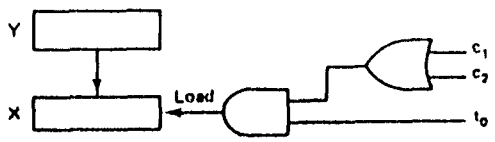
Kita telah menjabarkan konsep dasar RTL. Lebih lanjut kita akan melihat lebih banyak pernyataan RTL yang kita gunakan untuk menggambarkan suatu sistem tertentu.



Gambar 4-2 Transfer register yang diatur oleh sebuah fungsi pengendalian

4.3 SEBUAH CONTOH KOMPUTER SEDERHANA

Pada bagian ini, kita masuk ke dalam detail perancangan arsitektur untuk sebuah komputer yang sangat sederhana. Komputer yang kita gunakan didasarkan pada *Simplified Instructional Computer* (**SIC**) yang dipersembahkan oleh Beck (1985). Di sana, SIC dijabarkan oleh ukuran memori, ukuran word, register, format data, format instruksi, mode pengalamatan, kumpulan instruksi dan provisi input/output-putnya. Berdasarkan informasi ini, kita dapat menentukan sebuah rancangan arsitektur untuk SIC. Tentu saja, ada kemungkinan untuk rancangan lain yang sesuai dengan definisi SIC; namun satu rancangan sudah cukup untuk bahasan ini. Pertama-tama, kita akan menggambarkan struktur SIC (dimodifikasi sedikit dari definisi Beck) kemudian melihat pada detail rancangannya.

Struktur Mesin SIC

Mesin SIC terdiri atas CPU, unit memori dan minimal satu piranti I/O. CPU terdiri atas 13 register khusus. Tujuan dan ukuran register (dalam bit) diberikan dalam Tabel 4.1. Register *akumulator*, A, digunakan untuk semua operasi aritmatika dan logika. Dalam instruksi semacam ini, A selalu merupakan salah satu dari operand dan hasilnya selalu disimpan kembali dalam A. Register *indeks*, X, digunakan untuk menghitung address memori dari operand tertentu tergantung pada mode pengalamatan instruksi. Alamat memori dari instruksi berikutnya

TABEL 4.1 KUMPULAN REGISTER PADA SIC

Register	Size (bits)	Name	Intended use
A	24	Accumulator	Main calculational register
X	15	Index register	Indexed addressing
PC	15	Program counter	Contains address of next instruction
L	15	Linkage register	Stores return address for subroutines
IR	24	Instruction register	Stores current instruction
MBR	24	Memory buffer register	For input to or output from memory
MAR	15	Memory address register	Address of memory for all reads/writes
SW	11	Status word	Contains status information relative to previous instruction
C	2	Counter	Generates timing signals, t_0, t_1, t_2, t_3
INT	1	Interrupt flag	Signals that an interrupt has occurred
F	1	Fetch cycle flag	Specifies the fetch cycle
E	1	Execute cycle flag	Specifies the execute cycle
S	1	Start/stop flag	Enables C

pada suatu program disimpan dalam register *program counter*, PC. Register *linkage*, L, melayani penggunaan subroutine. Ketika subroutine dipanggil, kita perlu mengingat PC saat ini sehingga eksekusi dapat dilanjutkan dari titik ini jika subroutine telah berakhir. Kita harus memahami bagaimana hal ini dikerjakan ketika kita melihat *kumpulan instruksi*. *Instruction register*, **IR**, akan menyimpan instruksi memori saat ini yang sedang dijalankan. *Status word register*, **SW**, memiliki sejumlah field yang menyimpan informasi relatif pada operasi saat ini seperti program saat ini. Informasi yang disimpan dalam SW digunakan oleh programmer dan CPU untuk membuat keputusan berdasarkan hasil sebelumnya.

Akses ke dan dari unit memori hanya melalui register **MAR** dan **MBR**. Pada sebuah penulisan (*write*), data di dalam MBR ditulis ke word memori yang direferensikan oleh MAR:

$$M[MAR] \leftarrow (MBR)$$

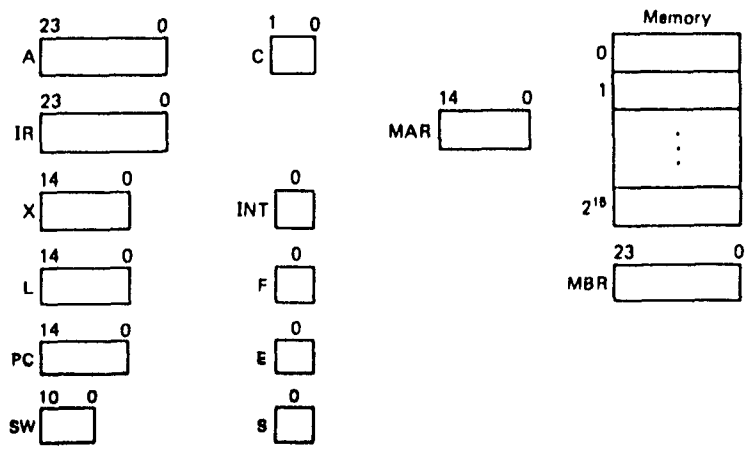
Pada sebuah pembacaan (*read*), data dibaca dari word memori yang direferensikan oleh MAR dan disimpan dalam MBR:

$$MBR \leftarrow (M[MAR])$$

Unit memori merupakan RAM yang terbentuk atas 2^{15} word dan ukuran tiap-tiap word adalah 24 bit. Hal ini berarti bahwa register yang digunakan untuk address memori mempunyai panjang 15 bit (PC, L, X dan MAR) dan yang menyimpan word memori secara lengkap memiliki panjang 24 bit (A, IR dan MBR). Ukuran dari SW berhubungan dengan jumlah field yang dikandungnya dan jumlah bit yang diperlukan oleh setiap field. Seperti yang akan kita lihat, 11 bit sudah cukup untuk SIC versi kita.

Untuk kesederhanaan, kita akan menganggap bahwa waktu akses unit memori kurang dari satu periode waktu. (Perhatikan bahwa hal ini *tidak* umum). Juga karena unit memori merupakan RAM yang synchronous, baris pengalamatan dari MAR dan juga baris data dari MBR harus tetap dipertahankan untuk durasi pengaksesan memori. Gambar 3-4 menunjukkan register dan unit memori pada SIC. Pembahasan lebih lanjut tentang bagaimana register tersebut bekerja, juga deskripsi register yang lain pada Tabel 4.1, akan diberikan pada bahasan berikut ini.

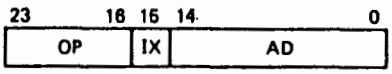
Input ke dan output ~~dari CPU~~ (tidak termasuk transfer ke atau dari memori) mencapai 1 byte dalam satu waktu. Informasi ditransfer antara 8 bit paling kanan pada register A dengan piranti I/O. Tiap piranti pada mesin diwakili oleh sebuah kode 8 bit yang dijabarkan seperti operand pada instruksi yang berhubungan dengan I/O.



Gambar 4-3 Kumpulan register dan memori pada SIC

Data untuk SIC terdiri dari integer ataupun karakter. (Tidak ada perangkat keras *floating-point* pada mesin ini). Integer disimpan sebagai angka 24 bit, dimana angka negatif disimpan sebagai bentuk komplemen 2's-nya. Karakter diwakili oleh kode ASCII 8-bit.

Format instruksi pada SIC adalah



dimana OP adalah opcode 8-bit yang memerinci operasi yang akan dijalankan, IX merupakan flag indeks yang menunjukkan mode pengalamatan yang harus digunakan dan AD merupakan address memori operand 15-bit. Jika bit IX pada instruksi bernilai 0, maka operand disimpan di dalam M[AD]. Hal ini disebut sebagai **pengalamatan langsung** (*direct addressing*). Jika bit IX bernilai 1, maka operand akan disimpan di dalam M[AD + (X)]. Yaitu register indeks yang digunakan untuk menyeimbangkan address yang ditentukan pada instruksi. Hal ini disebut sebagai **pengalamatan berindeks** (*indexed addressing*). Perhatikan bahwa instruksi hanya menjabarkan satu operand. Untuk operasi aritmatika dan logika, operand kedua secara implisit adalah nilai register A dan hasil operasinya selalu disimpan kembali dalam A.

Kumpulan Instruksi untuk SIC

Ada 21 instruksi SIC yang diberikan pada Tabel 4.2. Dalam tabel ini, *m* menunjukkan address memori dari operand dan (*m*) menunjukkan nilai yang disimpan pada address memori tersebut. Opcode instruksinya ditulis dalam notasi heksadesimal.

JSUB dan **RSUB** merupakan dua instruksi yang berhubungan dengan subroutine. JSUB menyimpan PC saat ini ke L dan kemudian melompat ke subroutine dengan menyimpan operand ke PC. RSUB kembali dari subroutine dengan melompat ke lokasi yang dinyatakan oleh L.

Instruksi **TD** digunakan untuk menguji piranti I/O sebelum berusaha untuk membaca dari atau menulis ke piranti tersebut. Hasil pengujian tersebut disimpan di dalam *kode kondisi* (*condition code*), field CC, pada SW. Panjang field ini 2 bit dan digunakan untuk mewakili salah satu dari tiga nilai: <, = dan >. Jika instruksi TD dijalankan, nilai field CC akan di-*set* menurut kode berikut ini:

< menunjukkan bahwa piranti telah siap.

TABEL 4.2 KUMPULAN INSTRUKSI PADA SIC

Instruction	Mnemonic	Opcode	Effect
Add	ADD *m*	00	A ← (A) + (*m*)
And	AND *m*	01	A ← (A) AND (*m*)
Compare	COMP *m*	02	(A):(*m*); CC ← result
Jump	J *m*	03	PC ← *m*
Jump Equal	JEQ *m*	04	PC ← *m* if CC set to =
Jump Greater Than	JGT *m*	05	PC ← *m* if CC set to >
Jump Less Than	JLT *m*	06	PC ← *m* if CC set to <
Jump Subroutine	JSUB *m*	07	L ← (PC); PC ← *m*
Load A	LDA *m*	08	A ← (*m*)
Load L	LDL *m*	09	L ← (*m*)
Load X	LDX *m*	0A	X ← (*m*)
Or	OR *m*	0B	A ← (A) OR (*m*)
Read Device	RD *m*	0C	A[0..7] ← byte from device (*m*)
Return Subroutine	RSUB	0D	PC ← (L)
Store A	STA *m*	0E	*m* ← (A)
Store L	STL *m*	0F	*m* ← (L)
Store SW	STSW *m*	10	*m* ← (SW)
Store X	STX *m*	11	*m* ← (X)
Subtract	SUB *m*	12	A ← (A) - (*m*)
Test Device	TD *m*	13	Test device (*m*); CC ← result
Write Device	WD *m*	14	Device (*m*) ← (A[0..7])
Interrupt Return	IRT	15	PC ← (M[0]); SW[MASK] ← 0

= menunjukkan bahwa piranti sedang sibuk dan tidak dapat digunakan pada saat itu.

> menunjukkan bahwa piranti tidak beroperasi.

Kemudian field ini digunakan oleh instruksi jump seperti yang telah ditentukan oleh program. Contoh bagaimana instruksi tersebut harus digunakan oleh programmer diberikan pada kode yang ditunjukkan dalam Tabel 4.3. Dalam routine ini, piranti X secara terus menerus diuji hingga siap atau hingga ditemukan tidak beroperasi. Dalam kasus berikutnya, program akan menyebabkan lompatan ke subroutine error. Jika pirantinya telah siap, maka program akan membaca sebuah byte data dari piranti X dan kemudian selesai.

Tambahan bagi instruksi TD; instruksi **COMP** juga men-*set* field CC. Nilai yang disimpan field CC setelah sebuah instruksi COMP menggambarkan hubungan antara A dan operand instruksi.

TABEL 4.3 CONTOH KODE UNTUK MEMBACA 1 BYTE DARI PIRANTI X

TEST	TD	"X"
	JEQ	TEST
	JGT	ERROR
	RD	"X"
	J	END
ERROR	JSUB	EROUTINE
END	STOP	

Instruksi **IRT** digunakan oleh *interrupt handler* agar menyebabkan lompatan kembali ke tempat dimana CPU berada sebelum interupsi terjadi. Jika interupsi terjadi, CPU akan menyimpan PC saat ini ke dalam memori pada address 0. Untuk kembali dari sebuah interupsi, isi dari alamat memori ini harus di-*load* kembali ke dalam PC. Seperti yang akan dijelaskan nanti, interrupt mask juga dibersihkan pada saat itu. Instruksi ini hanya dimaksudkan untuk piranti lunak (sistem) *interrupt handler* dan tidak diterbitkan atau dapat diakses pada bahasa assembly pemakai-akhir (*end-user*).

Instruksi-instruksi lainnya adalah untuk operasi aritmatika dan logika, transfer dari pengendalian (jump), *loading register*, *storing register* atau membaca dari dan menulis ke piranti I/O. Meskipun instruksi-instruksi lainnya mungkin dibutuhkan oleh jenis komputer umum (*general-purpose computer*), instruksi-instruksi tersebut telah cukup memberikan gambaran dasar tentang bagaimana seluruh bagian dari sebuah komputer bekerja bersama-sama.

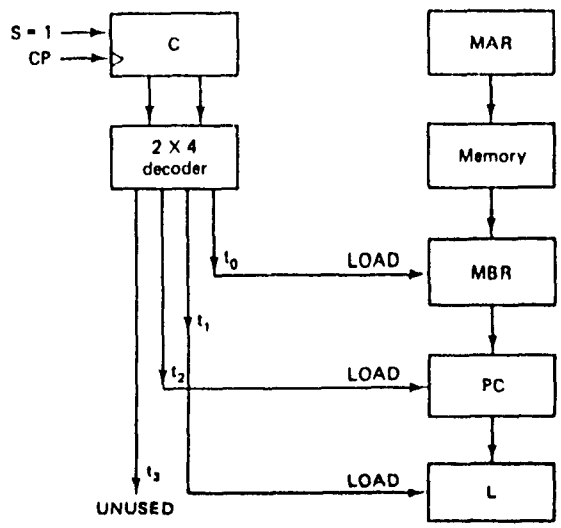
Pemilihan Waktu dan Pengendalian

CPU menjalankan program untuk satu instruksi pada satu satuan waktu. Agar CPU dapat menjalankan instruksi di dalam memori, pertama-tama ia harus memfetch instruksi dan kemudian menjalankannya. Selama pelaksanaan instruksi tersebut, ada kemungkinan terjadi sebuah interupsi. Sebuah interupsi merupakan sinyal kepada CPU tentang timbulnya suatu peristiwa. Contoh-contohnya interupsi: "*I/O completed*", "*program tried to devide by zero*" dan "*time expired*". Jika interupsi muncul maka CPU perlu memperhatikan sinyal tersebut dan mentransfer pengendalian kepada program *interrupt handler*. CPU akan mengkoordinasikan ketiga tugasnya dengan men-*set* suatu siklus: yaitu **siklus fetch, siklus eksekusi** dan **siklus interupsi**. (Tidak diperlukan siklus translasi address pada

mesin ini). Dalam setiap siklus, sejumlah operasi-mikro dilakukan untuk menjalankan tugas-tugas yang perlu.

Agar semuanya dikerjakan dengan benar, maka operasi-mikro dari setiap siklus harus dikerjakan dengan urutan tertentu. Salah satu cara untuk menyediakan urutan yang benar adalah dengan menggunakan counter sekuensial untuk mengendalikan operasi-mikro yang akan dijalankan nantinya. Counter sekuensial 2-bit, **C**, dapat digunakan untuk membuat empat sinyal waktu: yaitu t_0, t_1, t_2 dan t_{30}. Hal ini dilakukan dengan meningkatkan *C* pada setiap pulsa waktu. (Karena hanya terdapat 2 bit, maka penambahan akan kembali ke 00 setelah mencapai 11). Flag **S** digunakan untuk mengendalikan kapan C ditingkatkan. Jika S bernilai 1, C akan bertambah pada setiap pulsa waktu dan jika S bernilai 0, C tidak akan berubah. (S ditentukan dengan saklar manual yang diletakkan pada kotak komputer). Perhatikan proses *setup* pada Gambar 4-4. Penggunaan tiga sinyal waktu sebagai input *load* ke register akan menyebabkan register di-load dengan urutan tertentu: pertama MBR, kemudian L dan kemudian PC. Urutan operasi-mikro ini dapat juga digambarkan dengan pernyataan transfer register pada Tabel 4.4.

Dengan menggunakan prinsip yang sama dari pengendalian urutan, kita dapat men-*setup* siklus fetch, eksekusi dan interupsi. Namun jika kita menggunakan sinyal waktu yang sama untuk setiap siklus, maka operasi mikro dari siklus



Gambar 4-4 Urutan proses load pada MBR, PC dan L

TABEL 4.4 PERNYATAAN TRANSFER REGISTER UNTUK GAMBAR 4.4

t_0:	MBR ← (M[MAR])
t_1:	L ← (PC)
t_2:	PC ← (MBR[AD])

lainnya akan dijalankan pada waktu yang bersamaan. Untuk mencegah hal ini, CPU harus menyimpan track dimana siklus berada saat ini. Inilah fungsi register flag F dan E. Table 4.5 menghubungkan nilai F dan E dengan siklus CPU. Karena kita hanya memiliki tiga siklus, maka kombinasi dari E = F = 1 tidak dipakai. Dapat dilihat bahwa jika F = 1 dan E = 0 (FE'), maka CPU berada pada siklus fetch. Seperti yang akan kita lihat, jumlah terbesar dari sinyal waktu yang diperlukan oleh siklus manapun adalah empat. Ini berarti bahwa counter sekuensial, C, harus dirancang untuk menghasilkan empat sinyal waktu. Hal ini juga berarti bahwa meskipun sebuah siklus tidak memerlukan keempat periode siklus tersebut, maka siklus berikutnya tidak dapat diteruskan sampai t_0 berikutnya. Karena itu, prinsip perancangan yang dihasilkan adalah untuk meminimalkan jumlah sinyal waktu yang diperlukan oleh semua siklus dan juga untuk memperoleh jumlah sinyal waktu yang hampir sama pada setiap siklus. Sekarang mari kita lihat detail setiap siklus.

Siklus Fetch. Fungsi utama dari siklus fetch adalah agar sebuah salinan instruksi berikutnya dari memori ke dalam CPU siap dikerjakan. Register PC berisi alamat memori dari instruksi berikutnya, sehingga tugas pertama adalah me-load nilai tersebut ke dalam MAR. Kemudian memori dapat dibaca dan menyebabkan word memori disimpan ke dalam MBR. Word ini kemudian ditransfer ke dalam IR (*instruction register*) untuk decoding selanjutnya. Karena

TABEL 4.5 SIKLUS-SIKLUS PADA CPU

F	E	
1	0	Siklus
0	1	fetch
0	0	eksekusi
1	1	interupsi (tidak terpakai)

pengalamatan komputer ini sederhana, maka pengalamatan memori operand tersebut dapat ditentukan saat itu juga, tidak pada siklus translasi pengalamatan yang terpisah. Jika field indeks, IX bernilai 1, maka pertama-tama register X ditambahkan pada field address instruksi tersebut. Kemudian MAR dapat di-*setup* dengan address operand yang benar. Agar PC menunjukkan word memori selanjutnya dengan benar selama siklus fetch, maka PC perlu ditingkatkan pada waktu tertentu selama siklus ini setelah digunakan. Dengan cara ini, PC juga akan berisi nilai yang benar ketika instruksi JSUB dijalankan. Jika instruksi tersebut ternyata sejenis instruksi jump, maka PC akan di-*set* kembali ke nilai yang benar selama siklus eksekusi. Hal terakhir harus dikerjakan dalam siklus ini adalah men-*setup* flag siklus untuk siklus eksekusi. Hal ini dilakukan dengan mematikan flag F dan men-*set* flag E. Pernyataan transfer register yang berhubungan dengan siklus fetch diberikan dalam Tabel 4.6.

Setiap pernyataan pada Tabel 4.6 hanya dikerjakan pada kondisi dimana FE' adalah benar dan pada sinyal waktu tertentu. Namun, perhatikan bahwa beberapa pernyataan sebenarnya dikerjakan pada sinyal waktu yang bersamaan, khususnya, t_3. Hal ini berarti bahwa ketiga pernyataan tersebut dapat dikerjakan pada waktu yang sama. Tidak ada masalah bagi hal ini selama pernyataan tersebut tidak harus dikerjakan dengan urutan khusus. Gambar 4-5 menunjukkan hubungan register dan memori pada siklus fetch.

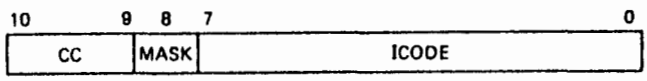
Siklus Interupsi. Jika terjadi siklus interupsi, beberapa tindakan harus dilakukan. Tergantung interupsinya, hal ini mungkin hanya memerlukan cukup sedikit instruksi. Selain itu ada kemungkinan bahwa tindakan untuk sebuah interupsi akan berubah di kemudian waktu. Hal ini bagi CPU berarti bahwa instruksi untuk menangani interupsi disimpan dalam memori, entah di mana. Instruksi-instruksi ini membentuk sebuah program yang disebut sebagai ***interrupt handler***. Oleh karena itu, fungsi dari siklus interupsi adalah untuk menyimpan PC

TABLE 4.6 PERNYATAAN TRANSFER REGISTER PADA SIKLUS FETCH

FE't_0	MAR ← (PC)
FE't_1	MBR ← (M[MAR]);PC ←(PC) + 1
FE't_2	IR ← (MBR)
FE't_3	IF IR[IX] = 1 THEN MAR ← (IR[AD]) + (X)
FE't_3	IF IR[IX] = 0 THEN MAR ← (IR[AD])
FE't_3	F ← 0;E ← 1

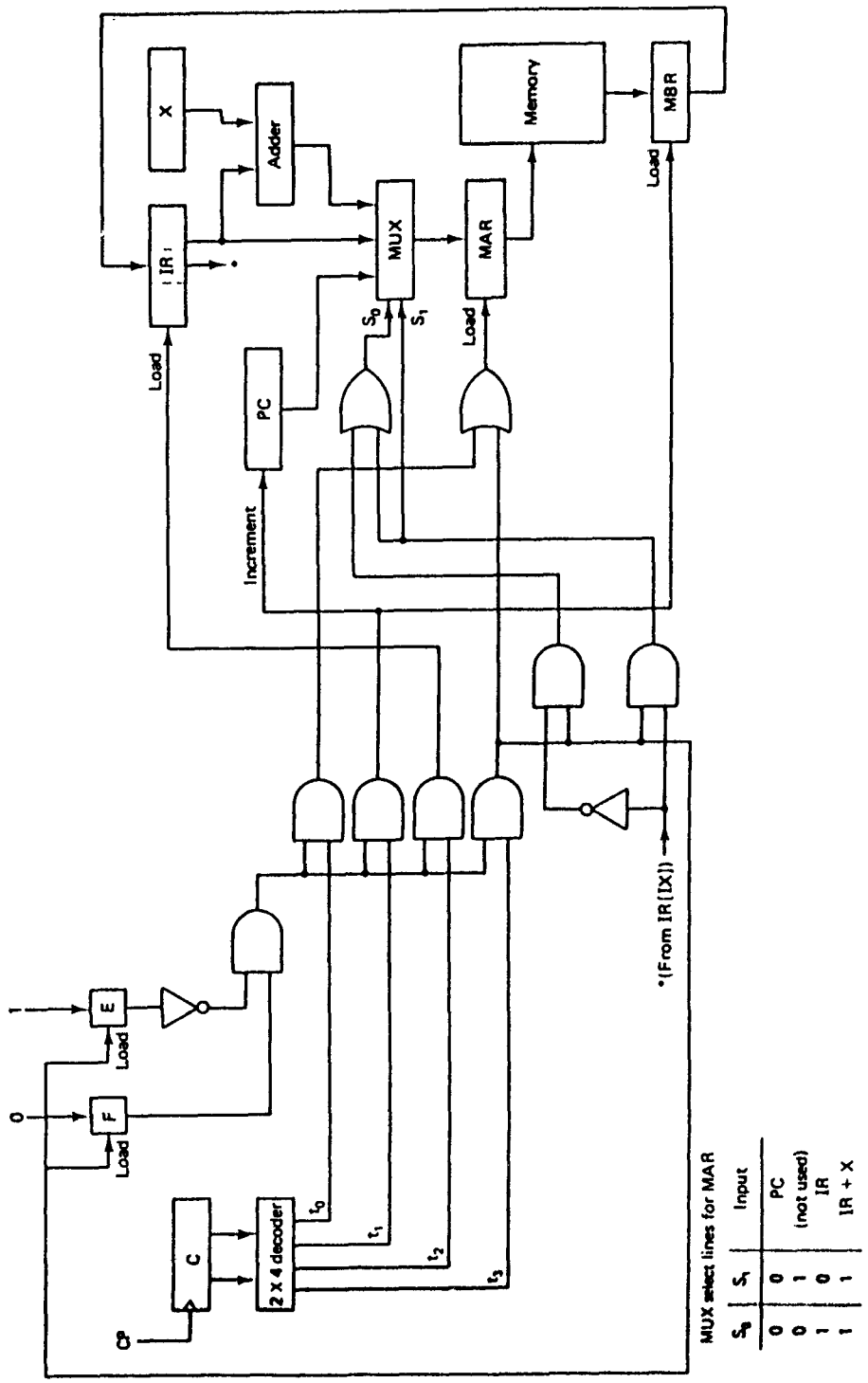
saat ini ke suatu tempat tertentu dan melompat ke kode *interrupt handler*. Kemudian kode ini akan di-fetch dan dilaksanakan satu instruksi per satu satuan waktu. Pada instruksi yang terakhir, *interrupt handler* akan menyebabkan lompatan kembali ke nilai PC tersimpan. Konsep yang penting di sini adalah bahwa *siklus interupsi tidak melayani interupsi*; ia hanya akan menyebabkan pengendalian diberikan kepada program tertentu yang akan mengintepretasikan dan melayani interupsi. Kita akan menganggap bahwa kode *interrupt handler* dimulai dari M[1] dan PC akan disimpan dalam memori pada M[0].

Pada mesin SIC, register SW terdiri atas dua field relatif untuk interupsi: sebuah field 8-bit, ICODE dan sebuah field 1-bit,



Field kode interupsi, ICODE, secara otomatis di-*set* ke nilai yang menunjukkan penyebab interupsi (jenis error, piranti memerlukan I/O, sistem dimulai, dsb.). Hal ini diatur oleh piranti yang menyebabkan interupsi dan diinterpretasikan oleh *interrupt handler*. Field MASK mengendalikan apakah interupsi diizinkan atau tidak. Field ini digunakan oleh mesin sederhana ini untuk mencegah adanya interupsi bercabang (*nested interruption*). Dengan kata lain, kita tidak perlu khawatir jika terjadi interupsi sewaktu interupsi lainnya sedang berlangsung. Dalam mesin yang lebih canggih, interupsi bercabang biasanya diizinkan sesuai prioritas interupsi tersebut. Tambahan bagi register SW; register INT juga terlibat dalam proses interupsi. Register 1-bit ini digunakan untuk menyampaikan sinyal kepada CPU bahwa telah terjadi sebuah interupsi. Hal ini dapat diatur oleh semua piranti I/O untuk interupsi yang berhubungan dengan I/O tersebut atau oleh CPU itu sendiri untuk interupsi program dan yang berhubungan dengan waktu. Setelah CPU menerima sinyal tersebut, ia akan membersihkan flag INT. (Interupsi akan dibahas lebih lanjut pada Bab 6).

Sekarang kita dapat membahas siklus interupsi secara detail. Dalam siklus tersebut, baik flag F dan E sama-sama bernilai 0. Pernyataan transfer register diberikan dalam Tabel 4.7. Perhatikan bahwa pada t_0, flag interupsi, INT, dibersihkan dan interrupt mask, SW[MASK], di-*set*. Jika akan terjadi interupsi yang lain pada saat itu, INT akan di-*set* kembali, namun SW[MASK] akan mencegah interupsi yang baru tersebut agar tidak menyela interupsi yang telah ada. Jika routine *interrupt handler* telah selesai melayani interupsi, ia akan memungkinkan terjadinya interupsi lagi dengan membersihkan SW[MASK]. (Hal ini dikerjakan



Gambar 4-5 Transfer register pada siklus fetch

TABEL 4.7 PERNYATAAN TRANSFER REGISTER PADA SIKLUS INTERRUPT

F'E't_0:	MBR[AD] ← (PC), INT ← 0, SW[MASK] ← 1
F'E't_1:	MAR ← 0, PC ← 1
F'E't_2:	M[MAR] ← (MBR)
F'E't_3:	F ← 1

oleh instruksi IRT pada routine *interrupt handler*). Perhatikan juga bahwa F perlu di-*set* bernilai 1 untuk siklus fetch jika E telah bernilai 0.

Siklus eksekusi. Siklus eksekusi ini berbeda dengan siklus fetch dan interupsi. dimana setiap instruksinya adalah unik. Satu-satunya persamaan yang dimiliki siklus eksekusi adalah pada sinyal waktu yang teakhir, t_3, flag F dan E di-*set* untuk siklus fetch ataupun interupsi. Interrupt mask, SW[MASK] harus diuji untuk apakah dapat terjadi interupsi. (Jika ya, SW[MASK] akan bernilai 0). Jika dapat terjadi interupsi, maka F dan E harus dibersihkan untuk siklus interupsi; Jika tidak maka F harus di-*set* dan E harus dibersihkan agar siklus fetch berikutnya dapat berlangsung. Sehingga, jika (NOT SW[MASK] AND INT} bernilai benar, maka F dan E harus dibersihkan. Tetapi, karena F telah bernilai 0 dalam siklus ini, akan lebih efisien untuk selalu membersihkan E dan kemudian men-*set* F hanya jika siklus interupsi *tidak* diperlukan. Siklus interupsi tidak diperlukan jika SW[MASK] bernilai 1 atau jika flag INT bernilai 0. Dengan demikian seluruh siklus eksekusi mencakup pernyataan transfer berikut ini:

$$F'Et_3: E \leftarrow 0; \text{IF (SW[MASK] OR NOT(INT)) THEN } F \leftarrow 1 \qquad (4.15)$$

Sekarang, jika kondisi pengendalian untuk siklus eksekusi hanya mencakup flag F dan E serta sinyal waktu, setiap instruksi akan dilaksanakan pada setiap siklus eksekusi. Sesungguhnya, hal ini tidak kita inginkan. Selama siklus eksekusi, kita hanya ingin menjalankan instruksi yang tersimpan dalam IR. CPU akan meyakinkan hal tersebut dengan menggunakan opcode instruksi tersebut (field OP pada IR) sebagai bagian dari kondisi pengendalian. Karena opcode bersifat unik untuk setiap instruksi maka hanya ada satu instruksi yang akan dijalankan selama siklus eksekusi yang diberikan. Untuk kesederhanaan, kita akan menganggap p_i mewakili opcode i pada kondisi pengendalian dari siklus eksekusi. (Opcode untuk setiap instruksi dapat dilihat pada Tabel 4.2).

Mari kita perhatikan siklus eksekusi untuk instruksi tertentu. Untuk instruksi aritmatika ADD (opcode = 0), siklus eksekusinya diperlihatkan oleh Tabel 4.8. Ingat kembali bahwa pada akhir siklus fetch, MAR di-*setup* oleh address memori operand tersebut. Dengan demikian pada siklus eksekusi ini, pembacaan memori diawali dari sebelah kanan t_0. Pernyataan transfer yang dikerjakan pada t_1 akan menyebabkan penambahan yang sebenarnya terjadi. Pada saat t_3, siklus berikutnya ditentukan dengan menggunakan pernyataan transfer pada Persamaan (4.15). Karena tidak ada hal lain yang harus dikerjakan, maka t_2 merupakan sinyal waktu yang menganggur. Seperti yang akan kita bahas pada Bab 6, siklus yang menganggur seperti ini dapat digunakan untuk *pencurian siklus* (*cycle stealing*). Penting untuk menyadari bahwa tidak perduli berapa banyak sinyal waktu yang diperlukan untuk instruksi tertentu yang sedang dijalankan, perubahan flag F dan E hanya dapat dikerjakan pada sinyal yang terakhir. Pada contoh kita, hanya satu sinyal waktu yang diperlukan untuk instruksi tersebut. Namun, jika F dan E dirubah saat t_1 ataupun t_2, maka siklus fetch atau interupsi akan mulai dijalankan tepat di tengah-tengah. Tentu saja hal ini menimbulkan malapetaka bagi sistem tersebut.

Untuk beberapa instruksi, proses pembacaan memori tidak diperlukan. Misalnya, perhatikan dua instruksi J dan LDL. Untuk instruksi J, kita ingin menyimpan address memori operand di dalam PC, sedangkan pada instruksi LDL, kita ingin menyimpan nilai pada memori yang dijabarkan oleh address operand. Dengan kata lain, kita melakukan pembacaan memori untuk mendapatkan nilai yang diperlukan bagi instruksi LDL tetapi tidak bagi instruksi J. Siklus eksekusi untuk kedua instruksi semacam ini diberikan dalam Tabel 4.9.

Untuk beberapa contoh lainnya, siklus eksekusi untuk instruksi JSUB dan RSUB diberikan dalam Tabel 4.10. (JSUB merupakan instruksi pada Gambar 4-4 dan Tabel 4.4).

Siklus eksekusi untuk instruksi yang lain dapat diturunkan dari informasi yang diberikan dalam Tabel 4.2.

Unit Fungsional

Satu-satunya komponen lain yang diperlukan oleh CPU untuk melaksanakan suatu instruksi adalah sirkuit kombinasional untuk mengerjakan operasi aritmatika dan logika komputer.

Sirkuit ini disebut dengan **unit fungsional**. Unit fungsional pada SIC adalah sebagai berikut:

TABEL 4.8 SIKLUS EKSEKUSI UNTUK INSTRUKSI ADD

$p_0F'Et_0$:	MBR ← (M[MAR])
$p_0F'Et_1$:	A ← (A) + (MBR)
$p_0F'Et_2$:	
$p_0F'Et_3$:	E ← 0;IF (SW[MASK] OR NOT (INT)) THEN F ← 1

TABEL 4.9 SIKLUS EKSEKUSI UNTUK INSTRUKSI J DAN LDL

Instruksi J	$p_3F'Et_0$:	PC ← (IR[AD])
(opcode = 3)	$p_3F'Et_1$:	E ← 0;IF (SW[MASK] OR NOT (INT)) THEN F ← 1
	$p_3F'Et_2$:	
	$p_3F'Et_3$:	
Instruksi LDL	$p_9F'Et_0$:	MBR ← (M[MAR])
(opcode = 9)	$p_9F'Et_1$:	L ← (MBR[AD])
	$p_9F'Et_2$:	E ← 0;IF (SW[MASK] OR NOT (INT)) THEN F ← 1
	$p_9F'Et_3$:	

TABEL 4.10 SIKLUS EXECUTE UNTUK INSTRUKSI JSUB DAN RSUB

Instruksi JSUB	$p_7F'Et_0$:	MBR ← (M[MAR])
(opcode = 7)	$p_7F'Et_1$:	L ← (PC)
	$p_7F'Et_2$:	PC ← (MBR[AD])
	$p_7F'Et_3$:	E ← 0;IF (SW[MASK] OR NOT (INT)) THEN F ← 1
Instruksi RSUB	$p_DF'Et_0$:	PC ← (L)
(opcode = D)	$p_DF'Et_1$:	
	$p_DF'Et_2$:	
	$p_DF'Et_3$:	E ← 0;IF (SW[MASK] OR NOT (INT)) THEN F ← 1

1. Dua **adder biner**: satu untuk mode pengalamatan indeks dan satu lagi untuk operasi ADD dan SUB.
2. Satu unit **COMPARE logika** untuk operasi COMP.
3. Satu unit **AND logika** untuk operasi AND.
4. Satu unit **OR logika** untuk operasi OR.

Kecuali untuk mode pengalamatan adder, input untuk setiap unit fungsional adalah dari register A dan MBR, dan output dikembalikan ke A. Input dan output pada mode pengalamatan adder dapat dilihat pada Gambar 4-5.

Startup pada Sistem

Setiap komputer pasti memiliki cara-cara tertentu untuk memulai segala sesuatunya sewaktu ia dinyalakan. Hal ini biasanya termasuk inisialisasi register-register tertentu dan kemudian menjalankan instruksi-instruksi tertentu yang menangani inisialisasi tersebut selebihnya. Instruksi-instruksi khusus ini dapat berupa *resident* (menduduki) pada bagian memori tertentu atau secara otomatis dibaca ke dalam memori dari piranti I/O khusus.

Pada mesin SIC, penyalaan sistem menyebabkan S dan F di-*set* bernilai 1; E dan C dibersihkan; SW[ICODE] di-*set* pada "startup"; dan PC di-*set* bernilai 1. Hal ini menyebabkan CPU langsung menjalankan routine *interrupt handler* yang resident, yang mulai mengintepretasikan interupsi startup. Tambahan bagi proses startup; sistem, diperlukan juga agar dapat me-*reset* sistem tanpa mematikannya dan kemudian menyalakannya kembali. Pada SIC hal ini dapat dilakukan dengan menggunakan **saklar start/stop** manual. Dengan mengubah flip-flop mekanis menjadi 1, akan menyebabkan sistem menjalani routine power-up. Memindahkan saklar ke 0 hanya membersihkan S, dengan demikian akan memberhentikan counter dan memberhentikan seluruh mesin.

4.4 PELACAKAN EKSEKUSI PADA SEBUAH INSTRUKSI SIC

Setelah menjabarkan siklus CPU, mekanisme pemilihan waktu dan pengendalian serta unit-unit fungsional pada SIC, sebenarnya kita telah mendefinisikan bagaimana register dan unit-unit pada komputer saling berhubungan dan bagaimana data ditranfer di antara mereka. Pada bagian ini, kita akan melacak suatu instruksi melalui keseluruhan siklus fecth dan eksekusinya sebagai sebuah contoh tentang cara kerja SIC.

Untuk memulainya, anggaplah bahwa isi unit memori adalah seperti yang ditunjukkan pada Gambar 4-6(a). Kita akan melacak melalui pelaksanaan instruksi yang sekarang tersimpan dalam M[10]. Kita anggap instruksi sebelumnya, M[9], telah dijalankan dan isi dari register CPU adalah seperti yang ditunjukkan pada Gambar 4-6(b). Perhatikan bahwa flag F dan E telah disusun untuk siklus fetch berikutnya, tetapi counter C belum kembali ke 00.