ditulis untuk komputer yang beremulasi akan berjalan pada komputer *microprogrammed*. Jika kita mempunyai komputer dengan kumpulan instruksi C_1, sekarang kita dapat menambahkan kumpulan instruksi C_2 dari suatu komputer yang benar-benar berbeda kepada C_1. Dengan demikian program yang ditulis dalam bahasa mesin C_2 dapat berjalan pada C_1 dan kita katakan bahwa C_1 *mengemulasi* C_2.

Emulasi memungkinkan kita untuk mengganti peralatan yang lama dengan mesin-mesin yang lebih *up-to-date* tanpa mengharuskan pemakai untuk menulis kembali keseluruhan perangkat lunak mereka. Jika komputer pengganti mengemulasi komputer yang asli secara keseluruhan maka tidak perlu ada perubahan perangkat lunak untuk menjalankan program-program yang ada. Hal ini menjadi pertimbangan yang penting bagi para pemakai karena penulisan kembali perangkat lunak biasanya merupakan pekerjaan yang menghabiskan biaya dan waktu. Dengan demikian, emulasi ini memungkinkan transisi ke sistem yang baru dengan kerusakan yang minimal. Emulasi dapat diterapkan dengan mudah jika mesin-mesin tersebut mempunyai arsitektur yang sama. Namun, banyak juga yang berhasil melakukan emulasi komputer dengan menggunakan mesin-mesin yang arsitekturnya benar-benar berbeda.

8.5 IRISAN BIT

Sistem yang dapat diprogram secara mikro (*microprogrammable*) dapat dibuat dengan menggunakan sirkuit terpadu yang khusus (*IC* atau *integrated circuit*)



Gambar 8-9 Unit pengolah pusat irisan-bit

yang disebut sebagai prosesor **irisan-bit** (*bit-sliced*), untuk merancang perangkat keras perangkai-instruksimikro dan menggunakannya sehubungan dengan beragamnya konfigurasi path data dan kumpulan instruksi. Dengan demikian kita dapat menentukan pembangunan blok yang dengan mudah dapat disusun ke dalam suatu komputer untuk memenuhi persyaratan aplikasi tertentu. Kemampuan dan kumpulan instruksi dari komputer semacam ini ditentukan oleh pembangunan blok tertentu yang dipilih, cara mereka saling dihubungkan dan oleh program-mikro.

Fleksibilitas yang terdapat dalam pendekatan ini dapat ditingkatkan jika blok-blok tersebut dapat digunakan untuk membangun prosesor dengan panjang word yang arbitrer. Hal ini merupakan dasar dari ide irisan bit. **Irisan bit** (*bit slice*) merupakan irisan yang melalui data path prosesor tersebut. Irisan itu hanya terdiri atas beberapa bit (umumnya 2 atau 8), yang berisi semua sirkuit logika yang diperlukan untuk menyiapkan fungsi-fungsi ALU, transfer register dan fungsi-fungsi kendali. Pola-pola hubungan untuk sirkuit pada irisan tersebut dihasilkan di bawah kendali program-mikro, sementara sinyal kendali yang dibutuhkan disediakan oleh pembangunan blok lainnya.

Sebagai contoh, kita pakai unit prosesor IC irisan 4-bit yang berisi register, ALU dan penggeser untuk memanipulasi data 4-bit. Jika kita merangkai kedua IC tersebut, kita dapat membuat sebuah unit prosesor 8-bit. Dengan merangkai keempat irisan semacam itu, kita dapat membuat sebuah unit pengolah pusat

16-bit seperti diperlihatkan pada Gambar 8-9. Ide pengirisan bit ini akan dibahas lebih lanjut pada Bagian 11.8.

8.6 PERALATAN PENDUKUNG BAGI PROGRAM-MIKRO

Mari kita perhatikan beberapa peralatan yang paling sering dibutuhkan oleh para perancang komputer untuk mendukung konsep pemrograman-mikro. Peralatan-peralatan yang akan kita bahas terdiri atas empat kategori yaitu: assembler-mikro, formatter, sistem pengembangan dan instrumentasi, dan simulator perangkat keras.

Assembler-mikro

Assembler-mikro merupakan program-program perangkat lunak yang memungkinkan para perancang untuk meng-encode suatu program-mikro dalam suatu bahasa simbolik (menggunakan mnemonics) dan menterjemahkan representasi ini kedalam representasi absolut untuk di-load ke dalam memori kendali. Keuntungan assembler-mikro adalah:

1. Meningkatkan produktivitas dalam penulisan program-mikro. Assembler-mikro memungkinkan para programmer untuk mengacuhkan ribuan rincian yang berfokus pada representasi bit tertentu dari instruksi-mikro dan kebutuhan untuk me-*lay-out* program-mikro dalam memori kendali dan menyusun address cabang absolut.
2. Tingkat independensi terhadap struktur perangkat keras. Jika program-mikro di-encode secara simbolik, seseorang dapat mengubah assembler-mikro dan menyusun kembali program mikro simbolik tersebut.
3. Kemudahan untuk berubah. Jika programer harus meng-encode nilai address cabang biner dalam instruksi-mikro, penyisipan instruksi-mikro yang baru dapat mengacaukan keseluruhan program-mikro dan akan memaksa programer untuk mengubah seluruh address percabangan. Sebaliknya, pada suatu bahasa simbolik, address percabangan diwakili oleh label simbol yang berhubungan dengan instruksi-mikro. Dengan demikian programer dapat dengan mudah menyisipkan instruksi-mikro yang baru dan menyusun kembali program-mikro, yang menyebabkan assembler-mikro mengatur kembali seluruh alamatnya.

4. Pengujian kesalahan. Assembler-mikro dapat mendeteksi jenis-jenis kesalahan tertentu dalam program-mikro, seperti percabangan ke instruksi-mikro yang tidak ada dan penggunaan nilai operasi-mikro yang mutual eksklusif atau kontradiktif.
5. Informasi referensi silang. Hampir semua assembler-mikro menghasilkan suatu daftar yang memperlihatkan semua instruksi-mikro yang mereferensikan nilai-nilai simbolik tertentu. Informasi ini sangat bernilai selama proses debugging dan siklus modifikasi program mikro tersebut.
6. Peningkatan kemampuan dapat terbaca (*readability*). Suatu representasi simbolik (lawan dari bentuk kode-biner) dari sebuah program-mikro lebih mudah untuk dibaca dan dimengerti.

Assembler-mikro terdiri atas dua kategori. Yang pertama terdiri atas assembler-mikro umum atau sesuai definisi (*definition-driven*). Biasanya assembler-mikro semacam ini mempunyai dua input: (1) sebuah definisi format instruksi-mikro dan interpretasi nilai-nilai simbolik dan (2) program-mikro simbolik itu sendiri. Kategori assembler-mikro yang kedua terdiri atas program-program yang dikembangkan untuk bahasa assembly-mikro tertentu dan rancangan instruksi-mikro tertentu. Biasanya, assembler-mikro semacam itu dikembangkan untuk organisasi komputer tertentu, baik yang berasal dari (scratch) atau dengan menggunakan peralatan perangkat lunak yang tersedia untuk pengembangan kompiler atau assembler.

Formatter

Formatter adalah program-program yang memberi fasilitas bagi pemrograman PROM (lihat Bagian 3.8) yang digunakan untu mengimplementasikan memori kendali. Pada dasarnya, suatu formatter mengambil program-mikro yang harus ditulis ke dalam memori kendali dan mengirisnya sedemikian rupa agar dapat dimuat ke dalam berbagai macam chip PROM yang menyatakan memori kendali.

Sistem Pengembangan

Kategori umum ini terdiri atas peralatan perangkat keras dan perangkat lunak untuk mendukung perancangan sistem *microprogrammed*. Peralatan pengembangan ini memungkinkan programer untuk menyimpan program-mikro dan menguji data pada file disk, meng-editnya dari terminal dan mensimulasikan memori

kendali. Sistem pengembangan juga menyediakan dukungan debugging dan layanan emulasi untuk sistem yang sedang dikembangkan.

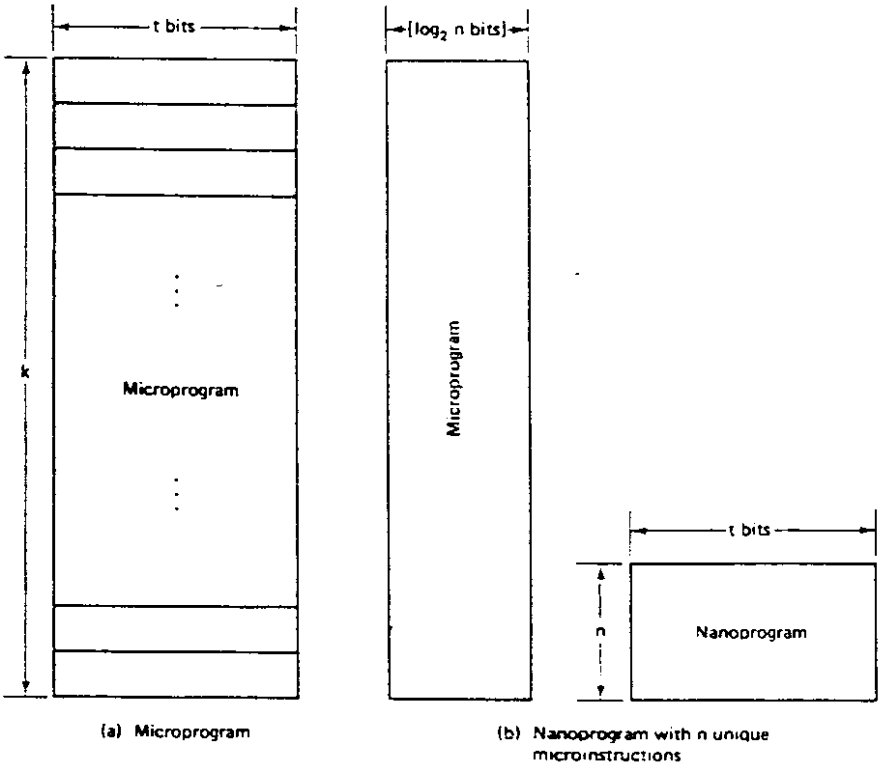
Simulator Perangkat Keras

Simulator perangkat keras (*hardware simulator*) merupakan program-program yang mensimulasikan rincian arus data di dalam perangkat keras yang sedang dirancang. Simulator ini memungkinkan perancang untuk mengembangkan, mendebug dan menguji logika program-mikro bersamaan dengan pengembangan sistem perangkat keras.

8.7 BIAYA DAN KEUNTUNGAN PEMROGRAMAN-MIKRO

Kendali *microprogrammed* menawarkan suatu pendekatan yang lebih terstruktur untuk merancang unit kendali logika (CLU) dibandingkan dengan kendali *hard-wired*. Rancangan *microprogrammed* relatif mudah diubah-ubah dan dibetulkan, menawarkan kemampuan diagnostik yang lebih baik dan lebih dapat diandalkan daripada rancangan *hard-wired*. Karena waktu akses memori kendali ROM menentukan kecepatan operasi CLU maka kendali *microprogrammed* mungkin menghasilkan CLU yang lebih lambat dibandingkan dengan kendali *hard-wired*. Alasannya adalah bahwa waktu yang diperlukan untuk menjalankan suatu instruksi-mikro juga harus mencakup waktu akses ROM. Sebaliknya, suatu keterlambatan dalam CLU *hard-wired* hanya mungkin disebabkan oleh keterlambatan waktu penyebaran melalui perangkat keras, yang relatif sangat kecil. Bagaimanapun juga, ilmu ekonomi kelihatannya lebih menyukai kendali *hard-wired* hanya jika sistem itu tidak terlalu kompleks dan hanya memerlukan beberapa operasi kendali.

Utilisasi memori utama dalam komputer *microprogrammed* biasanya lebih baik karena perangkat lunak yang seharusnya menggunakan ruang memori utama justru ditempatkan pada memori kendali. Pengembangan ROM yang lebih lanjut (dalam kaitannya dengan harga dan waktu akses) secara lebih jauh justru menguatkan posisi dominan pemrograman-mikro. Salah satu ide dalam pengarahan ini adalah untuk menyertakan unit memori ketiga, yang disebut sebagai **memori-nano** (*nanomemory*), sebagai tambahan bagi memori utama dan memori kendali. Dalam mengerjakan hal ini, mungkin terjadi pertukaran (*trade-off*) yang menarik antara pemrograman-mikro horisontal dan vertikal.



Gambar 8-10 Konsep pemrograman-nano

Penyertaan memori-nano sangat tepat jika banyak instruksi-mikro yang terjadi berkali-kali. Perhatikan Gambar 8-10(a) yang memperlihatkan suatu program-mikro yang berisi sejumlah k instruksi-mikro t-bit. Untuk menyimpan program-mikro ini, kita memerlukan sejumlah total kt bit memori kendali. Anggaplah bahwa studi program-mikro menyatakan bahwa hanya sejumlah $n << k$ instruksi-mikro yang berbeda yang benar-benar digunakan. Dalam kasus ini, kita dapat menyimpan instruksi-mikro ini dalam memori-nano khusus n-word, t-bit. Dengan kata lain, setiap instruksi-mikro dalam program asli sekarang dapat digantikan oleh address word memori-nano yang berisi instruksi-mikro itu. Dengan cara ini, memori kendali hanya memerlukan lebar $\log_2 n$ bit, seperti diperlihatkan pada Gambar 8-10(b), karena hanya terdapat n word dalam memori-nano.

Untuk menilai penghematan ruang memori kendali yang dihasilkan oleh pendekatan ini, anggaplah bahwa program-mikro asli tersebut [Gambar 8-10(a)] memerlukan suatu memori kendali sebesar 16.384 x 128 bit namun hanya terjadi sejumlah 256 instruksi-mikro yang berbeda. Dengan demikian kita dapat menggunakan sebuah memori-nano sebesar 256 x 128 bit untuk menyimpan semua instruksi-mikro yang diperlukan, sementara memori kendali dapat dikurangi kapasitasnya menjadi 16.384 x 8 bit. Dengan demikian kita menghemat (16.384 x 128) - (16.384 X 8) - (256 x 128) = 1.933.312 bit. Perhatikan bahwa suatu komputer dengan memori kendali *dua-tingkatan* semacam itu akan bekerja lebih lambat dibandingkan komputer yang tidak menggunakannya karena siklus fetch-mikro memerlukan dua referensi memori: satu ke memori kendali dan satu lagi ke memori-nano. Namun, unjuk kerja dapat ditingkatkan dengan menggunakan teknik-teknik tambahan seperti pipelining dan multiway branching.

SOAL

1. Berikan definisi istilah-istilah berikut ini:
   (a) kendali *hard-wired* **(c)** instruksi-makro
   (b) instruksi-mikro **(d)** program-mikro
2. Tuliskan program-mikro untuk routine pengurangan.
3. Diskusikan perbedaan antara instruksi-mikro vertikal dan horisontal.
4. Diskusikan kondisi-kondisi yang lebih memungkinkan penggunaan kendali *hard-wired* dibandingkan dengan kendali *microprogrammed.*
5. Kemampuan *address-sequencing* apa yang diperlukan dalam suatu memori kendali?
6. Ilustrasikan dan diskusikan logika percabangan kondisional.
7. Tunjukkan sebuah diagram blok untuk mentransformasi sebuah instruksi-mikro ke dalam address memori kendali dengan menggunakan suatu memori pemetaan (*mapping memory*) (ROM atau PLA).
8. Diskusikan operasi-mikro yang diperlukan untuk suatu routine siklus fetch.
9. Apa tujuan suatu perangkai program-mikro?
10. Berikan perbandingan antara program-mikro dan program bahasa mesin.
11. Dengan menggunakan ALU pada Gambar 8-4 dan format yang disarankan pada Gambar 8-6 dan 8-7, berikan program-mikro horisontal dan vertikal untuk menjalankan operasi-operasi berikut ini:
    (a) $R_0 \leftarrow (R_1) + (R_2) + (R_3) + (R_4) + (R_5)$

(b) $R_0 \leftarrow (R_1) + (R_2) - (R_3) - (R_4)$

(c) $R_5 \leftarrow (R_0) + (R_5)$

12. Apa keuntungan suatu CPU *microprogrammable* irisan-bit?

13. Suatu program-mikro terdiri atas 1024 word dengan masing-masing 100 bit, tetapi hanya 120 instruksi-mikro yan berbeda yang digunakan. Berapa bit memori kendali yang dapat dihemat dengan menggunakan memori-nano?