

Arsitektur Komputer

Struktur dan Fungsi

Disusun oleh:

Elly Mufida

Henny Leidiyana

Eva Rachmawati

Hylenarti Hertyana

KATA PENGANTAR

Buku ini diperuntukkan bagi anda yang telah memiliki pemahaman menganai konsep dasar sistem digital, karena sistem komputer bekerja dalam lingkungan sistem digital. Seperti yang telah diketahui secara umum, perangkat keras sistem komputer terdiri dari prosesor, memori, modul Input/output, dan sistem interkoneksi.

Prosesor sebagai pengendali pusat memiliki empat fungsi utama, yaitu: data processing, Data storage, Data movement, dan Control. Untuk menjalankan fungsi tersebut, prosesor bergantung penuh pada format instruksi pada set instruksiannya, sehingga rancangan komputer dapat dikelompokkan berdasarkan set instruksiannya menjadi RISC dan CISC. Selama melakukan eksekusi instruksi, prosesor melakukan siklus mesin yang secara umum terdiri dari siklus pejemputan (*fetch cycle*), siklus eksekusi (*execute cycle*), dan siklus interupsi (*interrupt cycle*).

Buku ini terdiri dari 11 Bab, yaitu: Pengenalan dan Evolusi Komputer, Internal Memori, Eksternal Memori, Sistem Interkoneksi, Komputer Aritmatika, Struktur dan Fungsi Prosesor, Set Instruksi, Assembler Language, Reduce Instruction Set Komputer, dan Control Unit. Pembahasan diawali dengan arsitektur von Neumann yang merupakan konsep “*stored program Computer*” pertama dalam sejarah perkembangan komputer, evolusi komputer, sistem memori, Input-output, sistem interkoneksi, dan prosesor sebagai pengendali pusat sistem komputer. Pada umumnya, perkembangan komputer selanjutnya mengacu pada arsitektur von Neumann. Prosesor sebagai komponen pengendali utama pada sebuah sistem komputer tidak akan berguna jika tidak dilengkapi dengan internal memori dan modul I/O. Sistem interkoneksi adalah bagian yang juga sangat menentukan kinerja dari sebuah sistem komputer secara keseluruhan. Kemudian pembahasan mengenai set instruksi berguna untuk menentukan rancangan sebuah prosesor, mengikuti arsitektur RISC atau CISC. Terakhir, pengetahuan tentang assembler language juga tidak boleh diabaikan oleh pembaca, karena pada setiap menganalisa perilaku prosesor selalu dilakukan dengan menggunakan perintah-perintah dari assembler language.

Harapan penulis, semoga buku ini dapat membantu semua pembaca khususnya mahasiswa yang sedang mempelajari rancangan perangkat keras sistem komputer. Tak ada gading yang tak retak, begitu kata pepatah. Untuk itu penulis akan terus melakukan perbaikan dan pemutakhiran terhadap buku ini, untuk kesempatan penerbitan berikutnya.

Jakarta, Maret 2021

Tim Penulis

DAFTAR ISI

KATA PENGANTAR.....	ii
DAFTAR ISI.....	iii
DAFTAR GAMBAR.....	viii
DAFTAR TABEL	xiii
1. Pengenalan dan Evolusi Komputer	1
1.1. Pengenalan	1
A. Arsitektur dan Organisasi.....	1
B. Struktur dan Fungsi	1
C. Arsitektur von Neumann.....	3
D. Struktur Komputer Multicore	5
E. <i>Embedded System</i>	7
F. <i>Internet of Things</i>	8
G. Mikroprosesor versus Mikrokontroler.....	9
1.2. Evolusi Komputer.....	11
A. Generasi Pertama: Tabung Hampa	11
B. Generasi Kedua: Transistor	16
C. Generasi Ketiga: IC	17
D. Generasi Masa Depan	21
1.3. Fungsi Komputer dan Interkoneksi.....	24
A. <i>Instruction Fetch</i> dan <i>Execute</i>	26
B. Interupsi dan Siklus Instruksi	31
C. Multiple Interrupt	35
D. Fungsi I/O	37
2. Internal Memori	38
2.1. Hirarki Memori	38
2.2. Karakteristik Memori Sistem Komputer	40
A. <i>Location</i>	41
B. <i>Capacity</i>	41
C. Unit of Transfer	41
D. Metode Akses (<i>Access Method</i>)	41
E. <i>Performance</i>	42
F. <i>Physical Type</i>	42
2.3. <i>Cache Memory</i>	43
A. <i>Cache Address</i>	47
B. <i>Cache Size</i>	48
C. Fungsi Pemetaan	49
D. <i>Replacement Algorithm</i>	59
E. <i>Write Policy</i>	59
2.4. <i>Read Only Memory</i> (ROM).....	60
A. Programmable-ROM (PROM).....	61
B. EPROM	61
C. EEPROM	62
2.5. Flash Memory	62
2.6. <i>Random Access Memory</i> (RAM)	62
D. Dynamic RAM (DRAM)	63
E. Static RAM (SRAM).....	64
2.7. Deteksi Kesalahan.....	65
3. Eksternal Memori	68
3.1. Magnetik Disk	68

3.2. <i>Redundant Array of Independent Disk (RAID)</i>	74
3.3. <i>Solid State Drives (SSD)</i>	83
4. Input/Output	86
4.1. External Devices.....	86
4.2. Modul I/O (I/O Modules).....	89
4.3. I/O Terisolasi dan I/O Terpeta-Memori.....	91
4.4. Teknik Pelayanan I/O.....	92
A. Programmed I/O	93
B. Interrupt Driven I/O	95
C. <i>Direct Memory Access (DMA)</i>	104
D. DMA Controller Intel 8237A.....	106
4.5. Saluran I/O dan Prosesor.....	108
A. Evolusi Fungsi I/O	108
B. Karakteristik Saluran I/O	109
5. Sistem Interkoneksi	111
5.1 Struktur Interkoneksi.....	111
5.2. Bus Interkoneksi	112
A. Struktur Bus.....	113
B. Hierarki Multi-Bus	114
C. Interkoneksi Point-to-Point.....	116
5.3. Elemen Desain Bus.	117
D. Dedicated dan Multiplexed.....	120
E. Metode Arbitrase	121
F. <i>Timing</i>	121
G. <i>Bus Width</i>	123
H. Jenis Transfer Data	123
5.4. PCI	124
A. Struktur Bus PCI.....	126
B. Perintah PCI.....	129
C. Transfer Data.....	130
D. Arbitrasi.....	131
5.5. PCI Express.....	133
5.6. Universal Serial Bus (USB).....	139
6. Komputer Aritmatika	142
6.1. ALU.....	142
6.2. Representasi Sembarang Bilangan Biner.....	142
A. Representasi <i>Sign-Magnitude</i>	143
B. Representasi Komplemen-2 (<i>Two's Complement</i>).....	144
6.3. Aritmatika Integer.....	146
A. Negasi.....	146
B. Penjumlahan dan Pengurangan	147
C. Perkalian.....	149
D. Pembagian (<i>Division</i>).....	153
6.4. Representasi Floating Point.....	155
7. Struktur dan Fungsi Prosesor	162
7.1 Organisasi Prosesor	162
7.2. Organisasi Register	163
A. <i>User Visible Register</i>	163
B. Kontrol dan Status Register.....	164
7.3. Siklus Insturksi	166
A. Siklus Tidak Langsung (<i>Indirect Cycle</i>)	166

B. Aliran Data Pada Siklus Instruksi.....	167
7.4. Prosesor Keluarga x86	169
A. Organisasi Register.....	169
B. <i>Control Register</i>	171
C. MMX Register.....	173
D. Pemrosesan Interupsi	173
E. Tabel Vektor Interupsi.....	174
F. Interrupt Handling.....	174
7.5. Pipelining Instruksi.....	175
8. Set Instruksi.....	183
8.1 Karakteristik Instruksi Mesin	183
A. Representasi Instruksi	184
B. Type Instruksi	185
C. Jumlah Alamat.....	186
8.2. Type <i>Operand</i>	187
A. Type Data Numerik atau Angka	188
B. Type Data Karakter.....	188
C. Type Data Logika	191
8.3. Type Operasi.....	191
A. Transfer data	192
B. Aritmatika.....	193
C. Logika	194
D. Kendali sistem	196
E. Transfer kendali.....	196
8.4. Mode Pengalamatan	201
A. <i>Immediate Addressing</i>	202
B. <i>Direct Addressing</i>	202
C. <i>Indirect Addressing</i>	203
D. <i>Register Addressing</i>	203
E. <i>Register Indirect Addressing</i>	204
F. <i>Displacement Addressing</i>	204
G. <i>Stack Addressing</i>	206
8.5. Format Instruksi.....	206
A. Panjang instruksi	206
B. Alokasi Bit.....	207
9. Assembler Language.....	211
9.1. Elemen Assembler Language.....	215
A. Label	215
B. <i>Operand</i>	216
C. <i>Comment</i>	217
9.2. Jenis Pernyataan Bahasa Rakitan	217
A. Mnemonic	217
B. Directive	217
C. Definisi Makro	218
10. Reduce Instruction Set Computer	220
10.1. Karakteristik Eksekusi Instruksi.....	222
10.2. Penggunaan Register File yang Besar	225
10.3. Register Windows.....	225
11. Control Unit	227
11.1. Mikro Operasi	227
A. <i>Fetch Cycle</i>	228

B. Siklus Tidak Langsung (<i>Indirect Cycle</i>).....	229
C. <i>Interrupt Cycle</i>	229
D. <i>Exexecute Cycle</i>	230
E. Instruction Cycle (Siklus Instruksi)	231
11.2. Organisasi Internal Prosesor.....	231
11.3. Organisasi Internal Prosesor.....	235
11.4. Intel 8085.....	236
Daftar Pustaka.....	242
INDEX	243
GLOSARIUM	246
RANGKUMAN	248

DAFTAR GAMBAR

Gambar 1.1. Skema Struktur Komputer.....	2
Gambar 1.2. Skema Arsitektur Komputer von Neumann.....	3
Gambar 1.3. Siklus Instruksi Komputer von Neumann.....	4
Gambar 1.4. Perbandingan arsitektur komputer Harvard dan von Neumann/Princeton.....	5
Gambar 1.5. Skema Sebuah Komputer Multicore	6
Gambar 1.6. Contoh Organisasi <i>Embedded System</i>	7
Gambar 1.7. Skema Dasar <i>Chip</i> Mikrokontroler	10
Gambar 1.8. Struktur Umum Komputer IAS	12
Gambar 1.9. Struktur komputer IAS	13
Gambar 1.10. Format sebuah <i>Word</i> pada memori IAS.....	14
Gambar 1.11. Siklus Instruksi Komputer IAS.....	15
Gambar 1.12. Elemen dasar komputer	18
Gambar 1.13. <i>Chip</i> dan <i>Gate</i>	19
Gambar 1.14. Pertumbuhan jumlah transistor pada sebuah IC berdasarkan hukum Moore	20
Gambar 1.15. Pendekatan Hardware dan Software.....	24
Gambar 1.16. Komponen Komputer: Tampilan Tingkat Atas	26
Gambar 1.17. Siklus Instruksi Dasar.....	26
Gambar 1.18. Format Instruksi dan Bilangan Integer.....	27
Gambar 1.19. Contoh eksekusi program (isi memori dan register dalam format heksadesimal)	28
Gambar 1.20. State Diagram Siklus Instruksi	30
Gambar 1.21. Arah Kendali Program Tanpa dan Dengan Interupsi.....	30
Gambar 1.22. Pengalihan kendali melalui Interupsi.....	32
Gambar 1.23. Siklus Instruksi dengan Interupsi	32
Gambar 1.24. Program Timing: Short I/O Wait	33
Gambar 1.25. Program Timing: Long I/O Wait	34
Gambar 1.26. State Diagram Siklus Instruksi dengan Interrupt	35
Gambar 1.27. Transfer Pengendalian dengan <i>Multiple Interrupt</i>	36
Gambar 1.28. Contoh Urutan waktu dari Multiple Interrupts.....	37
Gambar 2.1 Hirarki Memori pada Sistem Komputer	39
Gambar 2.2. Organisasi <i>Cache</i> dan <i>Main Memory</i>	44
Gambar 2.3. Struktur <i>Cache</i> dan <i>Main Memory</i>	45
Gambar 2.4. Siklus Operasi Baca pada <i>Cache</i>	46
Gambar 2.5. Organisasi <i>Cache</i> yang Umum.....	46
Gambar 2.6. Logika dan Fisik <i>Cache</i>	48
Gambar 2.7. Mapping dari <i>Main Memory</i> ke <i>Cache</i> : <i>Direct</i> dan <i>Associative</i>	50
Gambar 2.8. Organisasi Direct Mapping <i>Cache</i>	51
Gambar 2.9. Contoh Direct Mapping	52
Gambar 2.10. Organisasi Associative Mapping	53
Gambar 2.11. Contoh Associative Mapping	55
Gambar 2.12. Mapping dari <i>main memory</i> ke <i>cache</i> : <i>k-way Set Assosiative</i>	56
Gambar 2.13. Organisasi Set Associative K-way	57
Gambar 2.14. Contoh Two-Way Set Associative Mapping	58
Gambar 2.15. Pin-out EPROM 2716.....	61
Gambar 2.16. Pin-out DRAM TMS4464	64
Gambar 2.17. Multiplekser alamat untuk DRAM TMS4464	64

Gambar 2.18. Pin-out SRAM TMS4016	65
Gambar 2.19. Fungsi Error Correction Code	65
Gambar 2.20. Hamming Code <i>Error Correction Code</i>	66
Gambar 3.1. Magnetisasi <i>Read/Inductive Write</i> pada Head	69
Gambar 3.2. Layout Disk Data	70
Gambar 3.3. Perbandingan Metode Layout Disk	71
Gambar 3.4. Format Winchester Disk (Seagate ST506)	71
Gambar 3.5. Pewaktu I/O Transfer pada Disk	73
Gambar 3.6. Level RAID	76
Gambar 3.7. Data Mapping untuk Array RAID Level 0	78
Gambar 3.8. Arsitektur <i>Solid State Drive</i>	85
Gambar 4.1. Model Generik dari Modul I/O	87
Gambar 4.2. Diagram Blok External Device	87
Gambar 4.3. Blok Diagram Modul I/O	91
Gambar 4.4. Tiga Tehnik untuk Menginput Block Data	93
Gambar 4.5. Memory-Mapped dan Isolated I/O	95
Gambar 4.6. Pemrosesan Instrupsi Sederhana	97
Gambar 4.7. Perubahan dalam Memori dan Register pada sebuah Interupsi	98
Gambar 4.8. Penggunaan PIC 825C59A	100
Gambar 4.9. PPI Intel 8255A	101
Gambar 4.10. Control Word PPI Intel 8255A	102
Gambar 4.11. Antarmuka Keyboard/Display dengan PPI Intel 8255A	103
Gambar 4.12. Blok Diagram DMA yang umum	104
Gambar 4.13. DMA dan Interrupt Breakpoint selama Siklus Instruksi	105
Gambar 4.14. Konfigurasi Alternatif DMA	106
Gambar 4.15. Sytem Bus pada DMAC 8237	107
Gambar 4.16. Arsitekur I/O Channel	110
Gambar 5.1. Modul Komputer	112
Gambar 5.2. Skema interkoneksi bus	113
Gambar 5.3. Realisasi Fisik Arsitektur Bus yang umum	114
Gambar 5.4. Contoh Konfigurasi Bus	116
Gambar 5.5. Diagram Pewaktu Operasi Bus Sinkron	119
Gambar 5.6. Diagram Pewaktu Operasi Bus Asinkron	120
Gambar 5.7. Pewaktu operasi Synchronous Bus	122
Gambar 5.8. Pewaktu operasi Asynchronous Bus	123
Gambar 5.9. Type Transfer Bus Data	124
Gambar 5.10. Contoh Konfigurasi PCI	126
Gambar 5.11. Operasi Read PCI	131
Gambar 5.12. PCI Bus Arbiter	131
Gambar 5.13. PCI Bus Arbitration antara Dua Master	132
Gambar 5.14. Konfigurasi Khas Menggunakan PCIe dan QPI	134
Gambar 5.15. Layar pada Protokol PCIe	134
Gambar 5.16. Distribusi PCIe Multiline	135
Gambar 5.17. Blok Diagram PCIe Transmit and Receive Block	136
Gambar 5.18. Format Unit Data Protokol PCIe	138
Gambar 6.1. Skema <i>Input/Output</i> pada ALU	142
Gambar 6.2. Nilai Posisi antara Komplemen ke-2 dan Desimal	145
Gambar 6.3. Penambahan Bilangan dengan Representasi Komplemen-2	148

Gambar 6.4. Pengurangan Bilangan dengan Representasi Komplemen-2 (M-S)	148
Gambar 6.5. Blok Diagram Hardware untuk Penjumlahan dan Pengurangan	149
Gambar 6.6. Perkalian Biner	150
Gambar 6.7. Perkalian dua bilangan 4-bit <i>Unsigned Integer</i> menghasilkan 8 bit	151
Gambar 6.8. Perbandingan Perkalian <i>Unsigned Integer</i> dengan Komplemen-2	151
Gambar 6.9. <i>Flowchart Booth's Algorithm</i> untuk Perkalian Komplemen-2	152
Gambar 6.10. Contoh Booth's Algorithm (7x3)	152
Gambar 6.11. Contoh Penggunaan Booth's Algorithm.....	153
Gambar 6.12. Perkalian Biner	153
Gambar 6.13. <i>Flowchart</i> untuk Pembagian Biner tidak bertanda	154
Gambar 6.14. Contoh restoring pembagian komplemen-2.....	155
Gambar 6.15. Format Floating Point 32 bit	156
Gambar 6.16. Bilangan yang disajikan pada format 32 bit.....	157
Gambar 6.17. Format Bilangan Foating Point IEEE 754	158
Gambar 7.1. CPU dengan System Bus.....	162
Gambar 7.2. Struktur Internal CPU	163
Gambar 7.3. Contoh Organisasi Register Mikroprosesor	165
Gambar 7.4. Siklus Instruksi <i>Indirect</i>	166
Gambar 7.5. State Diagram dari Siklus Instruksi.....	167
Gambar 7.6. Aliran data dari Fetch Cycle	167
Gambar 7.7. Aliran data dari <i>Indirect Cycle</i>	168
Gambar 7.8. Aliran Data dari Interrupt Cycle	168
Gambar 7.9. Register EFFLAGS x86	171
Gambar 7.10. Register Control x86.....	172
Gambar 7.11. Mapping Register MMX ke Register Floating Point	173
Gambar 7.12. State Diagram Siklus Instruksi.....	176
Gambar 7.13. Dua Stage <i>Pipeline</i> Instruksi	177
Gambar 7.14. Time Diagram Untuk Operasi <i>Pipeline</i> Instruksi Dengan Enam Stage	178
Gambar 7.15 <i>Flowchart</i> 6 Stage <i>Pipeline</i> Instruksi	179
Gambar 7.16. Efek dari Cabang Bersyarat pada Operasi <i>Pipeline</i> Instruksi.....	179
Gambar 7.17. Alternatif Penggambaran <i>Pipeline</i>	180
Gambar 7.18. Contoh <i>Pipeline</i> Instruksi 80486	181
Gambar 8.1. State Diagram Siklus Instruksi	184
Gambar 8.2. Format isntruksi 16 bit sederhana	184
Gambar 8.3.Operasi <i>Shift</i> dan <i>Rotate</i>	195
Gambar 8.4. Instruksi Percabangan	197
Gambar 8.5. Prosedur <i>Nested</i>	199
Gambar 8.6. Penggunaan <i>Stack</i> pada implementasi Nestet Subroutine dari Gambar 8.5.....	200
Gambar 8.7. Penambahan isi Stack dengan prosedur sederhana P dan Q	200
Gambar 8.8. Mode Pengalamatan	201
Gambar 8.9. Format Instruksi PDP-8	209
Gambar 8.10. Format Instruksi x86	209
Gambar 9.1. Proses Pembuatan Executeble File	214
Gambar 9.2. Struktur code Assembly Language	215
Gambar 9.3. Penamaan Register pada Intel X86	216
Gambar 10.1. Overlapping Register Windows.....	226
Gambar 11.1. Elemen Penyusun Eksekusi Program	227
Gambar 11.2. Urutan Kejadian Fetch cycle.....	228

Gambar 11.3. <i>Flowchart</i> pada siklus insiktensi	231
Gambar 11.4. Blok Diagram <i>Control Unit</i>	233
Gambar 11.5. Jalur Data dan Sinyal Kontrol	234
Gambar 11.6. Prosesor dengan Internal Bus	236
Gambar 11.7. Blok Diagram Prosesor Intel 8085.....	237
Gambar 11.8. Pin pada IC Mikroprosesor Intel 8085.....	239

DAFTAR TABEL

Tabel 1.1. Set instruksi IAS	15
Tabel 1.2. Generasi Komputer Berdasarkan Teknologi yang digunakan	16
Tabel 1.3. Evolusi Prosesor Keluarga Intel	22
Tabel 2.1. Klasifikasi Karakteristik Memori Sistem Komputer	40
Tabel 2.2. Unit dari Kapasitas Memori.....	41
Tabel 2.3. Klasifikasi Karakteristik Cache	47
Tabel 2.4. Ukuran <i>Cache</i> untuk Beberapa Sistem Komputer.....	48
Tabel 2.5. Seri EPROM 27XXX	62
Tabel 2.6. Type Memori Semikonduktor	63
Tabel 2.7. Penambahan lebar word dengan <i>Error Correction</i>	67
Tabel 3.1. Karakteristik Fisik dari Disk System	72
Tabel 3.2. Level RAID.....	77
Tabel 3.3. Kelebihan dan Kekurangan RAID 0 sampai 6	82
Tabel 3.4. Perbandingan SSD dan Disk Drive	84
Tabel 4.1. Teknik I/O	92
Tabel 4.2. Register pada Intel DMAC 8237A.....	108
Tabel 5.1. Elemen-elemen Desain Bus.....	117
Tabel 5.2. Sinyal Wajib (Mandatory) PCI	126
Tabel 5.3. Sinyal Optional pada PCI	128
Tabel 5.4. Interpretasi Read Commands pada PCI.....	129
Tabel 5.5. Type Transaksi PCIe TLP	137
Tabel 5.6. Kecepatan Data beberapa versi USB.....	140
Tabel 6.1. Karakteristik Representasi Komplemen-2 dan Aritmatika	144
Tabel 6.2. Representasi Integer 4-Bit.....	145
Tabel 6.3. Parameter Format IEEE 754	159
Tabel 6.4. Format IEEE 754-2008	160
Tabel 6.5 Interpretasi Bilangan Floating Point dari IEEE.....	161
Tabel 7.1. Organisasi Register Keluarga Intel x86	170
Tabel 7.2. Tabel <i>Vector Interrupt and Exception</i> keluarga x86	175
Tabel 8.1. Program untuk mengeksekusi $Y=(A-B)/(C+DE)$	186
Tabel 8.2. Pemanfaatan Alamat Instruksi Bukan Percabangan	187
Tabel 8.3. Karakter ASCII.....	189
Tabel 8.4. Karakter Extended ASCII.....	190
Tabel 8.5. Type Operasi Dasar	191
Tabel 8.6. Contoh Operasi Transfer Data dari IBM EAS/390	193
Tabel 8.7. Tabel Kebenaran Logika Dasar	194
Tabel 8.8. Contoh dari Operasi <i>Shift</i> dan <i>Rotate</i>	196
Tabel 8.9. Mode Pengalamatan Dasar	202
Tabel 9.1. Directive pada NASM <i>Assembly-Language</i>	218
Tabel 10.1.Beberapa Karakteristik Prosesor CISC, RISC, dan Superscalar	222
Tabel 10.2. <i>Weighted Relative Dynamic Frequency</i> dari Karakteristik Operasi HLL	223
Tabel 10.3. Persentase Dinamik dari <i>Operand</i>	224
Tabel 10.4. <i>Procedure Argument</i> dan <i>Local Scalar Variabel</i>	224

Tabel 11.1. Micro Operation dan control Singnal.....	234
Tabel 11.2. Sinyal Eksternal pada Intel 8085	240

1. Pengenalan dan Evolusi Komputer

1.1. Pengenalan

A. Arsitektur dan Organisasi

Terdapat perbedaan deskripsi antara arsitektur komputer dan organisasi komputer. Arsitektur komputer adalah atribut-atribut sistem yang nampak bagi programmer, sehingga atribut-atribut tersebut memiliki dampak langsung pada eksekusi sebuah logika program. Istilah arsitektur komputer sering disamakan dengan istilah *Instruction Set Architecture* (ISA). ISA mendefinisikan format instruksi, kode operasi (*operation code - opcode*), register, dan memori data, efek dari instruksi yang dieksekusi pada register dan memori, serta algoritma yang digunakan untuk mengendalikan eksekusi intruksi. Contoh atribut arsitektural adalah: set instruksi, jumlah bit yang digunakan untuk merepresentasikan berbagai tipe data (misalnya: numerik, karakter), mekanisme I/O, dan teknik untuk mengatasi memori.

Organisasi komputer mengacu pada unit operasional dan interkoneksi yang mewujudkan spesifikasi arsitektur. Contoh atribut organisasional adalah: detil perangkat keras yang transparan bagi programmer (misalnya: sinyal kontrol, *interface*), dan teknologi memori yang digunakan

B. Struktur dan Fungsi

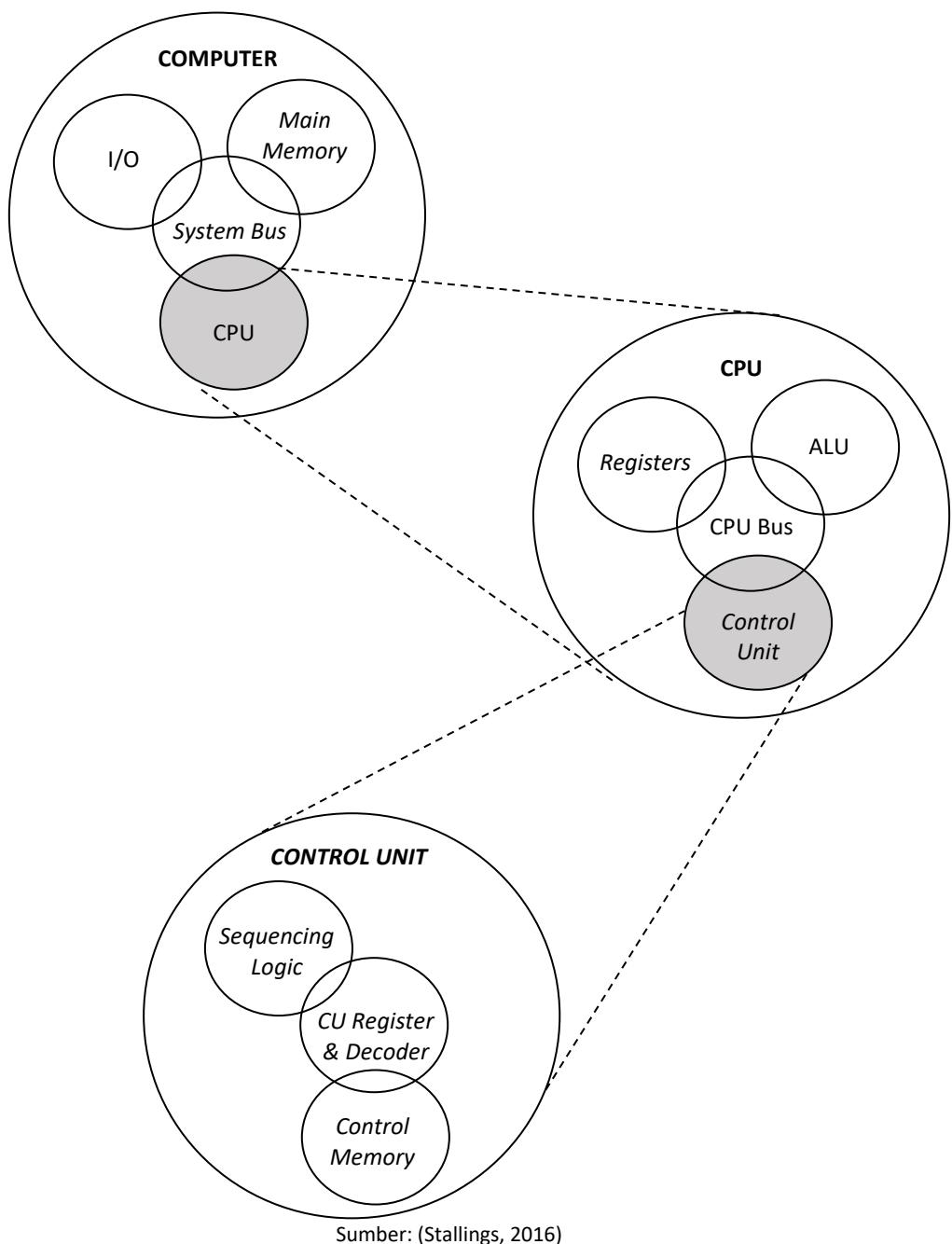
Komputer adalah sebuah sistem yang sangat kompleks, yang terdiri dari hardware softare dan brainware. Pada dewasa ini, sebuah mesin komputer mengandung jutaan komponen elektronik dasar. Untuk menggambarkan sebuah sistem komputer, maka diawali dengan menganalisis sifat hirarkisnya. Sistem hirarkis adalah sekumpulan subsistem yang saling terkait, dimana masing-masing subsistem tersusun dalam struktur, mulai dari tingkatan tertinggi sampai terendah. Pada setiap tingkat, sistem terdiri dari serangkaian komponen dan keterkaitannya yang dideskripsikan dalam struktur dan fungsi. Struktur adalah bagaimana cara komponen saling terkait, sedangkan fungsi adalah bagaimana pengoperasian setiap komponen individu sebagai bagian dari struktur.

Pada dasarnya struktur dan fungsi komputer terbentuk dari empat fungsi dasar, yaitu:

1. **Data processing.** Data dapat direpresentasikan dalam berbagai bentuk dan cara pemrosesan. Namun secara umum hanya ada beberapa metode dari atau jenis pemrosesan data yang digunakan.
2. **Data storage.** Meskipun komputer dapat memproses data dengan cepat (yaitu, data yang masuk dan diproses, dan hasilnya segera keluar), namun komputer harus menyimpan sementara bagian data yang sedang dikerjakan di suatu tempat. Dengan demikian, setidaknya ada fungsi penyimpanan data jangka pendek. Sama pentingnya, komputer melakukan fungsi penyimpanan data jangka panjang. *File* data disimpan di komputer untuk pengambilan dan pembaruan selanjutnya.
3. **Data movement.** Pada saat komputer bekerja, terdapat perangkat yang berfungsi sebagai sumber atau tujuan data. Proses dimana data diterima dari atau dikirim ke perangkat

tersebut, dikenal sebagai *input–output* (I/O), dan perangkat juga sisebut sebagai periferal. Ketika data dipindahkan dari jarak yang lebih jauh, ke atau dari perangkat jarak jauh, proses ini dikenal sebagai komunikasi data.

4. **Control.** Di dalam komputer, *Control Unit* mengelola sumber daya komputer dan mengatur kinerja bagian fungsionalnya dalam menanggapi instruksi.



Gambar 1.1. Skema Struktur Komputer

Gambar 1.1 menunjukkan skema hirarki struktur internal komputer dengan prosesor tunggal. Komputer tradisional dengan prosesor tunggal, memiliki empat komponen utama pada strukturnya, yaitu:

1. **Central Processing Unit** (CPU). Mengendalikan operasi komputer dan membentuk fungsi pemrosesan datanya. CPU sering disebut sebagai prosesor.
2. **Main Memory**: Adalah memori kerja yang digunakan untuk menyimpan data.
3. **I/O**. Pemindahan data antara komputer dengan perangkat luar
4. **System Interconnection**. Mekanisme yang digunakan untuk komunikasi antara CPU, memori utama, dan I/O. Contoh umum interkoneksi sistem adalah melalui bus sistem, yang terdiri dari sejumlah kabel konduktor yang terhubung dengan semua komponen lainnya

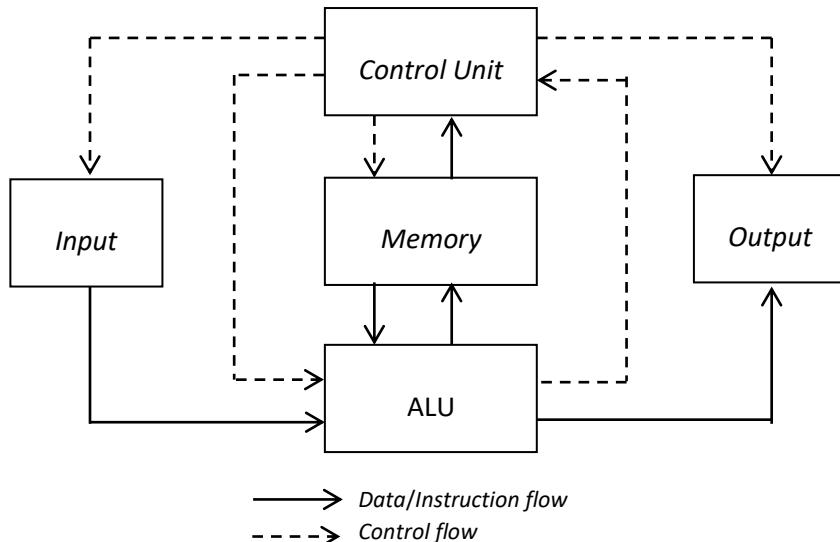
Dari keempat komponen utama tersebut, CPU adalah komponen yang paling kompleks. Struktur utama dari CPU adalah:

1. **Control Unit** (CU). Mengendalikan operasi CPU dan komputer secara keseluruhan.
2. **Arithmetic and Logic Unit** (ALU). Melakukan fungsi pemrosesan aritmatika dan logika.
3. **Register**. Menyediakan penyimpanan internal pada CPU.
4. **CPU Interconnection**. Beberapa mekanisme yang menyediakan komunikasi antara CU, ALU, dan register.

C. Arsitektur von Neumann

Pada desain awal, Sistem komputer digital menjalankan program pada satu CPU (prosesor). Hal ini masih berlaku untuk banyak sistem saat ini, terutama yang mementingkan biaya daripada kinerja komputasi yang tinggi. Meskipun banyak peningkatan telah dilakukan selama bertahun-tahun untuk ide aslinya, hampir setiap prosesor yang tersedia saat ini adalah turunan dari arsitektur von Neumann.

Arsitektur von Neumann dikembangkan untuk EDVAC dan EDSAC pada tahun 1940-an. Jadi, untuk memahami pengoperasian arsitektur sekarang dan masa depan, pertama-tama penting untuk melihat ke belakang dan mempertimbangkan karakteristik dari sistem komputasi modern praktis yang pertama, yaitu mesin von Neumann.

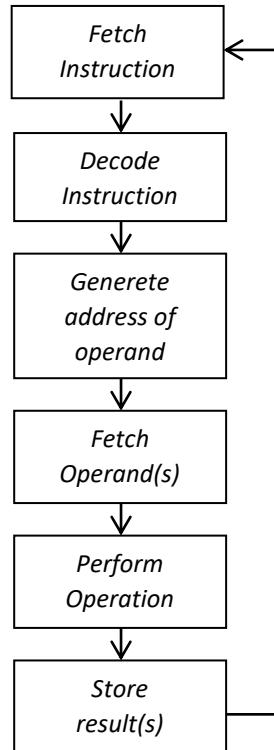


Sumber: (Stallings, 2016)

Gambar 1.2. Skema Arsitektur Komputer von Neumann

Arsitektur von Neumann, juga dikenal sebagai Arsitektur Princeton (karena John von Neumann adalah peneliti di Institut Studi Lanjutan Universitas Princeton) adalah desain modern pertama untuk komputer berdasarkan konsep program tersimpan (*stored program*) yang awalnya dikembangkan oleh Charles Babbage. Mesin yang diimpikan oleh tim von Neumann sangat sederhana menurut standar saat ini, tetapi fitur utamanya mudah dikenali oleh siapapun yang telah mempelajari organisasi dasar sebagian besar sistem komputer modern. Diagram blok dari komputer von Neumann

yang ditunjukkan pada Gambar 1.2 dengan jelas menunjukkan perangkat input dan output serta memori tunggal yang digunakan untuk menyimpan data dan instruksi program. Unit kontrol dan unit aritmatika/logika (ALU) dari mesin von Neumann adalah bagian penting dari CPU di mikroprosesor modern (register internal ditambahkan kemudian untuk menyediakan penyimpanan yang lebih cepat untuk jumlah data yang terbatas).



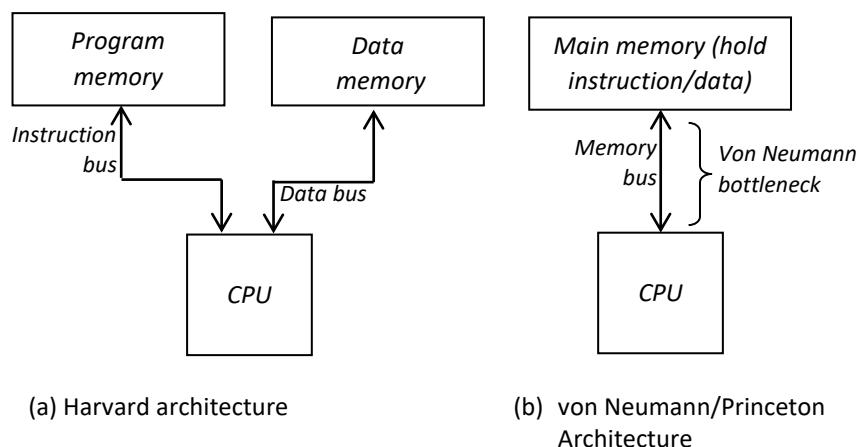
Sumber: (Stallings, 2016)

Gambar 1.3. Siklus Instruksi Komputer von Neumann

Faktor utama yang membedakan arsitektur von Neumann dari mesin sebelumnya adalah konsep program yang disimpan. Siklus mesin von Neumann diilustrasikan pada Gambar 1.3. Siklus ini adalah proses yang harus dilakukan untuk setiap instruksi dalam program komputer yang disimpan. Ketika semua langkah selesai untuk instruksi yang diberikan, CPU siap untuk memproses instruksi berikutnya. Siklus dimulai dengan CPU mengambil (membaca) instruksi dari memori. Instruksi tersebut kemudian diterjemahkan (*decode*), dimana perangkat keras di dalam CPU menafsirkan bit-bit pada instruksi dan menentukan: operasi apa yang perlu dilakukan; alamat (lokasi); dan data yang akan dioperasikan (*operand*). *Operand* yang dikirim ke ALU dapat bersumber dari register, lokasi memori, atau perangkat input. Akhirnya, hasil disimpan di lokasi yang ditentukan (atau dikirim ke perangkat keluaran), dan prosesor siap untuk mulai menjalankan instruksi berikutnya.

Arsitektur Harvard adalah organisasi komputer alternatif yang dikembangkan oleh Howard Aiken di Universitas Harvard dan digunakan di mesin Mark-I dan Mark-II. Ini bertujuan untuk menghindari "von Neumann bottleneck" yang terjadi pada jalur data tunggal ke memori untuk mengakses instruksi dan data, dengan menyediakan memori dan bus terpisah untuk instruksi dan data (Gambar 1.4), dengan demikian instruksi dapat diambil saat data sedang dibaca atau ditulis. Ini adalah salah satu dari banyak contoh di bidang desain sistem komputer di mana peningkatan biaya implementasi dan kompleksitas dapat dibenarkan untuk mendapatkan peningkatan kinerja yang sesuai.

Banyak sistem modern menggunakan jenis struktur ini, meskipun sangat jarang melihat mesin Harvard yang "benar" (dengan memori yang benar-benar terpisah untuk kode dan data). Sebaliknya, dalam sistem komputasi saat ini, adalah umum untuk menggunakan arsitektur Harvard yang dimodifikasi di mana memori utama "disatukan" (berisi kode dan data seperti mesin yang didasarkan pada arsitektur Princeton), tetapi memori cache terpisah disediakan untuk instruksi dan data. Model ini dapat mencapai keunggulan kinerja yang hampir sama dengan arsitektur Harvard asli, tetapi menyederhanakan desain dan pengoperasian sistem secara keseluruhan. Ada kalanya program itu sendiri harus dimanipulasi sebagai data, misalnya, saat menyusun atau merakit kode; ini akan jauh lebih rumit dengan memori yang benar-benar terpisah. Dengan penekanan pada kecepatan prosesor dan penggunaan memori cache untuk menjembatani kesenjangan kecepatan CPU dan memori, arsitektur Harvard yang telah dimodifikasi telah menjadi sangat banyak digunakan. Ini sangat sesuai untuk arsitektur modern (*pipelined RISC*).



Sumber: (Stallings, 2016)

Gambar 1.4. Perbandingan arsitektur komputer Harvard dan von Neumann/Princeton

D. Struktur Komputer Multicore

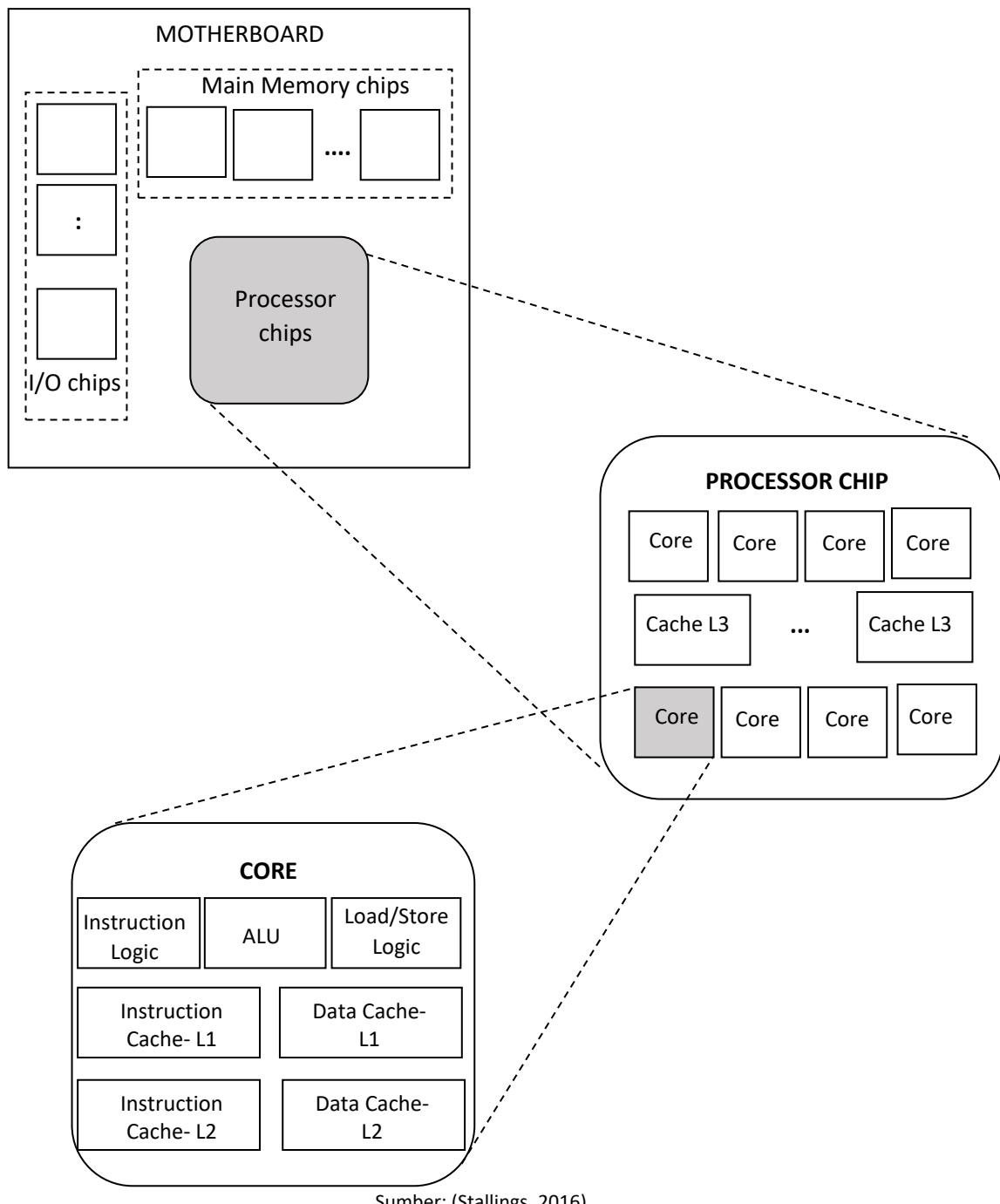
Komputer modern pada umumnya memiliki banyak prosesor. Ketika semua prosesor ini berada pada satu *chip*, maka disebut sebagai komputer *multicore*, dan setiap unit pemrosesan atau *Central Processing Unit* (terdiri dari *Control Unit*, *ALU*, *register*, dan mungkin *cache*) disebut *core*.

Core, adalah pemrosesan individual pada *chip* prosesor. *Core* fungsinya setara dengan prosesor pada komputer dengan unit pemrosesan tunggal. Unit pemrosesan khusus lainnya, seperti yang dioptimalkan untuk operasi vektor dan matriks, juga disebut sebagai *core*. Prosesor adalah komponen komputer yang menginterpretasikan dan menjalankan instruksi, terbuat dari sepotong fisik silikon yang mengandung satu atau lebih *core*. Jika sebuah prosesor mengandung banyak *core*, itu disebut sebagai prosesor *multicore*.

Ciri lain dari komputer modern adalah penggunaan *cache memory*, yaitu memori berlapis-lapis antara processor dan *main memory*. Salah satu tujuan menggunakan *cache memory* adalah untuk mempercepat akses memori, sehingga *cache memory* harus memiliki kapasitas lebih kecil dan lebih cepat dari *main memory*. Peningkatan kinerja yang lebih tinggi dapat diperoleh dengan menggunakan beberapa level *cache*, dengan level 1 (L1) paling dekat dengan *core*, dan level tambahan (L2, L3, dan seterusnya) semakin jauh dari *core*. Dalam skema ini, level n lebih kecil dan lebih cepat dari level n+1.

Gambar 1.5 adalah gambaran umum komponen utama komputer *multicore* yang disederhanakan. Komponen utama tersebut pada sebagian besar komputer ditempatkan pada sebuah *motherboard*. *Motherboard* berisi slot atau soket untuk *chip* prosesor, yang biasanya berisi beberapa *core*, yang

dikenal sebagai prosesor *multicore*. Ada juga slot untuk *chip* memori, *chip* pengontrol I/O, dan komponen komputer utama lainnya.



Gambar 1.5. Skema Sebuah Komputer Multicore

Chip prosesor pada gambar 1.5 berisi delapan *core* dan *cache* L3. Angka tersebut menunjukkan bahwa *cache* L3 menempati dua bagian berbeda dari permukaan *chip*. Namun, biasanya, semua *core* memiliki akses ke seluruh *cache* L3.

Secara umum, sebuah *core* memiliki elemen fungsional sebagai berikut:

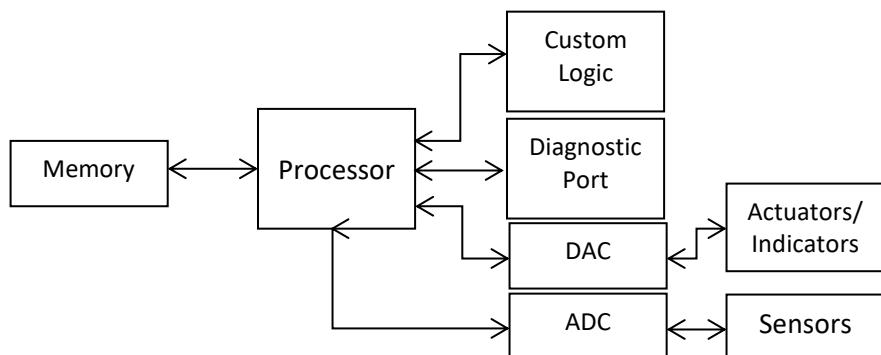
- 1) *Instruction Logic*. Bagian Ini berfungsi untuk mengambil instruksi, dan mendekode setiap instruksi yang diambilnya tersebut untuk menentukan operasi yang diinginkan dan lokasi memori dari setiap *operand*
- 2) *Arithmatic and Logic Unit (ALU)*. Bagian ini akan melakukan operasi yang ditentukan oleh instruksi.
- 3) *Load/Storage Logic*. Bagian ini mengelola transfer data ke dan dari memori utama melalui *cache*.

Cache pada *core* juga dapat berisi *cache* L1, meliputi *cache* instruksi (L1 - *Instruction cache*) yang digunakan untuk transfer instruksi ke dan dari memori utama, dan *cache* data (L1 - *Data cache*) untuk transfer *operand* dan hasil operasi. Biasanya, *chip* prosesor juga menyertakan *cache* L2 sebagai bagian dari *core*. Dalam banyak kasus, *cache* L2 ini juga dibagi antara *cache* instruksi dan data, meskipun *cache* L2 tunggal gabungan juga bisa digunakan.

E. Embedded System

Istilah *embedded system* mengacu pada penggunaan elektronik dan perangkat lunak dalam suatu produk, berbeda dari komputer tujuan umum, seperti laptop atau sistem desktop. Jutaan komputer dijual setiap tahun, termasuk laptop, komputer pribadi (*Personal Computer*), *workstation*, server, *mainframe*, dan superkomputer. Sementara, milyaran sistem komputer diproduksi setiap tahun yang tertanam di dalam perangkat yang lebih besar. *System embedded computer* dirancang untuk menjalankan satu aplikasi atau satu set aplikasi yang biasanya terintegrasi dengan perangkat keras dan dikirimkan sebagai sistem tunggal.

Jenis perangkat dengan sistem tertanam terlalu banyak untuk disebutkan, namun contoh yang paling umum adalah ponsel, kamera digital, kamera video, kalkulator, oven microwave, sistem keamanan rumah, mesin cuci, sistem pencahayaan, termostat, printer, berbagai sistem otomotif (misalnya, kontrol transmisi, kontrol jelajah, injeksi bahan bakar, rem *anti-lock*, dan sistem suspensi), raket tenis, sikat gigi, dan berbagai jenis sensor dan aktuator dalam sistem otomatis.



Sumber: (Stallings, 2016)

Gambar 1.6. Contoh Organisasi *Embedded System*

Gambar 1.6 menunjukkan secara umum organisasi *embedded System*. Selain prosesor dan memori, ada sejumlah elemen yang berbeda dari komputer desktop atau laptop, yaitu:

- 1) Ada berbagai antarmuka yang memungkinkan sistem untuk mengukur, memanipulasi, dan berinteraksi dengan lingkungan eksternal. Sistem tertanam sering berinteraksi (merasakan, memanipulasi, dan berkomunikasi) dengan dunia luar melalui sensor dan actuator, sehingga dapat disebut sebagai sistem reaktif. Sistem reaktif melakukan interaksi terus-menerus dengan lingkungan dan dijalankan pada kecepatan yang ditentukan oleh lingkungan itu.

- 2) Antarmuka manusia dapat sesederhana lampu yang berkedip atau serumit visi robot waktunya. Dalam banyak kasus, tidak ada antarmuka manusia.
- 3) *Port diagnostik* dapat digunakan untuk mendiagnosis sistem yang sedang dikontrol-bukan hanya untuk mendiagnosis komputer.
- 4) *Field-Purpose Programmable* (FPGA), *Aplication-Spesific* (ASIC), atau bahkan perangkat keras nondigital dapat digunakan untuk meningkatkan kinerja atau keandalan.
- 5) Perangkat lunak seringkali memiliki fungsi tetap dan khusus untuk aplikasi tersebut. Efisiensi sangat penting untuk *embedded system*.

Mereka dioptimalkan untuk energi, ukuran kode, waktu eksekusi, berat dan dimensi, serta biaya. Ada beberapa bidang penting yang mirip dengan sistem komputer untuk tujuan umum, yaitu:

- 1) Kemampuan untuk melakukan *upgrade* untuk memperbaiki *bug*, untuk meningkatkan keamanan, dan untuk menambah fungsionalitas, telah menjadi sangat penting untuk *embedded system*, dan tidak hanya di perangkat konsumen.
- 2) Salah satu perkembangan yang relatif baru adalah *platform* sistem tertanam yang mendukung beragam aplikasi, Contohnya adalah ponsel cerdas dan perangkat audio/visual, seperti smart TV.

F. *Internet of Things*

Internet of things (IoT) adalah istilah yang mengacu pada perluasan interkoneksi perangkat pintar, mulai dari *devices* hingga sensor kecil. Tema yang dominan adalah penanaman transceiver seluler jarak pendek ke dalam beragam gadget dan barang sehari-hari, memungkinkan bentuk komunikasi baru antara manusia dan benda, dan antara benda itu sendiri. Internet sekarang mendukung interkoneksi milyaran benda industri dan pribadi, biasanya melalui *cloud system*. Objek mengirimkan informasi dari sensor yang bertindak pada lingkungan mereka, dan dalam beberapa kasus, memodifikasi diri mereka sendiri, untuk menciptakan manajemen keseluruhan sistem yang lebih besar, seperti pabrik atau kota.

IoT terutama didorong oleh *Deeply Embedded Devices*, yaitu perangkat khusus yang bertujuan untuk mendeteksi sesuatu di lingkungan, melakukan pemrosesan tingkat dasar, dan kemudian melakukan sesuatu dengan hasilnya. *Deeply Embedded Devices* seringkali memiliki kemampuan nirkabel dan muncul dalam konfigurasi jaringan, seperti jaringan sensor yang digunakan di area yang luas (misalnya: Pabrik dan bidang pertanian). IoT sangat bergantung pada sistem yang *Deeply Embedded Devices*, biasanya memiliki kendala sumber daya ekstrem dalam hal memori, ukuran prosesor, waktu, dan konsumsi daya.

IoT biasanya memiliki *bandwidth* rendah, pengambilan data pengulangan rendah, dan peralatan penggunaan data *bandwidth* rendah yang saling berkomunikasi dan menyediakan data melalui *user interface*. Peralatan tertanam, seperti kamera keamanan video beresolusi tinggi, telepon VoIP video, dan beberapa lainnya, memerlukan kemampuan streaming bandwidth tinggi. Namun produk yang tak terhitung jumlahnya hanya membutuhkan paket data untuk dikirimkan sesekali.

Dengan mengacu pada sistem akhir yang didukung, Internet telah melalui sekitar empat generasi penyebaran yang berujung pada IoT, yaitu:

- 1) ***Information Technology*** (TI): PC, server, router, firewall, dan sebagainya, dibeli sebagai perangkat TI oleh orang-orang TI perusahaan dan terutama menggunakan koneksi kabel.
- 2) ***Operational Technology*** (OT): Mesin/peralatan dengan IT tertanam yang dibangun oleh perusahaan non-IT, seperti mesin medis, SCADA (Supervisory Control And Data Acquisition), kontrol proses, dan kiosks, dibeli sebagai peralatan oleh orang-orang IT perusahaan dan terutama menggunakan koneksi kabel.

- 3) **Personal Technology:** Ponsel pintar, tablet, dan pembaca e-book yang dibeli sebagai perangkat TI oleh konsumen (karyawan) secara eksklusif menggunakan koneksi nirkabel dan seringkali berbagai bentuk koneksi nirkabel.
- 4) **Teknologi sensor/aktuator:** *Single-purpose devices* yang dibeli oleh konsumen, TI, dan karyawan OT secara eksklusif menggunakan koneksi nirkabel, umumnya dalam bentuk tunggal, sebagai bagian dari sistem yang lebih besar. Ini adalah generasi keempat yang biasanya dianggap sebagai IoT, dan ditandai dengan penggunaan miliaran perangkat tertanam.

G. Mikroprosesor versus Mikrokontroler

Mikroprosesor adalah sebuah *chip* yang di dalamnya terkandung rangkaian **ALU** (*Arithmetic-Logic Unit*), **CU** (*Control Unit*) dan kumpulan register-register. Sebuah Mikroprosesor tidak dapat menjalankan sebuah sistem tanpa dilengkapi dengan komponen lainnya seperti RAM, ROM, *Clock*, *Serial Com Port*. Dengan kata lain, mikroprosesor tidak dapat berdiri sendiri, namun harus didukung komponen lain.

Mikrokontroler adalah sebuah *chip* yang didalamnya terdapat sebuah mikroprosesor yang telah dilengkapi dengan RAM, ROM, I/O Port, Timer dan Serial COM dalam satu paket. Dengan kata lain, pada sistem mikrokontroler tidak perlu lagi menambahkan komponen penunjang untuk bekerja. Biasanya sebuah mikrokontroler tinggal disambungkan ke sumber daya dan ditambah sebuah *Clock* eksternal, namun bisa juga menggunakan *Clock* internal dari IC.

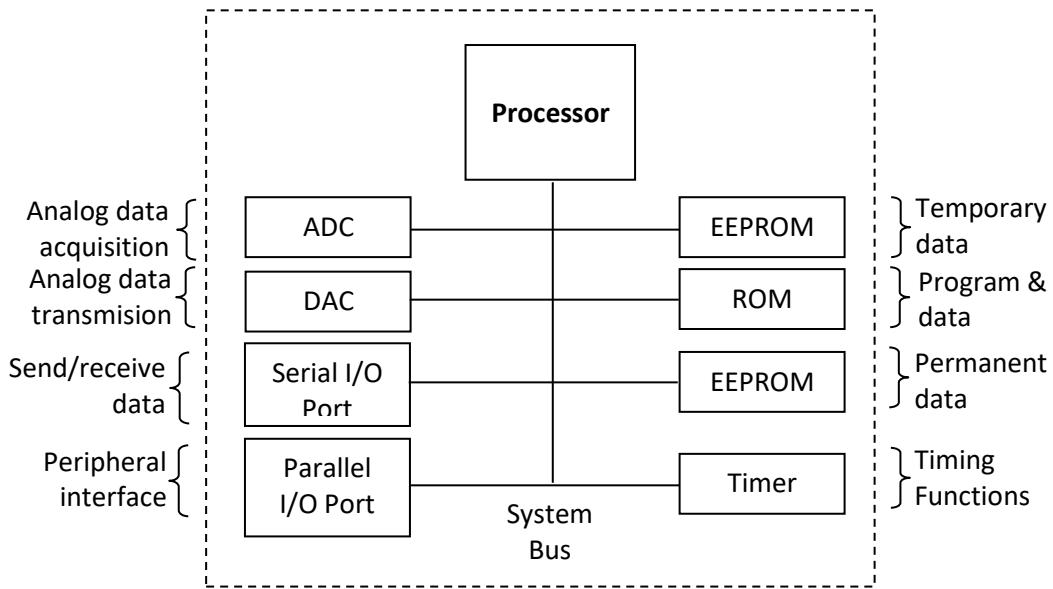
Prosesor mikrokontroler dapat didasarkan pada komputasi *Complex Instruction Set Computing* (CISC) atau *Reduced Instruction Set Computing* (RISC). CISC umumnya memiliki sekitar 80 instruksi pada set instruksinya, sementara RISC memiliki sekitar 30. CISC memiliki lebih banyak mode pengalaman yaitu 12-24 lebih banyak dibandingkan dengan RISC yang hanya 3-5. CISC dapat lebih mudah diimplementasikan dan memiliki penggunaan memori yang lebih efisien, tapi akan mengalami penurunan kinerja karena semakin banyak jumlah siklus *clock* yang diperlukan untuk menjalankan instruksi. RISC yang lebih menekankan pada perangkat lunak, seringkali memberikan kinerja yang lebih baik daripada prosesor CISC, yang lebih menekankan pada perangkat keras, karena set instruksi yang disederhanakan. RISC yang penekanannya lebih pada perangkat lunak, menyebabkan memiliki perangkat lunak menjadi lebih kompleks. Pilihan arsitektur mana yang digunakan, tergantung pada aplikasi yang akan diterapkan.

Arsitektur bus pada mikrokontroler dapat didasarkan pada arsitektur Harvard atau von Neumann. Kedua arsitektur tersebut memiliki teknik tertentu pada metode pertukaran data antara prosesor dan memori. Pada arsitektur Harvard, bus data dan alamat terpisah, memungkinkan untuk transfer data dan alamat secara simultan. Pada arsitektur Von Neumann, satu bus digunakan untuk alamat dan data.

Gambar 1.7 menunjukkan secara umum elemen-elemen yang biasanya ditemukan pada *chip* mikrokontroler. Mikrokontroler adalah *chip* tunggal yang berisi prosesor, memori *non-volatile* untuk program (ROM), memori *volatile* (RAM) untuk *input* dan *output*, *clock*, dan unit kontrol I/O. Bagian prosesor dari mikrokontroler memiliki area silikon yang jauh lebih rendah daripada mikroprosesor lainnya dan efisiensi energi yang jauh lebih tinggi.

Mikrokontroler Juga disebut "komputer pada *chip*," miliaran unit mikrokontroler tertanam setiap tahun dalam berbagai produk dari mainan hingga peralatan hingga mobil. Misalnya, satu kendaraan dapat menggunakan 70 atau lebih mikrokontroler. Biasanya, terutama untuk mikrokontroler yang lebih kecil, lebih murah, mereka digunakan sebagai prosesor khusus untuk tugas-tugas tertentu. Sebagai contoh, mikrokontroler banyak digunakan dalam proses otomasi. Dengan memberikan reaksi sederhana terhadap *input*, mikrokontroler dapat mengontrol mesin, menghidupkan dan mematikan kipas, membuka dan menutup katup, dan sebagainya. Mikrokontroler adalah bagian integral dari

teknologi industri modern dan merupakan salah satu cara paling murah untuk menghasilkan mesin yang dapat menangani fungsi yang sangat kompleks.



Sumber: (Stallings, 2016)

Gambar 1.7. Skema Dasar *Chip* Mikrokontroler

Mikrokontroler hadir dalam berbagai ukuran fisik dan kekuatan pemrosesan. Prosesor pada mikrokontroler berkisar dari arsitektur 4-bit hingga 32-bit. Mikrokontroler cenderung jauh lebih lambat daripada mikroprosesor, biasanya beroperasi dalam rentang MHz dari pada kecepatan mikroprosesor GHz. Fitur khas lain dari mikrokontroler adalah bahwa ia tidak menyediakan interaksi manusia. Mikrokontroler diprogram untuk tugas tertentu, tertanam di perangkatnya, dan dijalankan sesuai kebutuhan dan bila diperlukan.

Ada 4 keluarga mikrokontroler yang paling banyak dijumpai, diantaranya adalah:

- 1) **Keluarga AVR.** Mikrokontroler keluarga AVR adalah mikrokontroler yang dikembangkan oleh Perusahaan Atmel. Istilah AVR ada yang mengartikan sebagai Advanced Virtual Risc, ada pula yang mengartikan sebagai Alv-Vegard Risc. Mikrokontroler AVR memiliki beberapa tipe diantaranya AVR AT *Classic*, AVR AT *Tiny*, AVR AT *Mega*, dan AVR *Special Purpose*.
- 1) **Keluarga MCS 51.** MCS 51 ini termasuk dalam keluarga mikrokontroler CISC (*Complex Instruction Set Computer*). Mikrokontroler MCS 51 merupakan mikrokontroller 8 bit yang diproduksi oleh Perusahaan Atmel dan Intel.
- 2) **Keluarga PIC (*Programmabel Intelligent Computer*).** PIC termasuk keluarga mikrokontroler berarsitektur Harvard, yaitu arsitektur yang memisahkan bus data dengan bus alamat. PIC dibuat oleh Microchip Technology, awalnya dikembangkan oleh Divisi Mikroelectronic General Instruments dengan nama PIC1640.
- 3) **Keluarga ARM.** ARM (*Advanced RISC Machine*) awalnya dikenal sebagai Mesin Acorn RISC. mikrokontroler ini memiliki arsitektur 32 bit, diimplementasikan pada Windows, Unix, dan sistem operasi mirip Unix, termasuk Apple iOS, Android, BSD, Inferno, Solaris, WebOS, Plan 9 dan GNU/Linux.

1.2. Evolusi Komputer

A. Generasi Pertama: Tabung Hampa

ENIAC

ENIAC (*Electronic Numerical Integrator And Computer*), dirancang dan dibangun di University of Pennsylvania, adalah komputer digital elektronik serbaguna pertama di dunia. Proyek tersebut merupakan tanggapan atas kebutuhan AS selama Perang Dunia II. Laboratorium Penelitian Army's Ballistics Research Laboratory (BRL), sebuah badan yang bertanggung jawab untuk mengembangkan *range and trajectory tables* untuk senjata baru, mengalami kesulitan untuk memasok tabel-tabel ini secara akurat dan dalam jangka waktu yang wajar. Tanpa *firing tables* ini, senjata dan artileri baru tidak akan berguna bagi penembak. BRL mempekerjakan lebih dari 200 orang, untuk memecahkan persamaan balistik yang diperlukan dengan menggunakan kalkulator desktop. Persiapan meja untuk satu senjata akan memakan waktu satu orang berjam-jam, bahkan berhari-hari.

John Mauchly, profesor teknik kelistrikan di Universitas Pennsylvania, dan John Eckert, salah satu mahasiswa pascasarjana, mengusulkan untuk membangun komputer serba guna menggunakan tabung hampa untuk aplikasi BRL. Pada tahun 1943, Angkatan Darat menerima proposal ini, dan pekerjaan ENIAC dimulai. Mesin yang dihasilkan sangat besar, dengan berat 30 ton, menempati luas lantai 1.500 kaki persegi, dan berisi lebih dari 18.000 tabung vakum. Saat beroperasi, itu mengkonsumsi daya 140 kilowatt. Itu juga jauh lebih cepat daripada komputer elektromekanis mana pun, yang mampu melakukan 5.000 penambahan per detik.

ENIAC lebih merupakan desimal daripada mesin biner. Artinya, angka direpresentasikan dalam bentuk desimal, dan aritmatika dilakukan dalam sistem desimal. Memori terdiri dari 20 akumulator, masing-masing mampu menampung 10 digit bilangan desimal. Sebuah cincin dari 10 tabung hampa mewakili setiap digit. Setiap saat, hanya satu tabung vakum yang berada dalam status ON, mewakili salah satu dari 10 digit. Kelemahan utama dari ENIAC adalah bahwa ENIAC harus diprogram secara manual dengan mengatur sakelar dan mencolokkan serta mencabut kabel.

ENIAC selesai pada tahun 1946, terlambat untuk digunakan dalam upaya perang. Sebaliknya, tugas pertamanya adalah melakukan serangkaian perhitungan kompleks yang digunakan untuk membantu menentukan kelayakan bom hidrogen. Penggunaan ENIAC untuk tujuan selain tujuan pembuatannya menunjukkan sifat tujuan umumnya. ENIAC terus beroperasi di bawah manajemen BRL hingga tahun 1955.

Mesin von Neumann

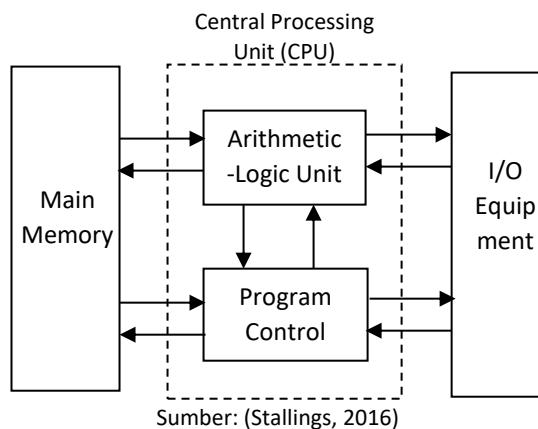
Tugas memasukkan dan mengubah program untuk ENIAC sangat melelahkan. Tetapi bila sebuah program dapat direpresentasikan dalam bentuk yang sesuai untuk disimpan dalam memori bersama dengan data. Kemudian, komputer bisa mendapatkan instruksinya dengan membacanya dari memori, dan sebuah program dapat diatur atau diubah dengan mengatur nilai-nilai sebagian dari memori. Ide ini dikenal sebagai konsep program tersimpan (*stored-program concept*), biasanya dikaitkan dengan desainer ENIAC, terutama matematikawan John von Neumann, yang merupakan konsultan pada proyek ENIAC. Alan Turing mengembangkan idenya pada waktu yang hampir bersamaan. Publikasi pertama dari gagasan tersebut ada dalam proposal tahun 1945 oleh von Neumann untuk komputer baru, EDVAC (*Electronic Discrete Variable Computer*). Gambar 1.8 menunjukkan struktur umum komputer IAS yang terdiri dari:

- *Main Memory*, yang menyimpan data dan instruksi
- *Arithmetic-Logic Unit (ALU)* yang mampu beroperasi pada data biner

- Control Unit, yang menafsirkan instruksi dalam memori dan menyebabkannya dieksekusi
- Peralatan input/output (I/O) yang dioperasikan oleh unit control

Komputer generasi pertama menggunakan tabung vakum untuk elemen logika dan memori. Sejumlah penelitian, dan kemudian komputer komersial dibangun dengan menggunakan tabung vakum. Komputer IAS adalah model yang ideal untuk memahami struktur dan fungsi komputer generasi pertama. Pendekatan desain mendasar yang pertama kali diterapkan pada komputer IAS dikenal sebagai konsep *store-program*. Gagasan ini biasanya dikaitkan dengan ahli matematika John von Neumann. Pada waktu yang bersamaan, Alan Turing juga mengembangkan gagasan tersebut. Konsep *stored-program* pertama kali dipublikasi dalam proposal yang dibuat oleh von Neumann pada tahun 1945 untuk komputer baru, EDVAC (*Electronic Variable Computer Variable*). Pada tahun 1946, von Neumann dan rekan-rekannya memulai desain komputer *store-program* yang baru, yang kemudian disebut sebagai Komputer IAS, di Princeton Institute. Meskipun tidak selesai sampai tahun 1952, komputer IAS adalah prototipe dari semua *general purpose computer* berikutnya. Struktur umum komputer IAS, yang terdiri dari:

- **Main Memory**, yang menyimpan data dan instruksi.
- **Arithmetic and Logic Unit (ALU)**, mampu mengoperasikan data biner.
- **Control Unit (CU)**, yang menterjemahkan dan mengeksekusi instruksi dalam memori.
- **Input-Output (I/O)**, peralatan yang dioperasikan oleh *Control Unit*.



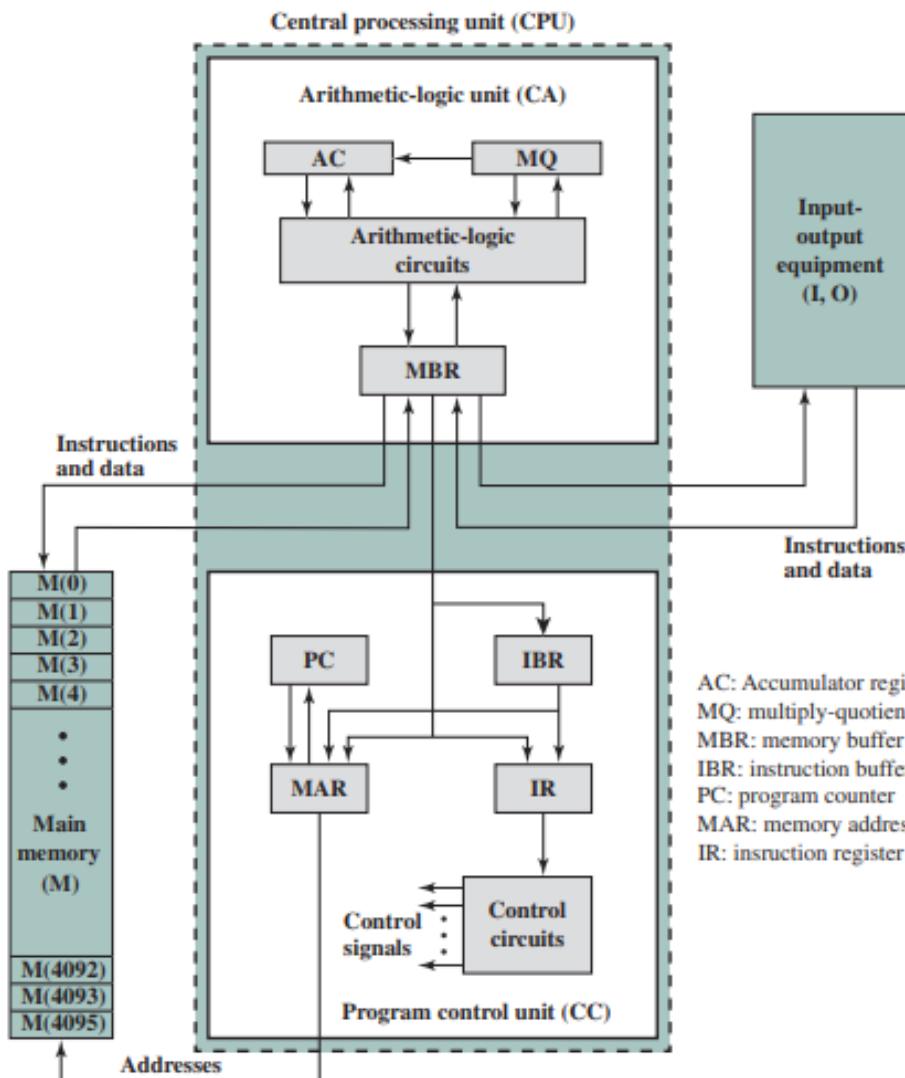
Gambar 1.8. Struktur Umum Komputer IAS

Pada umumnya semua komputer saat ini memiliki struktur dan fungsi umum yang sama dengan konsep komputer IAS, dan karenanya disebut sebagai mesin **von Neumann**. Memori pada komputer IAS terdiri dari 1.000 lokasi penyimpanan yang disebut *words*, masing-masing *word* terdiri dari 40 bit, dimana data dan instruksi tersimpan. Gambar 1.9 menunjukkan struktur komputer IAS dengan lebih rinci.

CU mengoperasikan IAS dengan mengambil instruksi dari memori, kemudian mengeksekusinya satu per satu. CU dan ALU berisi lokasi penyimpanan yang disebut register. Berikut adalah register yang terdapat pada komputer IAS:

- **Memory Buffer Register (MBR)**, berisi *word* 8 bit yang akan disimpan dalam memori atau dikirim ke unit I/O, serta digunakan untuk menerima *word* dari memori atau dari unit I/O.
- **Memory Address Register (MAR)**, register 12 bit yang menentukan alamat *word* dalam memori yang akan ditulis atau dibaca ke dalam MBR.
- **Instruction Register (IR)**: Berisi instruksi opcode 8-bit yang sedang dijalankan.
- **Instruction Buffer Register (IBR)**, register 40 bit digunakan untuk menyangga sementara dua buah instruksi yang baru diambil dari memori.

- **Program Counter (PC)**, register 20 bit yang berisi alamat instruksi berikutnya yang akan diambil dari memori.



Sumber: (Stallings, 2016)

Gambar 1.9. Struktur komputer IAS

Accumulator (AC) dan Multiplier Quotient (MQ), adalah register yang digunakan untuk menyangga *operand* sementara dan hasil operasi ALU. Sebagai contoh, hasil dari mengalikan dua angka 40-bit adalah angka 80-bit; 40 bit MSB disimpan di AC dan 40 bit LSB di MQ.

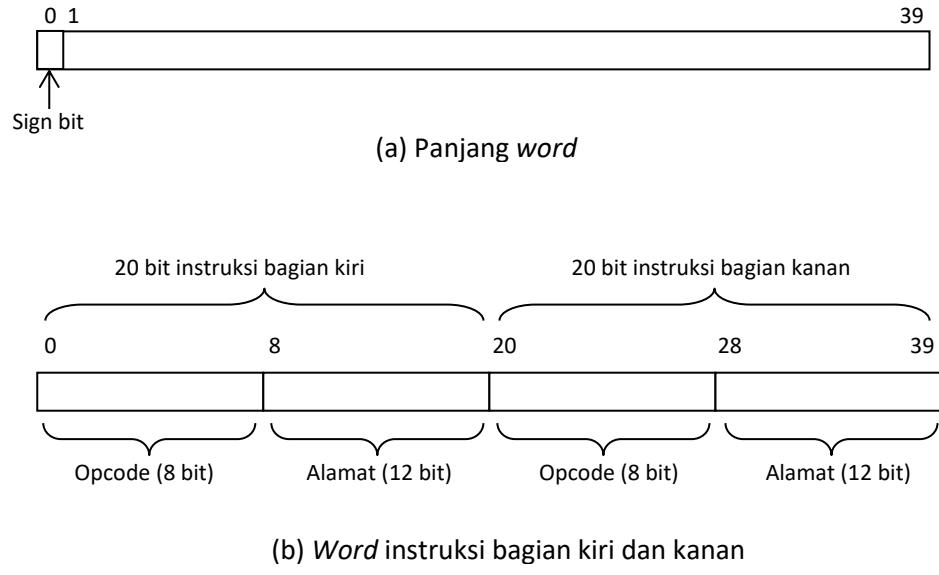
Komputer IAS beroperasi dengan melakukan siklus instruksi secara berulang, seperti yang ditunjukkan pada Gambar 1.11. Setiap siklus instruksi terdiri dari dua sub-siklus, fetch cycle dan execute cycle.

Selama siklus pengambilan (*fetch*), opcode dari instruksi selanjutnya dimuat ke IR dan bagian alamat dimuat ke dalam **MAR**. Instruksi ini dapat diambil dari **IBR**, atau dapat diperoleh dari memori dengan memuat *word* ke **MBR**, dan kemudian turun ke **IBR**, **IR**, dan **MAR**.

Setelah *opcode* berada di **IR**, siklus eksekusi (*execute*) dilakukan. Sirkuit kontrol mengartikan *opcode* dan menjalankan instruksi dengan mengirimkan sinyal kontrol yang sesuai untuk menyebabkan data dipindahkan atau operasi dilakukan oleh **ALU**.

Word pada komputer IAS berukuran 40 bit. Gambar 1.10 menunjukkan hubungan alamat dan jumlah *word* yang ada pada memori komputer IAS. Angka direpresentasikan dalam bentuk biner, dan setiap

instruksi adalah kode biner. Setiap bilangan diwakili oleh 40 bit, meliputi satu bit tanda dan 39-bit berikutnya menyatakan besaran bilangan tersebut. Suatu word juga dapat menyimpan dua instruksi 20-bit, dengan setiap instruksi terdiri dari kode operasi 8-bit (*opcode*) yang menentukan operasi yang akan dilakukan, dan alamat 12-bit yang menunjuk salah satu *word* dalam memori (diberi nomor dari 0 hingga 999).

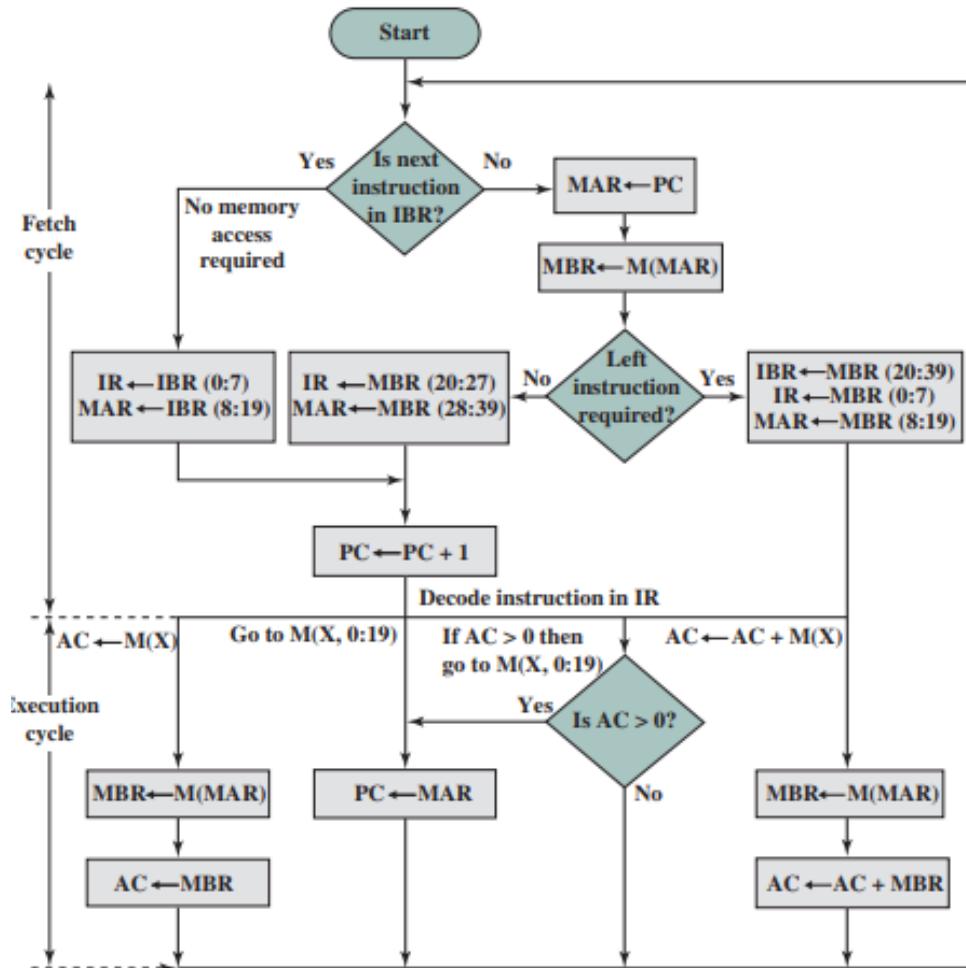


Sumber: (penulis, 2020)

Gambar 1.10. Format sebuah *Word* pada memori IAS

Komputer IAS memiliki total 21 instruksi, terangkum dalam Tabel 1.1. Set instruksi tersebut dapat dikelompokkan menjadi:

- **Data transfer:** instruksi yang digunakan untuk memindahkan data antara memori dan register ALU atau antara dua register **ALU**.
- **Unconditional branch** (Cabang tanpa syarat): adalah instruksi yang dapat mengubah urutan eksekusi instruksi yang biasa dilakukan oleh *Control Unit*. Urutan instruksi dapat diubah dengan instruksi cabang, yang memfasilitasi operasi berulang (*looping*).
- **Conditional branch** (Cabang bersyarat): Cabang dapat dibuat tergantung pada suatu kondisi, sehingga memungkinkan nilai keputusan.
- **Arithmatic:** Operasi yang dilakukan oleh **ALU**.
- **Address modify:** memanipulasi alamat dalam **ALU** dan kemudian dimasukkan ke dalam instruksi yang disimpan dalam memori. Ini memungkinkan suatu program menangani fleksibilitas yang cukup besar.



$M(X)$ = contents of memory location whose address is X
 $(i:j)$ = bits i through j

Sumber: (Stallings, 2016)

Gambar 1.11. Siklus Instruksi Komputer IAS

Tabel 1.1 menyajikan kumpulan instruksi komputer IAS (tidak termasuk instruksi I/O) dalam bentuk simbolis, disebut *mnemonic*. Penulisan instruksi dalam bentuk simbolik bertujuan agar mudah dibaca dan diingat. Dalam bentuk biner, setiap instruksi harus sesuai dengan format Gambar 1.10b. Bagian opcode (8 bit pertama) menentukan instruksi mana yang akan dieksekusi (dari 21 instruksi pada Tabel 1.1). Bagian alamat (sisa 12 bit) menentukan dari 1000 lokasi memori mana yang akan terlibat dalam pelaksanaan instruksi.

Tabel 1.1. Set instruksi IAS

Type Instruksi	Opcode	Simbolik	Deskripsi
Data transfer	00001010	LOAD MQ	Menyalin isi register MQ ke AC
	00001001	LOAD MQ, M(X)	Menyalin isi memori dengan alamat X ke MQ
	00100001	STOR M(X)	Menyalin isi AC ke memori dengan lokasi X
	00000001	LOAD M(X)	Menyalin isi memori dengan lokasi X ke AC
	00000010	LOAD -M(X)	Menyalin komplemen isi memori dengan lokasi X ke AC
	00000011	LOAD M(X)	Menyalin nilai absolut isi memori dengan lokasi X ke AC

	00000100	LOAD - M(X)	Menyalin komplemen dari nilai absolut isi memori dengan lokasi X ke AC
Unconditional branch	00001101	JUMP M(X, 0:19)	Eksekusi instruksi sebelah kiri M(X)
	00001110	JUMP M(X, 20:39)	Eksekusi instruksi sebelah kanan M(X)
Conditional branch	00001111	JUMP + M(X, 0:19)	Jika angka dalam akumulator tidak negatif, ambil yang berikutnya instruksi dari setengah kiri M(X)
	00010000	JUMP + M(X, 20:39)	Jika angka dalam akumulator tidak negatif, ambil yang berikutnya instruksi dari setengah kanan M(X)
Arithmetic	00000101	ADD M(X)	Tambahkan isi M(X) ke AC, hasilnya simpan di AC
	00000111	ADD M(X)	Tambahkan nilai mutlak dari isi M(X) ke AC, hasilnya simpan di AC
	00000110	SUB M(X)	kurangi isi M(X) dari AC, hasilnya simpan di AC
	00001000	SUB M(X)	kurangi nilai mutlak isi M(X) dari AC, hasilnya simpan di AC
	00001011	MUL M(X)	Kalikan M(X) dengan MQ; letakkan bit hasil paling signifikan di AC, masukkan bit paling tidak signifikan dalam MQ
	00001100	DIV M(X)	Bagilah AC dengan M(X); letakkan hasil bagi dalam MQ dan sisanya di AC
	00010100	LSH	Kalikan akumulator dengan 2; yaitu, bergeser ke kiri satu posisi bit
	00010101	RSH	Membagi akumulator dengan 2; yaitu, bergeser ke kanan satu posisi
Address modify	00010010	STOR M(X, 8:19)	Ganti bidang alamat kiri di M(X) dengan 12 bit paling kanan dari AC
	00010011	STOR M(X, 28:39)	Ganti bidang alamat kanan di M(X) dengan 12 bit paling kanan dari AC

Sumber: (Stallings, 2016)

B. Generasi Kedua: Transistor

Perubahan besar pertama dalam perkembangan komputer elektronik ditandai dengan penggantian tabung vakum oleh transistor. Dibandingkan dengan tabung vakum, transistor adalah komponen elektronika yang memiliki ukuran lebih kecil, lebih murah, dan menghasilkan lebih sedikit panas. Transistor dapat digunakan dengan cara yang sama seperti tabung vakum untuk membangun komputer. Berbeda dengan tabung vakum, yang membutuhkan kabel, pelat logam, kapsul kaca, dan ruang hampa, transistor adalah perangkat *solid-state* yang terbuat dari silikon. Transistor ditemukan di Bell Labs pada tahun 1947, kemudian pada 1950-an memicu terjadinya revolusi elektronik. Penggunaan transistor menjadi ciri khas komputer generasi kedua.

Komputer dapat diklasifikasikan berdasarkan teknologi dasar yang digunakan untuk membuat perangkat keras. Setiap generasi baru ditandai oleh kinerja pemrosesan yang lebih besar, kapasitas memori lebih besar, dan ukuran lebih kecil dari yang sebelumnya (Tabel 1.2).

Tabel 1.2. Generasi Komputer Berdasarkan Teknologi yang digunakan

Generasi	Perkiraan Tahun	Teknologi	Kecepatan (Operation per second)
1	1946-1957	Vacuum tube	40.000
2	1957-1964	Transistor	200.000
3	1965-1971	SSI	1.000.000
4	1972-1977	LSI	10.000.000
5	1978-1991	VLSI	100.000.000

6	1991-	ULSI	>1.000.000.000
Sumber: (Patterson & Hennessy, 2014)			

Selain mengganti fungsi tabung vakum dengan transistor, ada perubahan lain pada generasi kedua, yaitu: mulai mengenalkan **ALU** dan **CU** (*Control Unit*) yang lebih kompleks, penggunaan bahasa pemrograman tingkat tinggi, dan penyediaan perangkat lunak sistem dengan komputer. Secara umum, perangkat lunak sistem menyediakan kemampuan untuk memuat program, memindahkan data ke periferal, dan libraries untuk melakukan perhitungan umum, mirip dengan apa yang dilakukan sistem operasi modern, seperti Windows dan Linux.

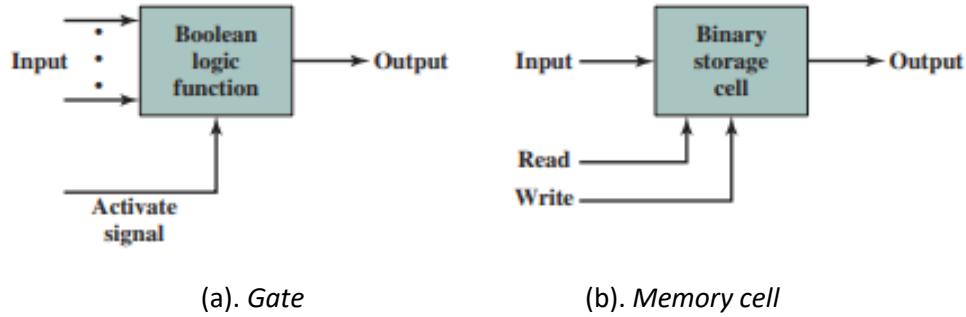
Salah satu contoh komputer generasi kedua adalah IBM 7094. Dari pengenalan seri 700 pada tahun 1952 hingga pengenalan anggota terakhir dari seri 7000 pada tahun 1964, basis produk komputer IBM ini mengalami evolusi yang cukup berarti. Anggota basis produk yang berturut-turut menunjukkan peningkatan kinerja, peningkatan kapasitas, dan/atau biaya yang lebih rendah. Ukuran memori utama, dalam kelipatan 2^{10} word 36-bit, tumbuh dari 2k ($1\text{k} = 2^{10}$) menjadi 32k word, sedangkan waktu untuk mengakses satu word memori (*memory cycle time*), turun dari 30 ms menjadi 1,4 ms. Jumlah *opcode* bertambah dari 24 menjadi 185.

Selain itu, selama masa pakai seri komputer ini, kecepatan relatif CPU meningkat sebesar faktor 50. Peningkatan kecepatan dicapai oleh elektronik yang lebih baik (misalnya, implementasi transistor lebih cepat daripada implementasi tabung vakum) dan lebih banyak lagi sirkuit yang kompleks. Sebagai contoh, IBM 7094 mencakup *Instruction Backup Register*, yang digunakan untuk buffer instruksi berikutnya. *Control Unit* mengambil dua word yang berdekatan dari memori untuk mengambil instruksi. Kecuali untuk terjadinya instruksi percabangan, yang relatif jarang (mungkin 10 hingga 15%), ini berarti bahwa *Control Unit* memiliki akses ke memori untuk instruksi pada hanya setengah siklus instruksi. Pengambilan awal ini secara signifikan mengurangi waktu siklus instruksi rata-rata.

C. Generasi Ketiga: IC

Sepanjang tahun 1950-an dan awal 1960-an, peralatan elektronik sebagian besar terdiri dari komponen diskrit yang berisi transistor, resistor, kapasitor, dan sebagainya. Komponen diskrit diproduksi secara terpisah, dikemas dalam wadahnya sendiri, dan disolder atau disambungkan bersama ke papan sirkuit, yang kemudian dipasang di komputer, osiloskop, dan peralatan elektronik lainnya. Seluruh proses pembuatan, dari transistor ke papan sirkuit pada saat itu masih mahal dan rumit. Komputer generasi kedua awal mengandung sekitar 10.000 transistor. Angka ini tumbuh hingga ratusan ribu, membuat pembuatan mesin yang lebih baru, lebih kuat semakin sulit. Pada tahun 1958 terjadi pencapaian yang merevolusi elektronik diskrit dan memulai era mikroelektronika, yaitu dengan penemuan *Integrated Circuit* (IC) yang mendefinisikan komputer generasi ketiga. Dua anggota paling penting dari generasi ketiga, yang diperkenalkan pada awal era itu adalah: Sistem IBM/360 dan DEC PDP-8.

Elemen dasar dari komputer digital, harus dapat melakukan penyimpanan data, perpindahan data, pemrosesan, dan fungsi kontrol. Hanya dua jenis komponen dasar yang diperlukan, yaitu gerbang (*gate*) dan sel memori. Gerbang (*gate*) adalah perangkat yang mengimplementasikan fungsi Boolean atau logika sederhana (Gambar 1.12). Sebagai contoh gerbang AND dengan *input* A dan B, serta *output* C, mengimplementasikan ekspresi JIKA A DAN B BENAR, maka C BENAR. Perangkat semacam itu disebut gerbang karena mereka mengontrol aliran data dengan cara yang hampir sama dengan gerbang kanal yang mengontrol aliran air.



Sumber: (Stallings, 2016)

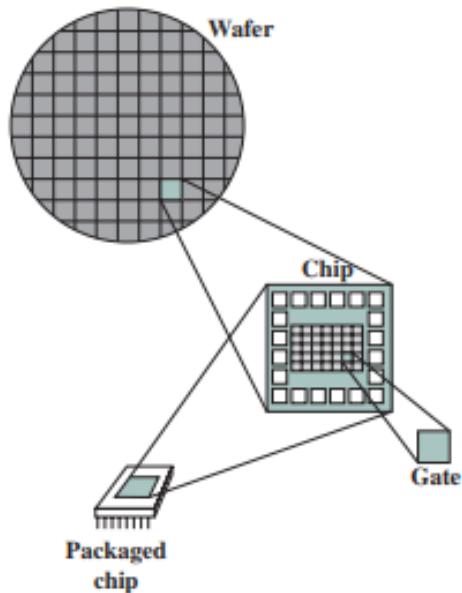
Gambar 1.12. Elemen dasar komputer

Sel memori adalah perangkat yang dapat menyimpan 1 bit data; artinya, perangkat dapat berada di salah satu dari dua kondisi stabil kapan saja. Dengan menghubungkan banyak perangkat fundamental ini, maka dapat dibangun sebuah komputer. Dengan gerbang dan sel memori, dapat dibangun empat fungsi dasar sebuah komputer berikut:

1. **Data storage**, disediakan oleh sel-sel memori.
2. **Data processing**, disediakan oleh gerbang-gerbang logika (*gates*).
3. **Data movement**, jalur antar komponen digunakan untuk memindahkan data dari memori ke memori dan dari memori melalui *gates* ke memori.
4. **Control**, jalur antar komponen dapat membawa sinyal kendali. Misalnya, gerbang akan memiliki satu atau dua *input* data berikut sinyal *input* kendali yang mengaktifkan gerbang. Ketika sinyal kendali AKTIF, gerbang melakukan fungsinya pada *input* data dan menghasilkan *output* data. Sebaliknya, ketika sinyal kendali MATI, garis *output* adalah nol, seperti yang dihasilkan oleh status dengan impedansi tinggi. Demikian pula, sel memori akan menyimpan bit yang ada pada pin *inputnya* ketika sinyal kontrol WRITE dalam posisi ON dan akan menempatkan bit yang ada di dalam sel pada pin *outputnya* ketika sinyal kontrol READ dalam posisi ON.

Dengan demikian, elemen dasar komputer terdiri dari *gates*, sel memori, dan interkoneksi di antara elemen-elemen tersebut. *Gates* dan sel memori, dibangun dari komponen elektronik sederhana, seperti transistor dan kapasitor. Sirkuit terpadu memanfaatkan keadaan dimana komponen seperti transistor, resistor, dan konduktor dapat dibuat dari bahan semikonduktor seperti silikon. Ini merupakan seni *solid state* untuk membuat seluruh rangkaian dalam sepotong kecil silikon daripada merakit komponen terpisah yang dibuat dari potongan-potongan silikon terpisah ke dalam sirkuit yang sama. Banyak transistor dapat diproduksi pada waktu bersamaan pada satu wafer silikon.

Gambar 1.13 menyajikan konsep-konsep kunci dalam sirkuit terintegrasi. *Wafer* tipis silikon dibagi menjadi matriks area kecil, masing-masing beberapa milimeter persegi. Pola rangkaian identik dibuat di setiap area, dan *wafer* dipecah menjadi *chips*. Setiap *chip* terdiri dari banyak gerbang dan/atau sel memori ditambah sejumlah titik lampiran *input* dan *output*. *Chip* ini kemudian dikemas dalam *package* yang melindunginya dan menyediakan pin untuk pemasangan ke perangkat di luar *chip*. Sejumlah *package* ini kemudian dapat dihubungkan pada papan sirkuit cetak untuk menghasilkan sirkuit yang lebih besar dan lebih kompleks.

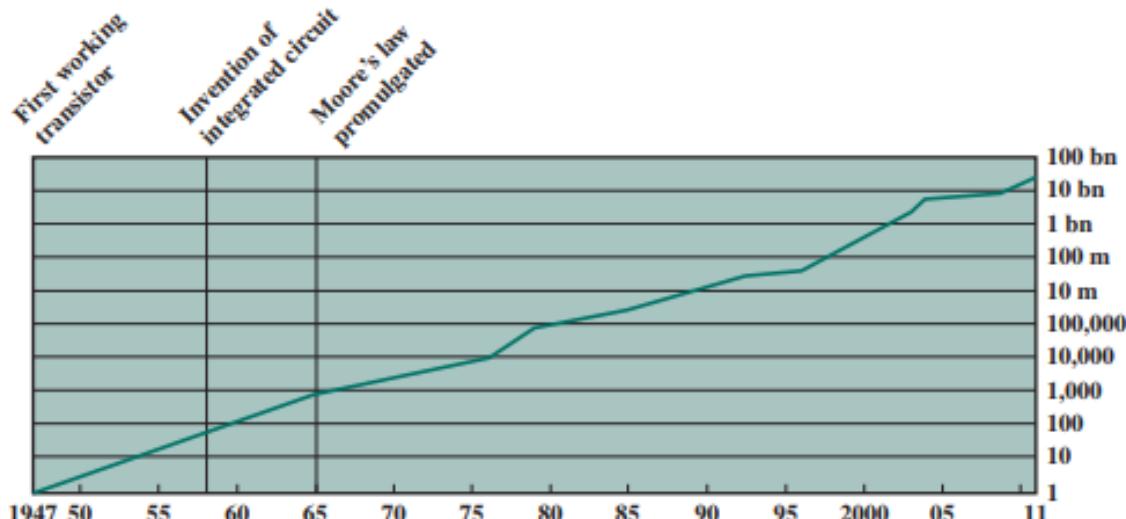


Sumber: (Stallings, 2016)

Gambar 1.13. *Chip* dan *Gate*

Awalnya, hanya beberapa gerbang atau sel memori yang dapat diproduksi dan dikemas secara andal. Teknologi IC awal ini disebut sebagai *Small Scale Integration* (SSI). Seiring berjalanannya waktu, teknologi terus berkembang, sehingga menjadi mungkin untuk mengemas lebih banyak komponen pada *chip* dengan ukuran yang sama. Pertumbuhan kepadatan transistor yang diilustrasikan pada Gambar 1.14 adalah salah satu tren teknologi paling luar biasa, yang kemukakan oleh Gordon Moore pada tahun 1965 dengan hukum Moore yang terkenal. Moore mengamati bahwa jumlah transistor yang dapat dimasukkan ke dalam satu *chip* meningkat dua kali lipat setiap tahun, dan dengan tepat meramalkan bahwa laju ini akan terus berlanjut dalam waktu dekat. Konsekuensi dari hukum Moore sangat besar, yaitu:

1. Biaya sebuah *chip* tetap tidak berubah selama periode pertumbuhan kepadatan yang cepat. Ini berarti bahwa biaya elemen logika komputer dan sirkuit memori telah turun pada tingkat yang dramatis.
2. Karena elemen logika dan memori ditempatkan berdekatan pada *chip* yang lebih padat, panjang jalur listrik dipersingkat, meningkatkan kecepatan pengoperasian.
3. Komputer menjadi lebih kecil, membuatnya lebih nyaman untuk ditempatkan di berbagai lingkungan.
4. Ada pengurangan kebutuhan daya.
5. Interkoneksi pada *Integrated Circuits* (IC) jauh lebih dapat diandalkan daripada koneksi solder. Dengan lebih banyak sirkuit pada setiap *chip*, ada lebih sedikit koneksi *interchip*.



Sumber: (Stallings, 2016)

Gambar 1.14. Pertumbuhan jumlah transistor pada sebuah IC berdasarkan hukum Moore

Pada tahun 1964, IBM memiliki pengaruh yang kuat di pasar komputer dengan 7000 seri mesinnya. Pada tahun itu, IBM mengumumkan System/360, keluarga baru produk komputernya. System/360 adalah keluarga komputer terencana pertama di industri. Model-model tersebut kompatibel, dalam arti bahwa program yang ditulis untuk suatu model harus mampu dieksekusi oleh model lain dalam seri yang sama, dengan hanya perbedaan waktu yang diperlukan untuk mengeksekusi. Konsep keluarga komputer yang kompatibel sangatlah sukses. Pengguna dengan persyaratan sederhana dan anggaran yang cocok dapat mulai dengan Model 360 yang relatif murah. Kemudian, jika kebutuhan transaksi bertambah, dimungkinkan untuk meningkatkan ke mesin yang lebih cepat dengan memori lebih besar tanpa mengorbankan investasi dalam perangkat lunak yang sudah dikembangkan. Karakteristik keluarga komputer adalah sebagai berikut:

- 1) Set instruksi yang serupa atau identik.
Dalam banyak kasus, set instruksi mesin yang sama persis didukung pada semua anggota keluarga. Dengan demikian, sebuah program yang dijalankan pada satu mesin juga akan dieksekusi pada yang lain. Dalam beberapa kasus, bagian bawah keluarga memiliki set instruksi yang merupakan bagian dari bagian atas keluarga. Ini berarti bahwa program dapat bergerak ke atas tetapi tidak ke bawah.
- 2) Sistem operasi serupa atau identik.
Sistem operasi dasar yang sama tersedia untuk semua anggota keluarga. Dalam beberapa kasus, fitur tambahan ditambahkan ke anggota kelas atas.
- 3) Peningkatan kecepatan.
Tingkat pelaksanaan instruksi meningkat dari yang lebih rendah ke anggota keluarga yang lebih tinggi.
- 4) Menambah jumlah port I/O.
Jumlah port I/O meningkat dari yang lebih rendah ke anggota keluarga yang lebih tinggi.
- 5) Meningkatkan ukuran memori.
Ukuran memori utama meningkat dari yang lebih rendah ke anggota keluarga yang lebih tinggi.
- 6) Meningkatnya biaya.
Pada titik waktu tertentu, biaya sistem meningkat dari yang lebih rendah ke anggota keluarga yang lebih tinggi.

Pada tahun yang sama ketika IBM memproduksi System/360 pertamanya, produksi pertama yang penting lainnya adalah: PDP-8 dari *Digital Equipment Corporation* (DEC). Pada saat komputer rata-rata

membutuhkan ruang ber-AC, PDP-8 cukup kecil sehingga dapat ditempatkan di atas bangku laboratorium atau dibangun ke peralatan lain (dijuluki komputer mini oleh industri). Hal tersebut tidak bisa dilakukan oleh semua *mainframe*, tetapi dengan harga \$ 16.000, pada saat itu cukup murah bagi setiap teknisi lab untuk memiliki. Biaya rendah dan ukuran kecil dari PDP-8 memungkinkan produsen lain untuk membeli PDP-8 dan mengintegrasikannya ke dalam sistem lain untuk dijual kembali. Pabrikan lain ini kemudian dikenal sebagai *Original Equipment Manufacturers* (OEM), dan pasar OEM selalu menjadi segmen utama pasar komputer.

D. Generasi Masa Depan

Setelah generasi ketiga, ada sedikit kesepakatan umum tentang mendefinisikan generasi komputer. Tabel 1.2 menunjukkan bahwa ada sejumlah generasi selanjutnya, berdasarkan kemajuan teknologi IC. Dengan diperkenalkannya *Large Scale Integrated* (LSI), lebih dari 1.000 komponen dapat ditempatkan pada satu *chip* sirkuit terpadu. *Very-Large Scale Integrated* (VLSI) mencapai lebih dari 10.000 komponen per-*chip*, sementara *chip* *Ultra Large Scale Integrated* (ULSI) saat ini dapat berisi lebih dari satu miliar komponen.

Dengan pesatnya perkembangan teknologi, tingginya tingkat pengenalan produk baru, serta pentingnya perangkat lunak dan serta perangkat keras, klasifikasi berdasarkan generasi menjadi kurang jelas dan kurang bermakna. Terdapat dua perkembangan terpenting pada generasi selanjutnya, yaitu memori semikonduktor dan prosesor.

Memori Semikonduktor

Aplikasi pertama teknologi IC untuk komputer adalah pembangunan prosesor (*Control Unit* dan unit aritmatika dan logika). Tetapi juga ditemukan bahwa teknologi yang sama ini dapat digunakan untuk membangun memori. Pada generasi sebelumnya (1950-an dan 1960-an), sebagian besar memori komputer dibangun dari cincin-cincin kecil bahan feromagnetik, masing-masing berdiameter sekitar 16 inch. Cincin ini digantung di kisi-kisi kabel halus yang tergantung di layar kecil di dalam komputer. Magnet satu arah, sebuah cincin (disebut inti) mewakili satu; magnet sebaliknya, itu berdiri untuk nol. Memori inti-magnetik agak cepat; hanya butuh sepersejuta detik untuk membaca sedikit yang tersimpan dalam memori. Tapi itu mahal dan besar, dan menggunakan pembacaan destruktif (tindakan sederhana membaca inti menghapus data yang tersimpan di dalamnya), oleh karena itu perlu menginstal sirkuit untuk memulihkan data segera setelah itu diekstraksi.

Kemudian, pada tahun 1970, Fairchild menghasilkan memori semikonduktor pertama yang berkapasitas relatif besar. *Chip* ini, seukuran inti tunggal, dapat menampung 256 bit memori. Itu tidak merusak dan jauh lebih cepat dari inti. Hanya butuh 70 miliar detik untuk membaca sedikit. Namun, biaya per bit lebih tinggi daripada untuk inti. Pada tahun 1974 harga per bit memori semikonduktor turun di bawah harga per bit memori inti. Kemudian, ada penurunan yang berkelanjutan dan cepat dalam biaya produksi memori disertai dengan peningkatan kepadatan memori fisik yang sesuai. Hal ini memungkinkan untuk membuat mesin yang lebih kecil, lebih cepat dengan ukuran memori mesin yang lebih besar dan lebih mahal dari hanya beberapa tahun sebelumnya. Perkembangan teknologi memori, bersama dengan perkembangan teknologi prosesor, mengubah sifat komputer dalam waktu kurang dari satu dekade. Sejak 1970 sampai 2016, memori semikonduktor telah melewati 13 generasi: 1k, 4k, 16k, 64k, 256k, 1M, 4M, 16M, 64M, 256M, 1G, 4G, dan, 8 Gb pada satu *chip* ($1k = 2^{10}$, $1M = 2^{20}$, $1G = 2^{30}$). Setiap generasi telah memberikan peningkatan kepadatan penyimpanan, disertai dengan penurunan biaya per bit dan menurunnya waktu akses.

Prosesor

Sama seperti kepadatan elemen pada *chip* memori yang terus meningkat, demikian juga kepadatan elemen pada *chip* prosesor. Seiring berjalannya waktu, semakin banyak elemen dapat ditempatkan pada setiap *chip*, sehingga semakin sedikit *chip* yang diperlukan untuk membangun komputer dengan

prosesor tunggal. Sebuah terobosan dicapai pada tahun 1971, ketika Intel mengembangkan 4004, mikroprosesor 4 bit pertama.

Intel 4004 adalah *chip* pertama yang mengandung semua komponen CPU pada satu *chip*. 4004 menandakan kelahiran mikroprosesor. 4004 dapat menambahkan dua angka 4-bit dan hanya dapat dikalikan dengan penambahan berulang. Menurut standar saat ini, 4004 itu sangat primitif, tetapi menandai awal dari evolusi berkelanjutan kemampuan dan kekuatan rōprosesor. Evolusi ini dapat dilihat paling mudah dalam jumlah bit yang berhubungan dengan prosesor pada suatu waktu. Tidak ada ukuran yang jelas tentang hal ini, tetapi mungkin ukuran terbaik adalah lebar bus data: jumlah bit data yang dapat dibawa masuk atau dikirim keluar dari prosesor pada suatu waktu. Ukuran lain adalah jumlah bit dalam akumulator atau dalam set General Purpose Register. Seringkali, langkah-langkah ini bersamaan, tetapi tidak selalu. Sebagai contoh, sejumlah mikroprosesor dikembangkan yang beroperasi pada 16-bit dalam register tetapi hanya dapat membaca dan menulis 8 bit pada satu waktu. Langkah besar berikutnya dalam evolusi mikroprosesor adalah pengenalan pada tahun 1972 dari Intel 8008. Ini adalah mikroprosesor 8-bit pertama dan hampir dua kali lebih kompleks dari 4004.

Peristiwa besar berikutnya pengenalan mikroprosesor Intel 8080 pada tahun 1974, yang merupakan mikroprosesor *General Purpose* pertama. Bila 4004 dan 8008 dirancang untuk aplikasi spesifik, maka 8080 dirancang untuk menjadi CPU dari *general purpose* komputer mikro. Seperti 8008, 8080 adalah mikroprosesor 8-bit, namun 8080 lebih cepat, memiliki set instruksi yang lebih banyak, dan memiliki kemampuan pengalaman yang besar. Pada waktu yang sama, mikroprosesor 16-bit mulai dikembangkan. Namun, baru pada akhir tahun 1970-an mikroprosesor 16-bit untuk *general purpose* muncul. Salah satunya adalah 8086. Intel memperkenalkan mikroprosesor 32-bit pertama pada tahun 1985, yaitu Intel 80386 (Tabel 1.3).

Tabel 1.3. Evolusi Prosesor Keluarga Intel

(a) Prosesor tahun 1970-an

Spesifikasi	4004	8008	8080	8086	8088
Spesifikasi	1971	1972	1974	1978	1979
Clock speeds	108 KHz	108 kHz	2 MHz	5 MHz, 8 MHz, 10 MHz	5 MHz, 8MHz
Lebar Bus	4 bit	8 bit	8 bit	16 bit	8 bit
Jumlah transistor	2.300	3.500	6.000	29.000	29.00
Ukuran (μm)	10	8	6	3	6
Jumlah word	640 Byte	16 KB	64 KB	1 MB	1 MB

(a) Prosesor tahun 1980-an

Spesifikasi	80286	386TM DX	386TM SX	486TM DX CPU
Spesifikasi	1982	1985	1988	1989
Clock speeds	6-12.5 MHz	16-33 MHz	16-33 MHz	25-50 MHz
Lebar Bus	16 bit	32 bit	16 bit	32 bit
Jumlah transistor	134.000	275.000	275.000	1.2 Juta
Ukuran (μm)	1.5	1	1	0.8-1
Jumlah word	16 MB	4 GB	16 MB	4 GB
Virtual memory Cache	1 GB	64 TB	64 TB	64 TB
	-	-	-	8 KB

(b) Prosesor tahun 1990-an

Spesifikasi	486TM SX	Pentium	Pentium Pro	Pentium II
Spesifikasi	1991	1993	1995	1997
Clock speeds	16-33 MHz	66-166 MHz	150-166 MHz	200-300 MHz
Lebar Bus	32 bit	32 bit	64 bit	64 bit
Jumlah transistor	1.185 juta	3.1 juta	5.5 juta	7.5 Juta
Ukuran (μm)	1	0.8	0.6	0.35

Jumlah word Virtual memory Cache	4 MB 64 GB 8 KB	4 GB 64 TB 8 KB	64 GB 64 TB 512 KB L1 dan 1 MB L2	64 GB 64 TB 8 KB
----------------------------------------	-----------------------	-----------------------	-----------------------------------------	------------------------

(a) Prosesor tahun 1999-2013

	Pentium III	Pentium 4	Core 2 Duo	Core i7 EE 4960X
Spesifikasi	1999	2000	2006	2013
Clock speeds	450-660 MHz	1.3-1.8 GHz	1.06-1.6 GHz	4 GHz
Lebar Bus	64 bit	64 bit	64 bit	64 bit
Jumlah transistor	9.5 juta	42 juta	167 juta	1.860 juta
Ukuran (nm)	250	180	65	22
Jumlah word	4 GB	4 GB	64 GB	64 GB
Virtual memory	64 TB	64 TB	64 TB	64 TB
Cache	512 KB L2	256 KB L2 TB	2 MB L2 TB	1.5 MB L2/15 MB L3
Jumlah core	1	1	2	6

Sumber: (Stallings, 2016)

Penawaran X86 saat itu merupakan hasil dari upaya desain pada *Complex Instruction Set Computer* (CISC). X86 menggabungkan prinsip-prinsip desain canggih yang dulu hanya ditemukan pada mainframe dan superkomputer dan berfungsi sebagai contoh sempurna desain CISC. Pendekatan alternatif untuk desain prosesor adalah *Reduce Instruction Set Computer* (RISC).

ARM (Advance RISC Mechan) merupakan arsitekur prosesor 32 bit yang dikembangkan oleh ARM Limited. Pada awalnya ARM digunakan pada komputer desktop. Karena ARM memiliki desain yang sederhana, sehingga banyak digunakan dalam berbagai *embedded system* dan merupakan salah satu sistem berbasis RISC dengan rancangan terbaik di pasaran. Selama tahun 2007 sampai 2009, ARM mendominasi pasar prosesor 32 bit RISC.

Dalam hal pangsa pasar, Intel memiliki peringkat sebagai pembuat mikroprosesor nomor satu untuk sistem yang tidak tertanam selama beberapa dekade. Evolusi produk mikroprosesor andalannya berfungsi sebagai indikator yang baik untuk evolusi teknologi komputer secara umum. Berikut adalah hal yang menjadi ciri khas dari masing-masing prosesor keluar Intel:

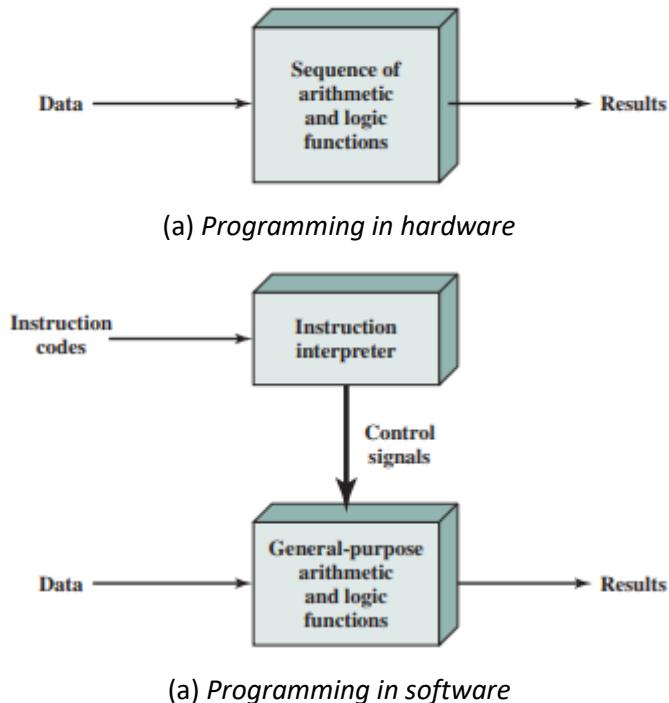
- 1) 8080: Mikroprosesor tujuan umum pertama di dunia. Ini adalah mesin 8-bit, dengan jalur data 8-bit ke memori. 8080 digunakan pada Personal Computer (PC) pertama, Altair.
- 2) 8086: Mesin 16-bit yang jauh lebih kuat. Selain jalur data yang lebih luas dan register yang lebih besar, 8086 memakai *cache* instruksi, atau antrian, yang mengambil beberapa instruksi sebelum dijalankan. Varian dari prosesor ini, 8088, digunakan pada PC pertama IBM. 8086 adalah penampilan pertama arsitektur x86.
- 3) 80286: Pengembangan dari 8086 ini memungkinkan untuk menangani memori 16-MB, bukan hanya 1 MB.
- 4) 80386: Mesin 32-bit pertama Intel, dan perombakan besar produk. Dengan arsitektur 32-bit, 80386 menyaingi kompleksitas dan kekuatan minicomputer dan mainframe yang diperkenalkan hanya beberapa tahun sebelumnya. Ini adalah prosesor Intel pertama yang mendukung multitasking, yang berarti dapat menjalankan banyak program secara bersamaan.
- 5) 80486: 80486 memperkenalkan penggunaan teknologi *cache* yang jauh lebih canggih dan kuat serta *pipeline* instruksi yang canggih. 80486 juga menawarkan coprocessor matematika built-in, memisahkan operasi matematika yang rumit dari CPU utama.
- 6) Pentium: Dengan Pentium, Intel memperkenalkan penggunaan teknik superscalar, yang memungkinkan beberapa instruksi dieksekusi secara paralel.
- 7) Pentium Pro: Pentium Pro melanjutkan perpindahannya ke organisasi superscalar yang dimulai dengan Pentium, dengan penggunaan penggantian nama register yang agresif, prediksi cabang, analisis aliran data, dan eksekusi spekulatif.

- 8) Pentium II: Pentium II menggabungkan teknologi Intel MMX, yang dirancang khusus untuk memproses data video, audio, dan grafik secara efisien.
- 9) Pentium III: Pentium III menggabungkan instruksi *floating-point* tambahan: Streaming set instruksi SSD Extensions (SSE) menambahkan 70 instruksi baru yang dirancang untuk meningkatkan kinerja ketika operasi yang sama persis dilakukan pada beberapa objek data. Aplikasi tipikal adalah pemrosesan sinyal digital dan pemrosesan grafik.
- 10) Pentium 4: Pentium 4 mencakup *floating-point* tambahan dan perangkat tambahan lainnya untuk multimedia
- 11) *Core*: Ini adalah mikroprosesor Intel x86 pertama dengan inti ganda, mengacu pada implementasi dua core pada satu *chip*.
- 12) *Core 2*: *Core 2* memperluas arsitektur *Core* hingga 64 bit. *Core 2 Quad* menyediakan empat core dalam satu *chip*. Penawaran *Core* terbaru memiliki hingga 10 *core* per *chip*.

1.3. Fungsi Komputer dan Interkoneksi

Hampir semua desain komputer kontemporer didasarkan pada konsep yang dikembangkan oleh John von Neumann di Institute for Advanced Studies, Princeton. Desain seperti itu disebut sebagai arsitektur von Neumann dan didasarkan pada tiga konsep utama, yaitu:

- 1) Data dan instruksi disimpan dalam satu memori baca-tulis.
- 2) Isi memori dapat dialamatkan berdasarkan lokasi, tanpa memperhatikan jenis data yang terkandung di dalamnya.
- 3) Eksekusi terjadi secara berurutan (kecuali diubah secara eksplisit) dari satu instruksi ke instruksi berikutnya.



Sumber: (Stallings, 2016)

Gambar 1.15. Pendekatan Hardware dan Software

Ada satu set kecil komponen logika dasar yang dapat digabungkan dengan berbagai cara untuk menyimpan data biner dan melakukan operasi aritmatika dan logika pada data tersebut. Jika ada perhitungan tertentu yang harus dilakukan, konfigurasi komponen logika yang dirancang khusus untuk perhitungan itu dapat dibangun. Proses menghubungkan berbagai komponen logika dalam konfigurasi

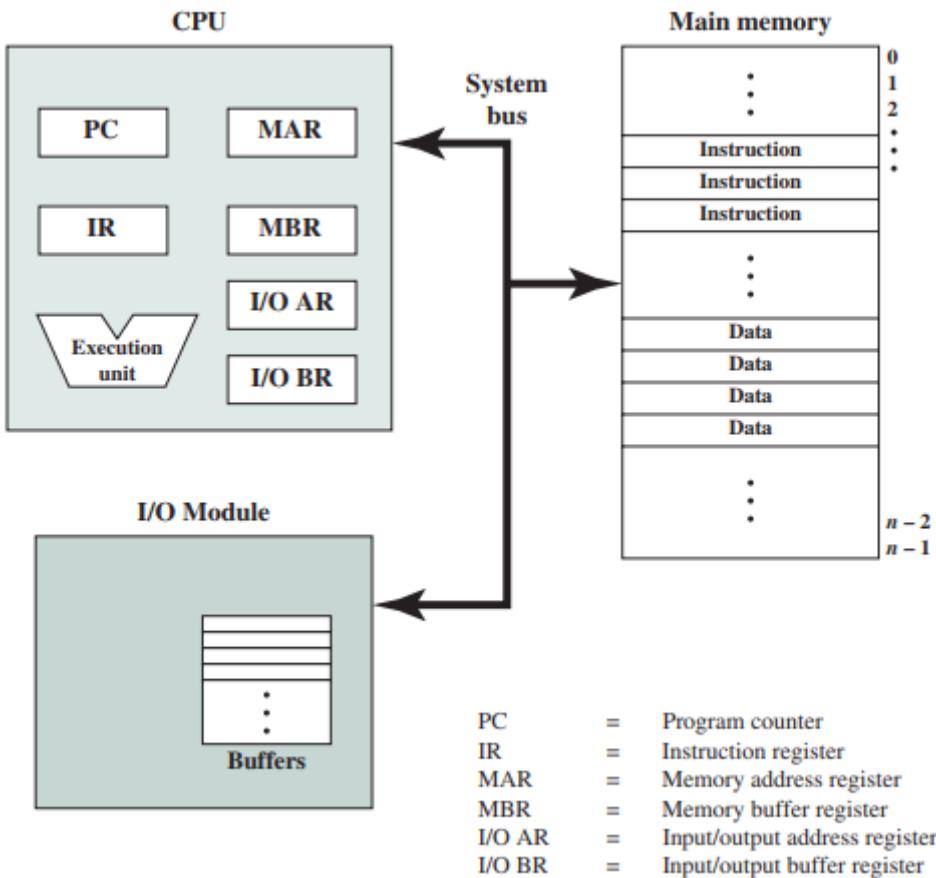
yang diinginkan sebagai bentuk pemrograman disebut sebagai *hardwired program*. Misalkan pada konfigurasi *Arithmatic and Ligit Unit* dengan fungsi umum. Perangkat keras ini akan melakukan berbagai fungsi pada data tergantung pada sinyal kontrol yang diterapkan pada perangkat keras. Dalam kasus asli perangkat keras khusus, sistem menerima data dan menghasilkan *output* (Gambar 1.15a). Dengan perangkat keras serba guna, sistem menerima data dan mengontrol sinyal dan menghasilkan *output*. Seluruh program sebenarnya adalah urutan langkah-langkah. Pada setiap langkah, beberapa operasi aritmatika atau logika dilakukan pada beberapa data. Untuk setiap langkah, satu set sinyal kontrol baru diperlukan (Gambar 1.15).

Pemrograman sekarang jauh lebih mudah. Alih-alih memasang ulang perangkat keras untuk setiap program baru, yang perlu dilakukan adalah menyediakan urutan kode baru. Setiap kode, pada dasarnya, merupakan instruksi, dan bagian dari perangkat keras mengartikan setiap instruksi dan menghasilkan sinyal kontrol. Untuk membedakan metode pemrograman baru ini, urutan kode atau instruksi disebut perangkat lunak.

Gambar 1.15b menunjukkan dua komponen utama sistem: instruksi interpreter dan modul fungsi *general-purpose Arithmatic and Logical Unit*. Keduanya merupakan bagian dari CPU. Data dan instruksi harus dimasukkan ke dalam sistem, untuk ini diperlukan semacam modul *input*. Modul input ini berisi komponen dasar untuk menerima data dan instruksi dalam beberapa bentuk dan mengubahnya menjadi bentuk internal sinyal yang dapat digunakan oleh sistem. Diperlukan sarana pelaporan, dan ini dalam bentuk modul keluaran. Secara keseluruhan, ini disebut sebagai komponen I/O. Diperlukan satu komponen lagi. Perangkat *input* akan membawa instruksi dan data secara berurutan. Tetapi suatu program tidak selalu dieksekusi secara berurutan; mungkin melompat (misalnya: instruksi lompatan). Demikian pula, operasi pada data mungkin memerlukan akses ke lebih dari satu elemen pada suatu waktu dalam urutan yang telah ditentukan. Dengan demikian, harus ada tempat untuk menyimpan sementara instruksi dan data. Modul itu disebut memori, atau memori utama, untuk membedakannya dari penyimpanan eksternal atau perangkat periferal. Von Neumann menunjukkan bahwa memori yang sama dapat digunakan untuk menyimpan instruksi dan data.

Gambar 1.16 mengilustrasikan komponen-komponen tingkat atas ini dan menyarankan interaksi di antara mereka. CPU bertukar data dengan memori. Untuk tujuan ini, biasanya menggunakan dua register internal (ke CPU): *Memory Address Register (MAR)*, yang menentukan alamat dalam memori untukaca atau tulis berikutnya, dan *Memory Buffer Register (MBR)*, yang berisi data yang akan ditulis ke dalam memori atau menerima data yang dibaca dari memori. Demikian pula, *I/O Address Register (I/O AR)* menentukan perangkat I/O tertentu. *I/O Buffer Register (I/O BR)* digunakan untuk pertukaran data antara modul I/O dan CPU.

Modul memori terdiri dari serangkaian lokasi, yang ditentukan oleh alamat yang diberi nomor urut. Setiap lokasi berisi nomor biner yang dapat diartikan sebagai instruksi atau data. Modul I/O mentransfer data dari perangkat eksternal ke CPU dan memori, dan sebaliknya. Ini berisi buffer internal untuk sementara menyimpan data ini sampai mereka dapat dikirim.



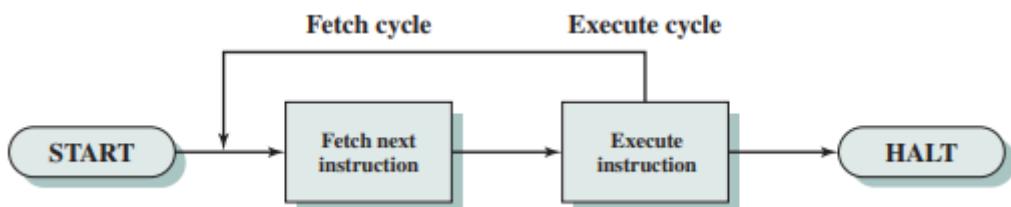
Sumber: (Stallings, 2016)

Gambar 1.16. Komponen Komputer: Tampilan Tingkat Atas

A. *Instruction Fetch* dan *Execute*

Fungsi dasar yang dilakukan oleh komputer adalah mengeksekusi suatu program, yang terdiri dari sekumpulan instruksi yang tersimpan dalam memori. Prosesor melakukan pekerjaan yang sebenarnya dengan mengeksekusi instruksi yang ditentukan dalam program.

Pemrosesan instruksi yang paling sederhana terdiri dari dua langkah: Prosesor membaca instruksi dari memori satu per satu, dan mengeksekusi setiap instruksi. Eksekusi instruksi dapat melibatkan beberapa operasi dan tergantung pada sifat instruksi. Pemrosesan yang diperlukan untuk satu instruksi disebut siklus instruksi.



Sumber: (Stallings, 2016)

Gambar 1.17. Siklus Instruksi Dasar

Siklus instruksi sederhana digambarkan pada Gambar 1.17. Dua langkah tersebut, adalah siklus pengambilan (*fetch cycle*) dan siklus eksekusi (*execute cycle*). Eksekusi program berhenti hanya jika mesin dimatikan, terjadi kesalahan yang tidak dapat dipulihkan, atau bertemu dengan instruksi program yang menghentikan komputer.

Di awal setiap siklus instruksi, prosesor mengambil instruksi dari memori. Register pada prosesor yang disebut *Program Counter (PC)* menyimpan alamat instruksi yang akan diambil berikutnya. Prosesor akan selalu menambah register **PC** setelah setiap instruksi diambil, sehingga register PC berisi alamat instruksi berikutnya yang akan diambil secara berurutan (misalnya: instruksi terletak di alamat memori yang lebih tinggi berikutnya). Asumsikan bahwa register **PC** diatur ke lokasi memori 300, di mana alamat lokasi merujuk pada *word* 16-bit. Prosesor selanjutnya akan mengambil instruksi di lokasi 300. Pada siklus instruksi berikutnya, ia akan mengambil instruksi dari lokasi 301, 302, 303, dan seterusnya. Urutan ini dapat diubah, dengan mengganti isi register PC dengan lokasi yang diinginkan.

Instruksi yang diambil dimuat ke register yang dikenal sebagai register instruksi (IR). Instruksi berisi bit yang menentukan tindakan yang harus dilakukan prosesor. Prosesor menginterpretasikan instruksi dan melakukan tindakan yang diperlukan. Secara umum, tindakan ini terbagi dalam empat kategori, dimana eksekusi sebuah instruksi dapat melibatkan kombinasi tindakan ini:

- 1) Memori-Prosesor: Data dapat ditransfer dari prosesor ke memori atau dari memori ke prosesor.
- 2) Processor-I/O: Data dapat ditransfer ke atau dari perangkat periferal dengan mentransfer antara prosesor dan modul I/O.
- 3) *Data processing*: Prosesor dapat melakukan beberapa operasi aritmatika atau logika pada data.
- 4) *Control*: Suatu instruksi dapat menentukan perubahan urutan eksekusi. Misalnya, prosesor dapat mengambil instruksi dari lokasi 149, yang menentukan bahwa instruksi berikutnya berasal dari lokasi 182. Prosesor akan mengingat fakta ini dengan mengatur **PC** ke 182. Dengan demikian, pada siklus pengambilan berikutnya, instruksi akan menjadi diambil dari lokasi 182 daripada 150. Gambar 1.18 menunjukkan contoh format instruksi dan bilangan integer 16 bit. Prosesor berisi register data tunggal, yang disebut akumulator (**AC**). Instruksi dan data panjangnya 16 bit. Dengan demikian, akan lebih mudah untuk mengatur memori menggunakan *words* 16-bit. Format instruksi menyediakan 4 bit untuk *opcode*, sehingga dapat menyediakan sebanyak $2^4 = 16$ *opcode* yang berbeda (mulai dari biner 0000 sampai 1111), dan sebanyak $2^{12} = 4096$ (4K) *word* memori dapat dialamat.

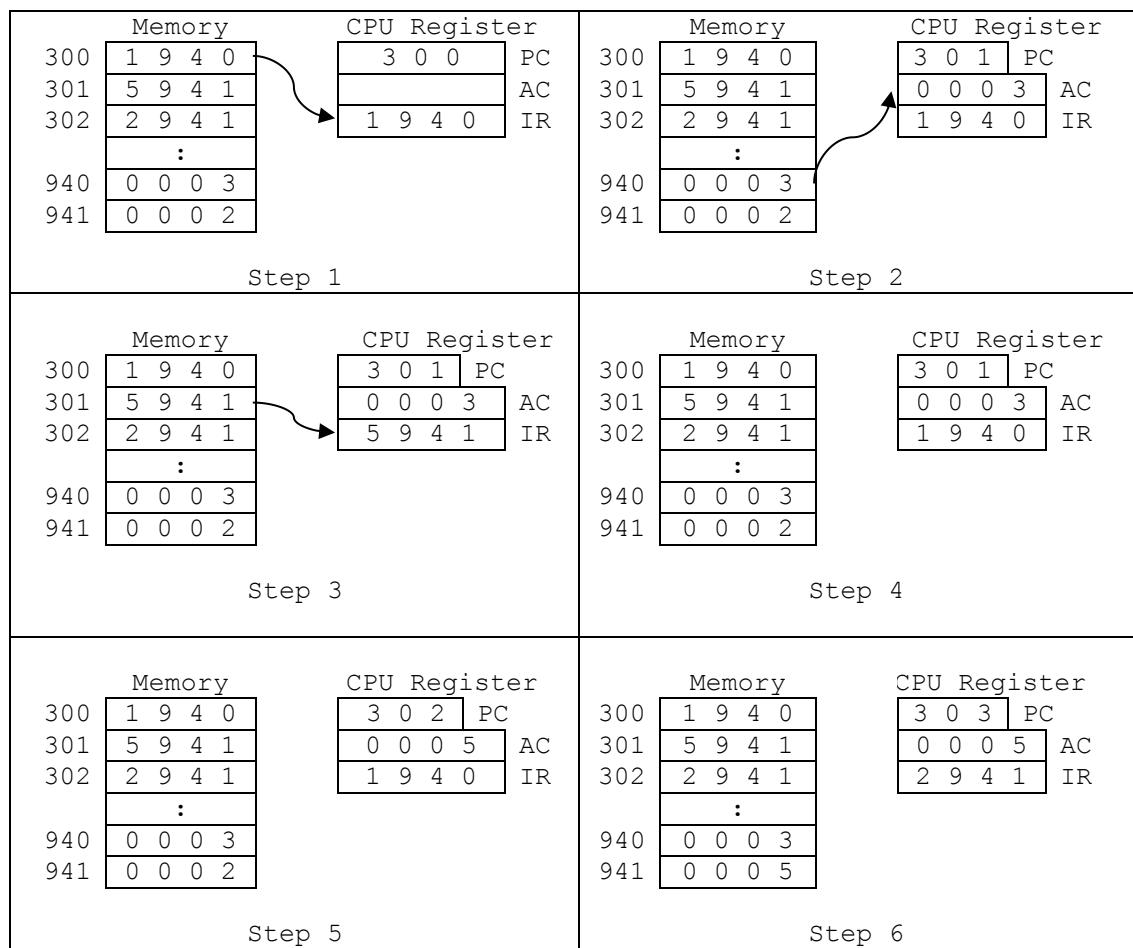
0	3 4	15
Opcode	Address	
(a) Format Instruksi 16 bit		

S	Magnitude
(b) Bilangan Integer 16 bit	

Sumber: (Stallings, 2016)

Gambar 1.18. Format Instruksi dan Bilangan Integer

Dalam eksekusi instruksi, register **PC** menyimpan alamat instruksi berikutnya yang akan dieksekusi, **IR** berisi instruksi yang sedang dieksekusi, dan register **AC** digunakan sebagai register penyimpanan sementara.



Sumber: (Stallings, 2016)

Gambar 1.19. Contoh eksekusi program (isi memori dan register dalam format heksadesimal)

Gambar 1.19 mengilustrasikan eksekusi sebagian program. Pada gambar tersebut menunjukkan bagian memori dan register prosesor yang dianalisa. Diasumsikan, pada awalnya register PC berisi 300 (instruksi yang dijemput berada pada lokasi 300). Tiga instruksi, yang dapat digambarkan sebagai tiga siklus pengambilan dan tiga siklus eksekusi, diuraikan sebagai berikut:

1. **PC** berisi 300, alamat instruksi pertama. Instruksi ini (nilai 1940 dalam heksadesimal) dimuat ke dalam register instruksi **IR**, dan **PC** bertambah 1, menjadi 3001. Proses ini melibatkan penggunaan MAR dan MBR, namun untuk mempermudah, register perantara ini diabaikan.
2. 4 bit pertama (digit heksadesimal pertama) dalam **IR** adalah opcode, menunjukkan bahwa **AC** harus dimuat. Sisa 12 bit (tiga digit heksadesimal) menentukan alamat (940) dari mana data akan dimuat.
3. Instruksi berikutnya (5941) diambil dari lokasi 301, dan **PC** bertambah.
4. Konten lama **AC** dan isi lokasi 941 ditambahkan, dan hasilnya disimpan dalam **AC**.
5. Instruksi berikutnya (2941) diambil dari lokasi 302, dan **PC** bertambah.
6. Isi **AC** disimpan di lokasi 941.

Dalam contoh ini, tiga siklus instruksi (masing-masing terdiri dari siklus pengambilan dan siklus eksekusi) diperlukan untuk menambahkan konten lokasi 940 ke konten 941. Beberapa prosesor yang lebih tua, misalnya, menyertakan instruksi yang berisi lebih dari satu alamat memori. Dengan demikian, siklus eksekusi untuk instruksi tertentu pada prosesor tersebut dapat melibatkan lebih dari satu referensi ke memori. Selain itu, alih-alih referensi memori, instruksi dapat menentukan operasi I/O.

Misalnya, prosesor PDP-11 menyertakan instruksi yang dinyatakan secara simbolis sebagai:

ADD B, A

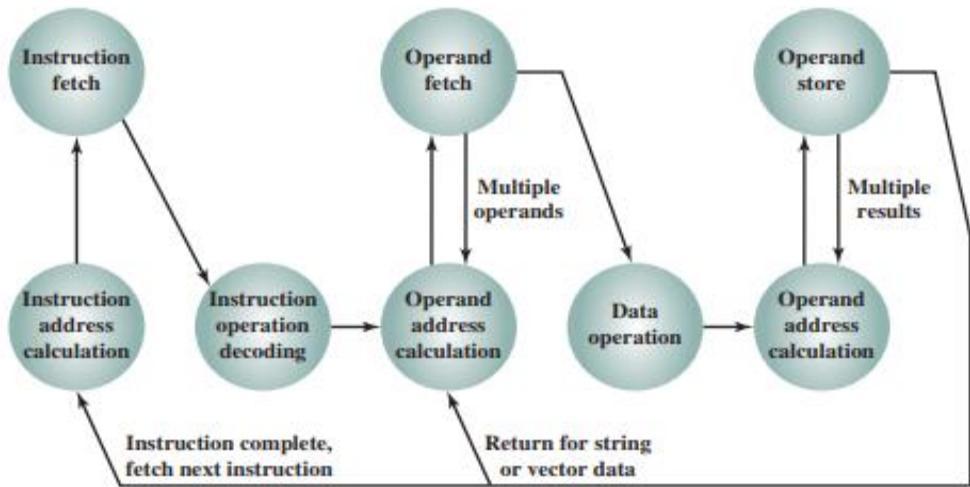
yang menyimpan jumlah isi lokasi memori B dan A, ke lokasi memori A. Siklus instruksi tunggal dengan langkah-langkah berikut terjadi:

- Ambil instruksi ADD.
- Baca konten lokasi memori A ke dalam prosesor.
- Baca konten lokasi memori B ke dalam prosesor. Agar isi A tidak hilang, prosesor harus memiliki setidaknya dua register untuk menyimpan nilai memori, daripada satu akumulator.
- Tambahkan dua nilai.
- Tulis hasil dari prosesor ke lokasi memori A

Dengan demikian, siklus eksekusi untuk instruksi tertentu dapat melibatkan lebih dari satu referensi ke memori. Selain itu, alih-alih referensi memori, instruksi dapat menentukan operasi I/O. Dengan pertimbangan tambahan ini, Gambar 1.20 memberikan pandangan yang lebih rinci pada siklus instruksi dasar pada Gambar 1.17.

Gambar 1.20 disajikan dalam bentuk state diagram. Untuk setiap siklus instruksi yang diberikan, beberapa state mungkin nol dan yang lain dapat dikunjungi lebih dari satu kali. Masing-masing state dapat dijelaskan sebagai berikut:

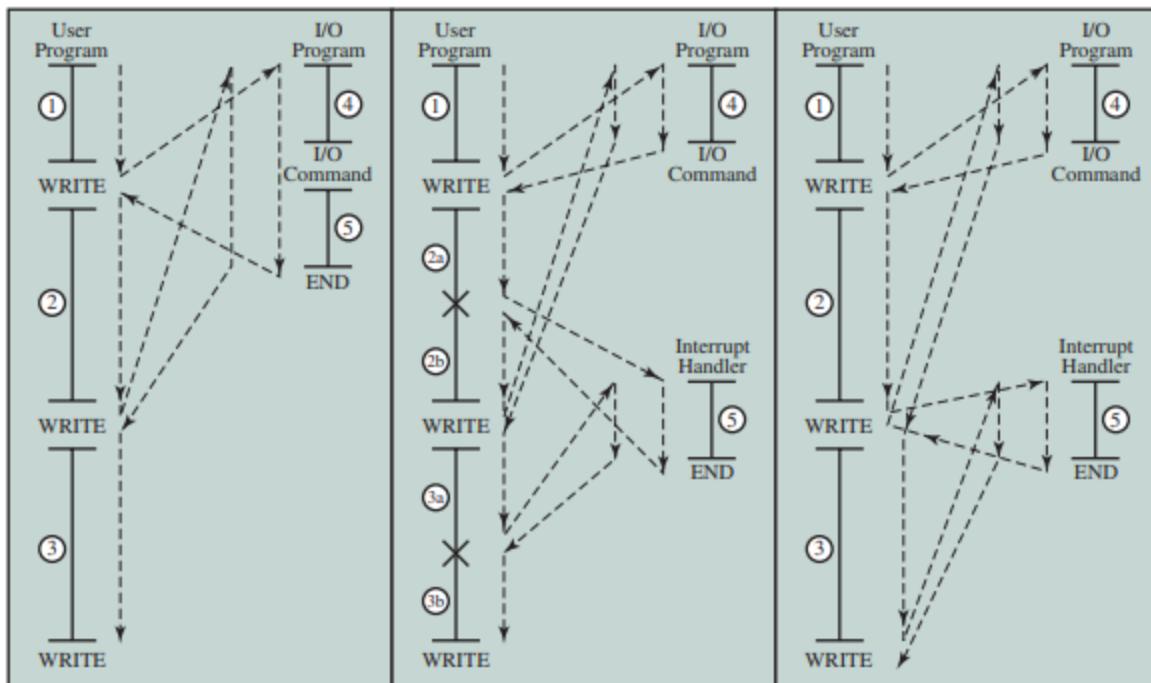
- *Instruction address calculate (iac)*: Menentukan alamat instruksi selanjutnya yang akan dieksekusi. Biasanya, ini melibatkan penambahan nomor tetap ke alamat instruksi sebelumnya. Misalnya, jika setiap instruksi panjangnya 16 bit dan memori disusun menjadi words 16-bit, maka tambahkan 1 ke alamat sebelumnya. Jika, alih-alih, memori diatur sebagai byte 8-bit yang dapat dialamatkan secara individual, kemudian tambahkan 2 ke alamat sebelumnya.
- *Instruction fetch (if)*: Baca instruksi dari lokasi memorinya ke dalam prosesor.
- *Instruction operation decoding (iod)*: Menganalisis instruksi untuk menentukan jenis operasi yang akan dilakukan dan *operand* yang akan digunakan.
- *Operand address calculation (oac)*: Jika operasi melibatkan referensi ke *operand* dalam memori atau tersedia melalui I/O, maka tentukan alamat *operand*.
- *Operand fetch (of)*: Mengambil *operand* dari memori atau membacanya dari I/O.
- *Data operation (do)*: Lakukan operasi yang ditunjukkan dalam instruksi.
- *Operand store (os)*: Tulis hasilnya ke dalam memori atau keluar ke I/O.



Sumber: (Stallings, 2016)

Gambar 1.20. State Diagram Siklus Instruksi

State di bagian atas (if, of, dan os) pada Gambar 1.20 melibatkan pertukaran antara prosesor dan memori atau modul I/O. State di bagian bawah diagram hanya melibatkan operasi prosesor internal. Keadaan oac muncul dua kali, karena instruksi dapat melibatkan membaca, menulis, atau keduanya. Namun, tindakan yang dilakukan selama keadaan itu secara fundamental sama dalam kedua kasus, dan hanya diperlukan satu state pengenal. Pada gambar tersebut juga memungkinkan untuk beberapa *operand* dan beberapa hasil, karena beberapa instruksi pada beberapa mesin memerlukan ini. Sebagai contoh, instruksi PDP-11 ADD A, B menghasilkan urutan status berikut: iac, if, iod, oac, dari, oac, dari, do, oac, os. Akhirnya, pada beberapa mesin, instruksi tunggal dapat menentukan operasi yang akan dilakukan pada vektor (array satu dimensi) angka atau string (array satu dimensi) karakter.



(a) *No interrupt*

(b) *Interrupt, short I/O wait*

(c) *Interrupt, long I/O wait*

Sumber: (Stallings, 2016)

Gambar 1.21. Arah Kendali Program Tanpa dan Dengan Interupsi

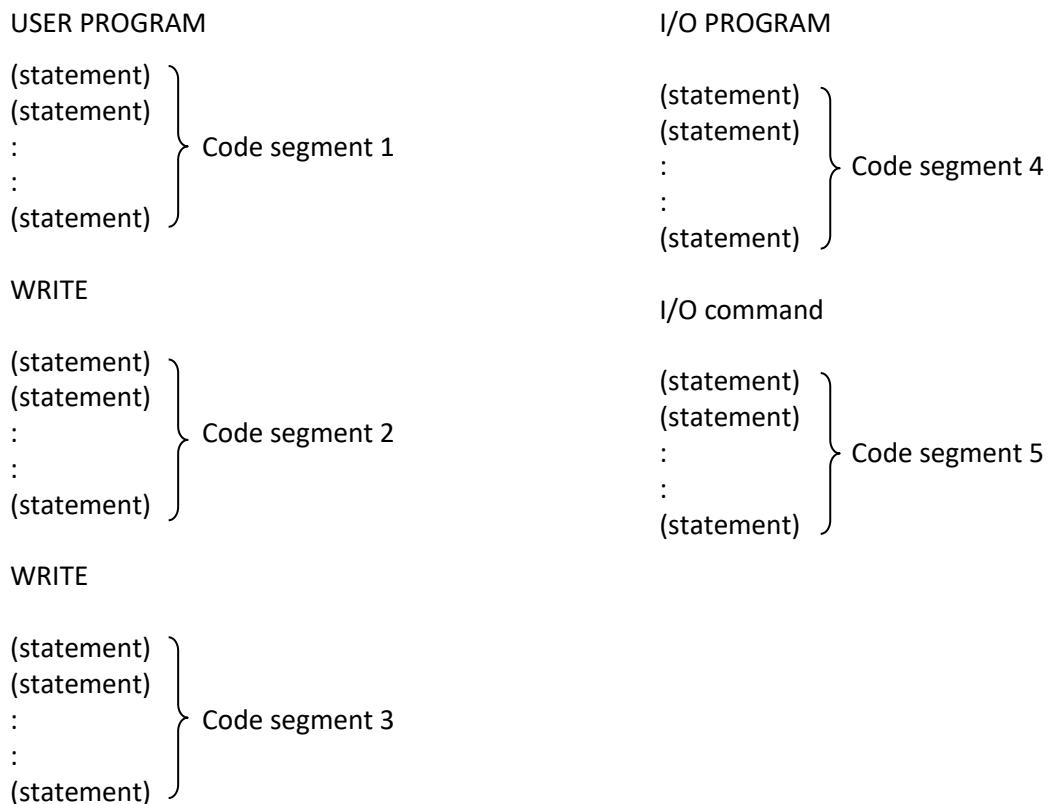
Karena operasi I/O mungkin membutuhkan waktu yang relatif lama untuk diselesaikan, program I/O digantung menunggu operasi selesai; karenanya, *user program* dihentikan pada titik panggilan WRITE untuk beberapa periode waktu yang cukup lama.

B. Interupsi dan Siklus Instruksi

Dengan interupsi, prosesor dapat terlibat dalam menjalankan instruksi lain saat operasi I/O sedang berlangsung. Pertimbangkan aliran kontrol pada Gambar 1.21b. Seperti sebelumnya, *user program* mencapai titik di mana ia membuat panggilan sistem dalam bentuk panggilan WRITE. Program I/O yang dipanggil dalam kasus ini hanya terdiri dari kode persiapan dan perintah I/O yang sebenarnya. Setelah beberapa instruksi ini dijalankan, kontrol kembali ke *user program*. Sementara itu, perangkat eksternal sibuk menerima data dari memori komputer dan mencetaknya. Operasi I/O ini dilakukan bersamaan dengan pelaksanaan instruksi dalam *user program*.

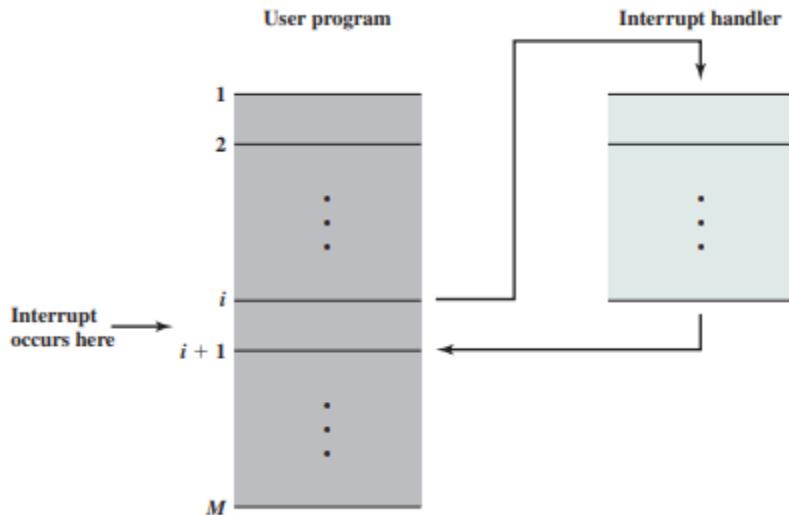
Ketika perangkat eksternal menjadi siap untuk dilayani, yaitu, ketika siap untuk menerima lebih banyak data dari prosesor, modul I/O untuk perangkat eksternal tersebut mengirimkan sinyal permintaan interupsi ke prosesor. Prosesor merespons dengan menangguhkan pengoperasian program saat ini, bercabang ke program untuk melayani perangkat I/O tertentu, yang dikenal sebagai *interrupt handler*, dan melanjutkan eksekusi asli setelah perangkat dilayani. Titik-titik di mana interupsi tersebut terjadi ditunjukkan oleh tanda silang pada Gambar 1.21b.

Untuk memperjelas apa yang terjadi pada Gambar 1.21, berikut adalah user program yang berisi dua perintah WRITE. Ada segmen kode di awal, lalu satu perintah WRITE, lalu segmen kode kedua, lalu perintah WRITE kedua, lalu segmen kode ketiga dan terakhir. Perintah WRITE memanggil program I/O yang disediakan oleh OS. Demikian pula, program I/O terdiri dari segmen kode, diikuti oleh perintah I/O, diikuti oleh segmen kode lainnya. Perintah I/O memanggil operasi I/O perangkat keras.



Dari sudut pandang *user program*, interupsi berupa Interupsi dari urutan eksekusi normal. Ketika pemrosesan interupsi selesai, eksekusi dilanjutkan (Gambar 1.22). Dengan demikian, *user program* tidak harus mengandung kode khusus untuk mengakomodasi interupsi; prosesor dan sistem operasi

bertanggung jawab untuk menangguhkan *user program* dan kemudian melanjutkannya pada titik yang sama.

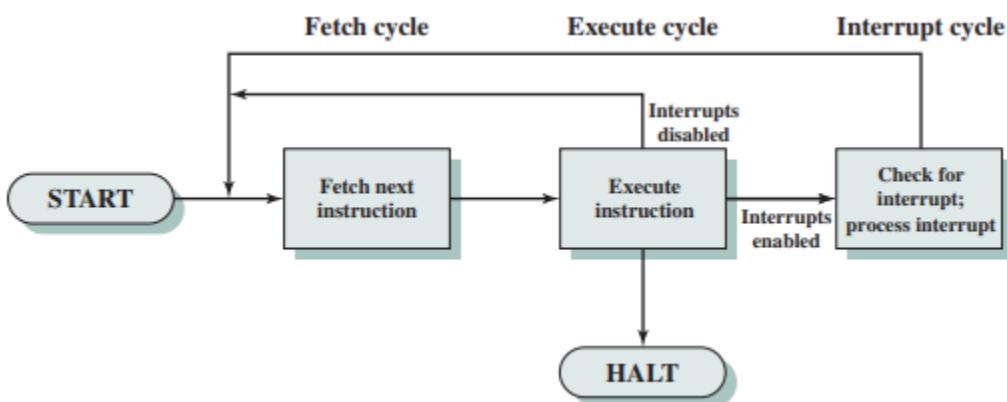


Sumber: (Stallings, 2016)

Gambar 1.22. Pengalihan kendali melalui Interupsi

Untuk mengakomodasi interupsi, siklus interupsi ditambahkan ke siklus instruksi, seperti yang ditunjukkan pada Gambar 1.23. Dalam siklus interupsi, prosesor memeriksa untuk melihat apakah ada interupsi yang terjadi, yang ditunjukkan oleh adanya sinyal interupsi. Jika tidak ada interupsi yang tertunda, prosesor melanjutkan ke siklus pengambilan dan mengambil instruksi selanjutnya dari program saat ini. Jika interupsi tertunda, prosesor akan melakukan hal berikut:

- Menunda eksekusi program saat ini dieksekusi dan menyimpan konteksnya. Ini berarti menyimpan alamat instruksi berikutnya yang akan dieksekusi (isi saat ini dari PC) dan data lain yang relevan dengan aktivitas prosesor saat ini.
- Mengatur **PC** ke alamat awal *routine interrupt handler*.

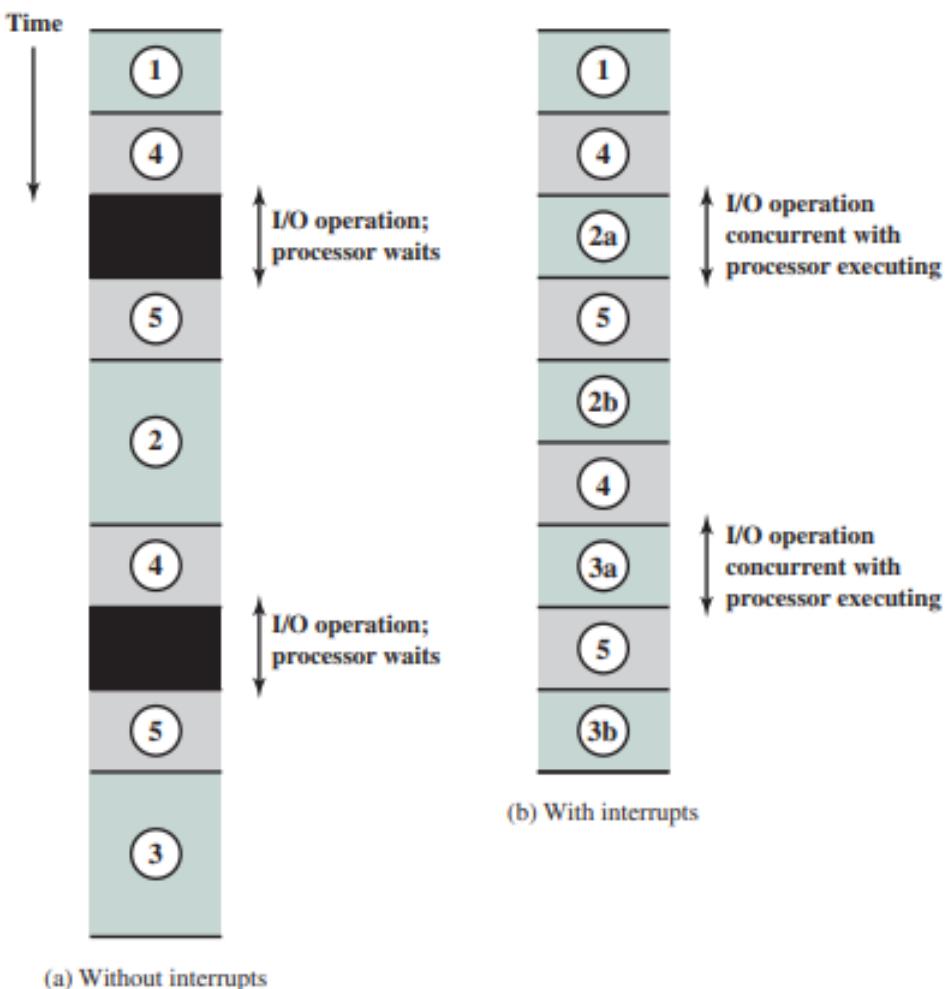


Sumber: (Stallings, 2016)

Gambar 1.23. Siklus Instruksi dengan Interupsi

Prosesor sekarang melanjutkan ke siklus pengambilan dan mengambil instruksi pertama dalam program pengendali interupsi, yang akan melayani interupsi. Program interrupt handler umumnya merupakan bagian dari sistem operasi. Biasanya, program ini menentukan sifat interupsi dan melakukan tindakan apa pun yang diperlukan. Dalam contoh digunakan, *interrupt handler* menentukan modul I/O mana yang menghasilkan interupsi dan dapat bercabang ke program yang

akan menulis lebih banyak data ke modul I/O tersebut. Ketika *routine interrupt handler* selesai, prosesor dapat melanjutkan eksekusi *user program* pada titik interupsi. Jelas bahwa ada beberapa *overhead* yang terlibat dalam proses ini. Instruksi tambahan harus dijalankan (dalam interrupt handler) untuk menentukan sifat interupsi dan untuk memutuskan tindakan yang sesuai. Namun demikian, karena jumlah waktu yang relatif besar yang akan terbuang hanya dengan menunggu operasi I/O, prosesor dapat digunakan jauh lebih efisien dengan penggunaan interupsi. Untuk menghargai peningkatan efisiensi, perhatikan Gambar 1.24, yang merupakan diagram waktu berdasarkan aliran kontrol pada Gambar 1.24a dan 1.24b. Dalam gambar ini, code segments dari user program adalah yang diarsir paling gelap, dan segmen kode program I/O berwarna abu-abu. Gambar 1.24a menunjukkan kasus di mana interupsi tidak digunakan. Prosesor harus menunggu saat operasi I/O dilakukan.



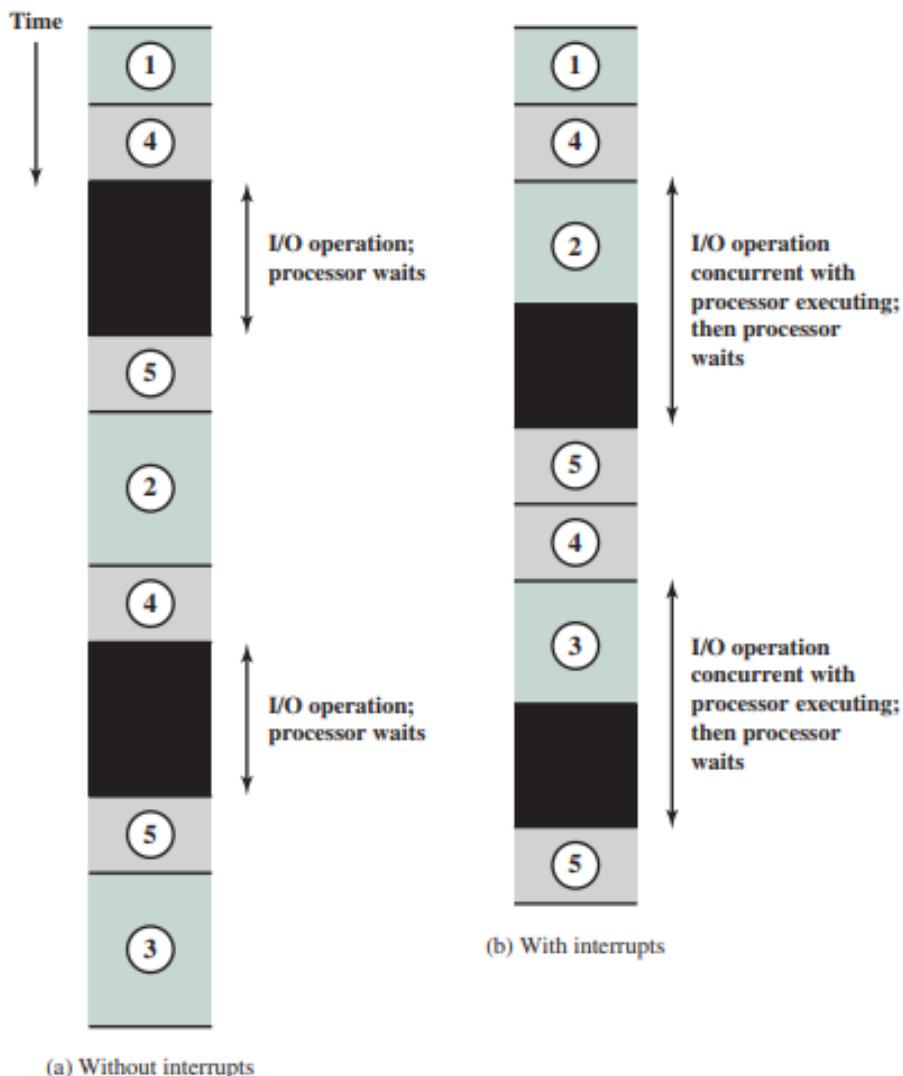
Sumber: (Stallings, 2016)

Gambar 1.24. Program Timing: Short I/O Wait

Gambar 1.24a dan 1.24b mengasumsikan bahwa waktu yang diperlukan untuk operasi I/O relatif singkat: kurang dari waktu untuk menyelesaikan eksekusi instruksi antara operasi tulis dalam *user program*. Dalam hal ini, segmen kode berlabel kode segmen 2 terputus. Sebagian kode (2a) dijalankan (saat operasi I/O dilakukan) dan kemudian interupsi terjadi (setelah selesai operasi I/O). Setelah interupsi dilayani, eksekusi dilanjutkan dengan sisa segmen kode 2 (2b).

Kasus yang lebih umum, terutama untuk perangkat yang lambat seperti printer, adalah bahwa operasi I/O akan memakan waktu lebih banyak daripada mengeksekusi urutan instruksi pengguna. Gambar 1.21c menunjukkan keadaan ini. Dalam hal ini, *user program* mencapai panggilan WRITE kedua

sebelum operasi I/O yang dihasilkan oleh panggilan pertama selesai. Hasilnya adalah bahwa *user program* ditutup pada saat itu. Ketika operasi I/O sebelumnya selesai, panggilan WRITE baru ini dapat diproses, dan operasi I/O baru dapat dimulai. Gambar 1.25 menunjukkan waktu untuk situasi ini dengan dan tanpa menggunakan interupsi. Dapat dilihat bahwa masih ada keuntungan dalam efisiensi karena sebagian waktu di mana operasi I/O sedang berjalan tumpang tindih dengan pelaksanaan instruksi pengguna. Gambar 1.26 menunjukkan state diagram siklus instruksi yang direvisi yang mencakup pemrosesan siklus interupsi.



Sumber: (Stallings, 2016)

Gambar 1.25. Program Timing: Long I/O Wait

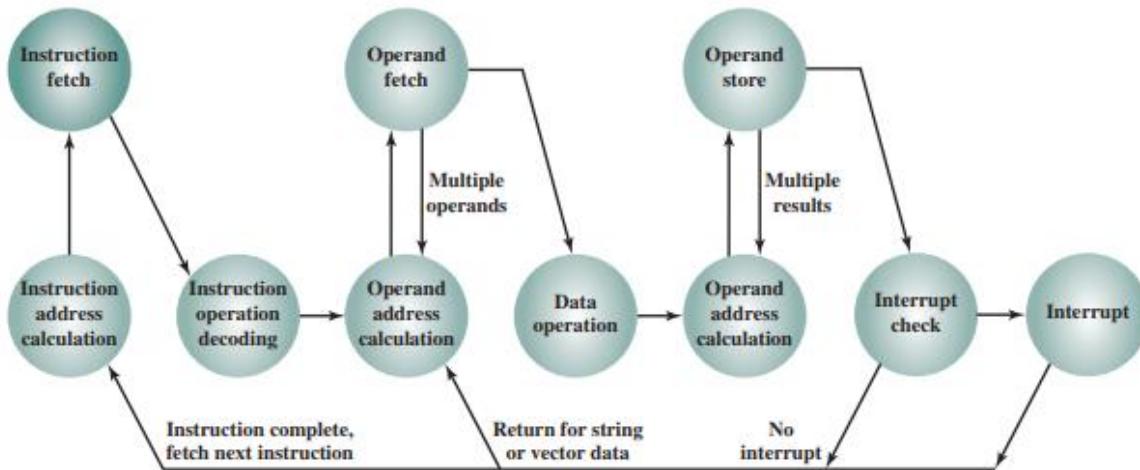


Figure 3.12 Instruction Cycle State Diagram, with Interrupts

Sumber: (Stallings, 2016)

Gambar 1.26. State Diagram Siklus Instruksi dengan Interrupt

C. Multiple Interrupt

Pada pembahasan interupsi sebelumnya hanya berfokus pada terjadinya interupsi tunggal. Namun, anggapan bahwa banyak interupsi dapat terjadi dalam waktu yang bersamaan. Misalnya, suatu program dapat menerima data dari jalur komunikasi dan hasil pencetakan. Printer akan menghasilkan interupsi setiap kali menyelesaikan operasi cetak. Pengontrol jalur komunikasi akan menghasilkan interupsi setiap kali unit data tiba. Unit dapat berupa karakter tunggal atau blok, tergantung pada sifat disiplin komunikasi. Dalam hal apa pun, interupsi komunikasi dapat terjadi saat interupsi printer sedang diproses.

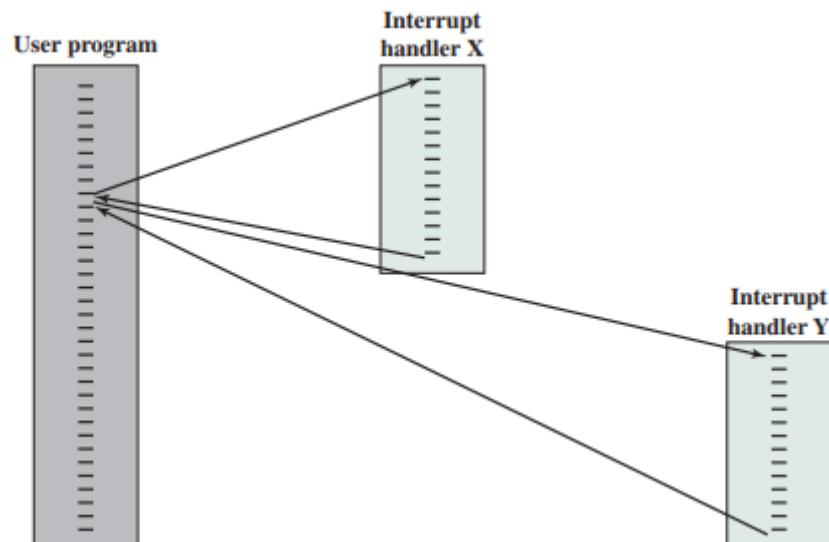
Dua pendekatan dapat diambil untuk menangani beberapa interupsi. Yang pertama adalah menonaktifkan interupsi saat interupsi sedang diproses. Interupsi yang dinonaktifkan hanya berarti bahwa prosesor dapat dan akan mengabaikan sinyal permintaan interupsi itu. Jika interupsi terjadi selama waktu ini, umumnya akan tertunda dan akan diperiksa oleh prosesor setelah prosesor mengaktifkan interupsi. Jadi, ketika *user program* mengeksekusi dan terjadi interupsi, interupsi segera dinonaktifkan. Setelah rutin interrupt handler selesai, interupsi diaktifkan sebelum melanjutkan *user program*, dan prosesor memeriksa untuk melihat apakah interupsi tambahan telah terjadi. Pendekatan ini bagus dan sederhana, karena interupsi ditangani secara berurutan yang ketat (Gambar 1.27a).

Kelemahan dari pendekatan sebelumnya adalah tidak memperhitungkan prioritas relatif atau kebutuhan kritis waktu. Misalnya, ketika *input* datang dari saluran komunikasi, mungkin perlu diserap dengan cepat untuk memberi ruang bagi lebih banyak *input*. Jika batch *input* pertama belum diproses sebelum batch kedua tiba, data mungkin hilang.

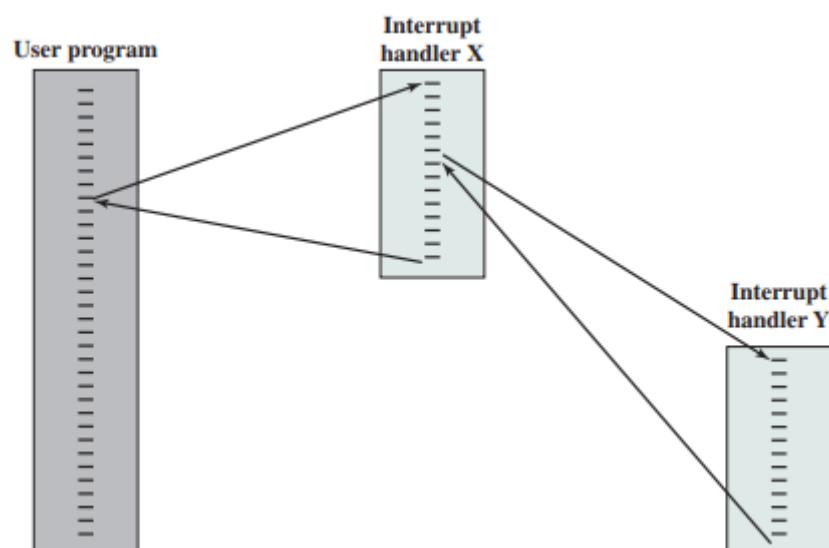
Pendekatan kedua adalah mendefinisikan prioritas untuk interupsi dan untuk memungkinkan interupsi dengan prioritas yang lebih tinggi menyebabkan interrupt handler dengan prioritas yang lebih rendah terganggu (Gambar 1.27b). Sebagai contoh dari pendekatan kedua ini, pertimbangkan sebuah sistem dengan tiga perangkat I/O: printer, disk, dan jalur komunikasi, dengan prioritas yang meningkat masing-masing 2, 4, dan 5.

Gambar 1.28 menggambarkan urutan yang mungkin. *user program* dimulai pada $t = 0$. Pada $t = 10$, interupsi printer terjadi; informasi pengguna ditempatkan pada *stack* sistem dan eksekusi berlanjut

pada *Interrupt Service Routine* (ISR) printer. Sementara rutin ini masih dijalankan, pada $t = 15$, interupsi komunikasi terjadi. Karena jalur komunikasi memiliki prioritas lebih tinggi daripada printer, interupsi itu dihormati. ISR printer terganggu, kondisinya didorong ke *stack*, dan eksekusi berlanjut di ISR komunikasi. Ketika rutin ini sedang dijalankan, sebuah interupsi disk terjadi ($t = 20$). Karena ganggu interupsi ini adalah prioritas yang lebih rendah, itu hanya diadakan, dan komunikasi ISR berjalan hingga selesai. Saat ISR komunikasi selesai ($t = 25$), status prosesor sebelumnya dipulihkan, yang merupakan eksekusi ISR printer. Namun, bahkan sebelum satu instruksi tunggal dalam rutinitas itu dapat dieksekusi, prosesor menghargai interupsi disk dengan prioritas lebih tinggi dan mengontrol transfer ke disk ISR. Hanya ketika rutin itu selesai ($t = 35$) adalah printer ISR dilanjutkan. Ketika rutin itu selesai ($t = 40$), kontrol akhirnya kembali ke *user program*.



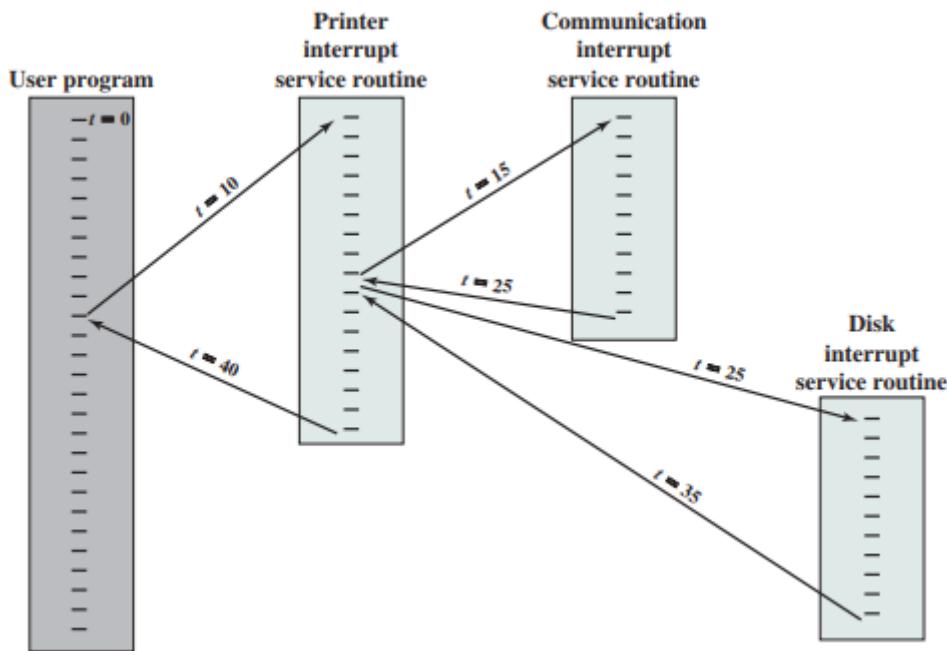
(a) *Sequential interrupt processing*



(b) *Nested interrupt processing*

Sumber: (Stallings, 2016)

Gambar 1.27. Transfer Pengendalian dengan *Multiple Interrupt*



Sumber: (Stallings, 2016)

Gambar 1.28. Contoh Urutan waktu dari Multiple Interrupts

Modul I/O (misalnya Pengontrol disk) dapat bertukar data secara langsung dengan prosesor. Sama seperti prosesor dapat memulai membaca atau menulis dengan memori, menunjuk alamat lokasi tertentu, prosesor juga dapat membaca data dari atau menulis data ke modul I/O. Dalam kasus terakhir ini, prosesor mengidentifikasi perangkat tertentu yang dikendalikan oleh modul I/O tertentu. Dengan demikian, urutan instruksi serupa pada Gambar 1.28 dapat terjadi, dengan instruksi I/O daripada instruksi referensi memori.

Dalam beberapa kasus, diinginkan untuk memungkinkan pertukaran I/O terjadi langsung dengan memori. Dalam kasus seperti itu, prosesor memberikan wewenang kepada modul I/O untuk membaca atau menulis ke memori, sehingga transfer memori I/O dapat terjadi tanpa mengikat prosesor. Selama transfer semacam itu, modul I/O mengeluarkan perintah membaca atau menulis ke memori, membebaskan prosesor dari tanggung jawab untuk pertukaran. Operasi ini dikenal sebagai akses memori langsung (DMA)

D. Fungsi I/O

Modul I/O (misalnya, pengontrol disk) dapat bertukar data secara langsung dengan prosesor. Sama seperti prosesor dapat memulai membaca atau menulis dengan memori, menunjuk alamat lokasi tertentu, prosesor juga dapat membaca data dari atau menulis data ke modul I/O. Dalam kasus terakhir ini, prosesor mengidentifikasi perangkat tertentu yang dikontrol oleh modul I/O tertentu. Dengan demikian, urutan instruksi yang mirip dalam bentuk dengan Gambar 1.28 dapat terjadi, dengan instruksi I/O daripada instruksi referensi memori.

Dalam beberapa kasus, diinginkan untuk mengizinkan pertukaran I/O terjadi secara langsung dengan memori. Dalam kasus seperti itu, prosesor memberi modul I/O hak kendali untuk membaca dari atau menulis ke memori, sehingga transfer memori I/O dapat terjadi tanpa melibatkan prosesor. Selama transfer tersebut, modul I/O mengeluarkan perintah baca atau tulis ke memori, membebaskan tanggung jawab prosesor untuk pertukaran tersebut. Operasi ini dikenal sebagai akses memori langsung (DMA).

2. Internal Memori

2.1. Hirarki Memori

Kinerja pada memori komputer dapat dilihat dari tiga karakteristik utama yaitu: kapasitas, kecepatan, dan harga. Seperti yang mungkin diharapkan, ada *trade-off* di antara tiga karakteristik tersebut. Berbagai teknologi digunakan untuk mengimplementasikan sistem memori, namun di seluruh spektrum teknologi, berlaku hal berikut:

- Waktu akses lebih cepat, biaya per bit lebih besar.
- Kapasitas lebih besar, biaya per bit lebih kecil.
- Kapasitas lebih besar, waktu akses lebih lambat.

Perancang ingin menggunakan teknologi memori yang menyediakan berkapasitas besar, baik karena kapasitas diperlukan dan karena biaya per bit rendah. Namun, untuk memenuhi persyaratan kinerja, perancang harus menggunakan memori yang mahal dan berkapasitas lebih rendah dengan waktu akses yang singkat. Jalan keluar dari dilema ini adalah tidak bergantung pada komponen atau teknologi memori tunggal, tetapi menggunakan hierarki memori. Hirarki tipikal diilustrasikan pada Gambar 2.1. Jika dilihat dari atas kebawah, maka berlaku hal berikut:

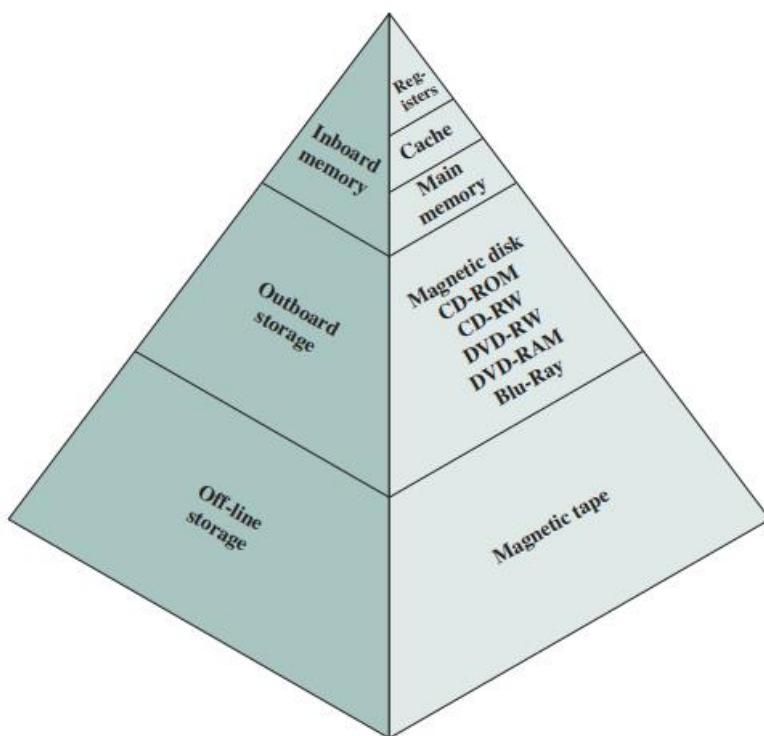
- Biaya per bit berkurang;
- Kapasitas meningkat;
- Waktu akses meningkat;
- Berkurangnya frekuensi akses memori oleh prosesor.
- Jadi, memori yang lebih kecil, lebih mahal, serta lebih cepat, dilengkapi dengan memori yang lebih besar, lebih murah, serta lebih lambat. Kunci keberhasilan organisasi ini adalah: mengurangi frekuensi akses terhadap memori..

Memori dalam sistem komputer modern bukanlah satu perangkat monolitik (kumpulan perangkat serupa), namun merupakan kumpulan dari banyak perangkat memori berbeda dengan karakteristik fisik dan mode operasi yang berbeda. Memori dalam sistem komputer tersebut ada yang lebih besar, dan lebih kecil (baik dari segi ukuran fisik dan kapasitas penyimpanan); beberapa lebih cepat, dan beberapa lebih lambat; beberapa lebih murah, dan beberapa lebih mahal. Memori sistem komputer dibangun dengan cara ini karena tidak ada perangkat memori yang memiliki semua karakteristik yang dianggap ideal. Setiap jenis teknologi memori yang tersedia memiliki keunggulan atau aspek tertentu di mana ia lebih unggul dari teknologi lain, tetapi juga beberapa kekurangan yang membuatnya kurang ideal. Dengan menggabungkan berbagai jenis perangkat memori dalam satu sistem secara cerdas, diberharap dapat memperoleh keuntungan dari masing-masing jenis sekaligus meminimalkan kerugiannya.

Setiap jenis memori memiliki kekurangan dan kelebihan tertentu. Oleh karena itu, campuran berbagai jenis perangkat memori dalam suatu sistem digunakan untuk menukar keuntungan dan kerugian masing-masing teknologi, memaksimalkan keuntungan tertentu dari setiap jenis memori, serta

meminimalkan, atau setidaknya menutupi kekurangannya. Dengan cara ini, keseluruhan sistem memori dapat mendekati sistem ideal: kapasitas besar, padat, cepat, kemampuan baca/tulis, dan murah dengan setidaknya beberapa bagian dapat dilepas dan bagian penting tidak mudah berubah. Solusi tipikal adalah desain sistem komputer di mana hierarki subsistem memori terdiri dari beberapa jenis perangkat. Konsep umum digambarkan pada Gambar 2.1, hierarki memori yang berbentuk piramida. Dari bawah ke atas, memori memiliki karakter sebagai berikut:

- Peningkatan waktu akses (*access time*) memori (semakin ke bawah semakin lambat, semakin ke atas semakin cepat)
- Peningkatan kapasitas (semakin ke bawah semakin besar, semakin ke atas semakin kecil)
- Peningkatan jarak dengan prosesor (semakin ke bawah semakin jauh, semakin ke atas semakin dekat)
- Penurunan harga memori tiap bitnya (semakin ke bawah semakin semakin murah, semakin ke atas semakin mahal)



Sumber: (Stallings, 2016)

Gambar 2.1. Hirarki Memori pada Sistem Komputer

Memori yang lebih kecil, lebih mahal dan lebih cepat diletakkan pada urutan teratas. Sehingga, jika diurutkan dari yang tercepat, maka urutannya adalah sebagai berikut:

1. Register mikroprosesor. Ukurannya yang paling kecil tetapi memiliki waktu akses yang paling cepat, umumnya hanya 1 siklus CPU saja.
2. Cache mikroprosesor, yang disusun berdasarkan kedekatannya dengan prosesor (level-1, level-2, level-3, dan seterusnya). Memori *cache* mikroprosesor dikelaskan ke dalam tingkatan-tingkatannya sendiri:

level-1: memiliki ukuran paling kecil di antara semua *cache*, sekitar puluhan kilobyte saja. Kecepatannya paling cepat di antara semua *cache*.

level-2: memiliki ukuran yang lebih besar dibandingkan dengan *cache* level-1, yakni sekitar 64 *kilobyte*, 256 *kilobyte*, 512 *kilobyte*, 1024 *kilobyte*, atau lebih besar. Meski demikian, kecepatannya lebih lambat dibandingkan dengan level-1, dengan nilai *latency* kira-kira 2 kali hingga 10 kali. *Cache* level-2 ini bersifat opsional. Beberapa prosesor murah dan prosesor sebelum Intel Pentium tidak memiliki *cache* level-2.

level-3: memiliki ukuran yang lebih besar dibandingkan dengan *cache* level-2, yakni sekitar beberapa *megabyte* tetapi agak lambat. *Cache* ini bersifat opsional. Umumnya digunakan pada prosesor-prosesor server dan *workstation*. Beberapa prosesor *desktop* juga menawarkan *cache* level-3, meski dimbangi dengan harga yang sangat tinggi.

3. Memori utama: memiliki akses yang jauh lebih lambat dibandingkan dengan memori *cache*, dengan waktu akses hingga beberapa ratus siklus CPU, tetapi ukurannya mencapai satuan *gigabyte*. Waktu akses pun kadang-kadang tidak seragam, khususnya dalam kasus mesin-mesin Non-uniform *memory access*.
4. *Outboard storage*: yang terdiri dari memori cakram magnetik.
5. *Offline storage*, misalnya magnetic tape

Data disimpan lebih permanen di perangkat penyimpanan massal eksternal, yang paling umum adalah hard disk dan media yang dapat dilepas, seperti *removable magnetic disk*, *tape*, dan penyimpanan optik. Memori eksternal dan non-volatile juga disebut sebagai memori sekunder atau memori tambahan. Disk juga digunakan untuk memberikan ekstensi ke memori utama yang dikenal sebagai memori virtual.

2.2. Karakteristik Memori Sistem Komputer

Memori komputer menjadi lebih mudah dipelajari jika diklasifikasikan berdasarkan karakteristiknya. Tabel 2.1. merangkum klasifikasi sistem memori berdasarkan delapan karakteristik utama.

Tabel 2.1. Klasifikasi Karakteristik Memori Sistem Komputer

Location	Performance
<i>Internal</i> (misal: register prosesor, <i>cache</i> , <i>main memory</i>)	<i>Access time</i> <i>Cycle time</i>
<i>External</i> (Misal: optical disk, magnetic disk, tapes)	<i>Transfer time</i>
Capacity	Physical Type
<i>Number of words</i>	<i>Semiconductor</i>
<i>Number of bytes</i>	<i>Magnetic</i> <i>Optical</i> <i>Magneto-optical</i>
Unit of Transfer	Physical Characteristics
<i>Word</i>	<i>Volatile/nonvolatile</i>
<i>Block</i>	<i>Erasable/nonerasable</i>
Access Method	Organization
<i>Sequencial</i>	<i>Memory modules</i>
<i>Direct</i>	
<i>Random</i>	
<i>Associative</i>	

Sumber: (Stallings, 2016)

A. Location

Memori internal sering disamakan dengan memori utama, tetapi ada bentuk lain dari memori internal. Prosesor membutuhkan memori lokalnya sendiri, dalam bentuk register. Bahkan *Control Unit* juga mungkin memerlukan memori internalnya sendiri. *Cache* adalah bentuk lain dari memori internal. Memori eksternal terdiri dari perangkat penyimpanan periferal, seperti disk dan pita, yang dapat diakses oleh prosesor melalui pengontrol I/O.

B. Capacity

Karakteristik memori yang jelas adalah kapasitasnya. Untuk memori internal, ini biasanya dinyatakan dalam bentuk *byte* (1 *byte* = 8 bit) atau *words*. Panjang *word* yang umum adalah 8, 16, dan 32 bit (kelipatan 8). Kapasitas memori eksternal biasanya dinyatakan dalam bentuk *byte*. Ukuran yang umum adalah *Gigabyte* dan *Terabyte*. Tabel 2.2 menunjukkan unit untuk kapasitas memori.

Tabel 2.2. Unit dari Kapasitas Memori

Unit Kapasitas Memori	Ukuran dalam Bit
<i>Nible</i>	4 bit
<i>Byte</i>	8 bit
KB (Kilo Byte)	2^{10} Byte (1024 Byte)
MB (Mega Byte)	2^{20} Byte
GB (Giga Byte)	2^{30} Byte
TB (Tera Byte)	2^{40} Byte
PB (Peta Byte)	2^{50} Byte
EB (Exa Byte)	2^{60} Byte
ZB (Zetta Byte)	2^{70} Byte
YB (Yotta Byte)	2^{80} Byte

Sumber: (Stallings, 2016)

C. Unit of Transfer

Konsep terkait adalah unit transfer. Untuk memori internal, unit transfer sama dengan jumlah saluran listrik yang masuk dan keluar dari modul memori. Ini mungkin sama dengan panjang *word*, tetapi seringkali lebih besar, seperti 64, 128, atau 256 bit. Untuk memperjelas poin ini, pertimbangkan tiga konsep terkait untuk memori internal:

- 1) *Word*: Unit organisasi memori yang "alami". Ukuran *word* biasanya sama dengan jumlah bit yang digunakan untuk mewakili integer dan panjang instruksi. Sayangnya, ada banyak pengecualian. Misalnya, CRAY C90 (model lama komputer super CRAY) memiliki panjang *word* 64-bit tetapi menggunakan representasi integer 46-bit. Arsitektur Intel x86 memiliki berbagai panjang instruksi, dinyatakan sebagai kelipatan *byte*, dan ukuran *word* 32 bit.
- 2) Unit yang dapat dialamatkan (*addressable unit*): Dalam beberapa sistem, *addressable unit* adalah *word*. Namun, banyak sistem memungkinkan pengalaman pada level *byte*. Hubungan antara panjang dalam bit dari bidang alamat A dan jumlah *Addressable unit* N adalah $2^A = N$. Misalnya jika bidang alamat memiliki panjang 21 bit, maka terdapat $2^{21} = 2$ Mega *addressable unit*.
- 3) *Transfer unit*: Untuk memori utama, *transfer unit* adalah jumlah bit yang dibaca dari atau ditulis ke dalam memori sekaligus. *Transfer unit* tidak perlu sama dengan *word* atau *addressable unit* yang bisa dialamatkan. Untuk memori eksternal, data sering ditransfer dalam unit yang jauh lebih besar daripada *word*, yang disebut sebagai *block*.

D. Metode Akses (Access Method)

Perbedaan lain di antara jenis memori adalah metode mengakses unit data. Metode akses memori pada umumnya bergantung pada bentuk fisiknya. Berikut adalah metode akses memori sesuai dengan fisiknya:

- 1) **Sequential access.** Memori diatur ke dalam unit data, yang disebut record. Akses dilakukan dalam urutan linier tertentu. Informasi pengalaman yang tersimpan digunakan untuk memisahkan record dan membantu dalam proses pencarian. Mekanisme baca-tulis digunakan secara bersama, dengan cara berjalan menuju lokasi yang diinginkan untuk mendapatkan record yang diinginkan. Contoh sequential access adalah pita. Waktu untuk mengakses record pada pita sangat bervariasi tergantung letak recordnya.
- 2) **Direct access.** Seperti halnya akses sekuensial, akses langsung menggunakan mekanisme baca-tulis bersama, setiap blok atau record memiliki alamat unik berdasarkan lokasi fisik. Akses dicapai secara langsung terhadap area *general vicinity* untuk mencapai lokasi akhir. Waktu aksesnya bervariasi. Contoh *direct access* adalah disk..
- 3) **Random access.** Setiap lokasi dalam memori dapat diakses secara random, serta dialamat secara langsung. Waktu untuk mengakses lokasi tertentu tidak tergantung pada urutan akses sebelumnya dan konstan. Dengan demikian, setiap lokasi dapat dipilih secara acak dan langsung dialamatkan dan diakses. *Main memory* dan beberapa sistem *cache* adalah contoh untuk *random access*.
- 4) **Asosiatif access.** Ini adalah jenis random access memori yang memungkinkan untuk membuat perbandingan lokasi bit yang diinginkan dalam sebuah *word* untuk kecocokan yang ditentukan, dan untuk melakukan ini untuk semua *word* secara bersamaan. Dengan demikian, sebuah *word* diambil berdasarkan sebagian dari isinya daripada alamatnya. Seperti halnya memori akses acak biasa, setiap lokasi memiliki mekanisme pengalamatannya sendiri, dan waktu pengambilan konstan tanpa tergantung pada lokasi atau pola akses sebelumnya. Memori *cache* dapat menggunakan akses asosiatif.

E. Performance

Dari sudut pandang pengguna, dua karakteristik memori yang paling penting adalah kapasitas dan kinerja. Tiga parameter kinerja yang digunakan adalah:

- 1) Waktu akses (*latensi*): Untuk memori akses-acak, ini adalah waktu yang diperlukan untuk melakukan operasi baca atau tulis, yaitu, waktu dari saat alamat disajikan ke memori hingga saat data telah disimpan atau tersedia untuk digunakan. Untuk memori non-akses acak, waktu akses adalah waktu yang diperlukan untuk memosisikan mekanisme baca-tulis di lokasi yang diinginkan.
- 2) Waktu siklus memori: Konsep ini terutama diterapkan pada memori akses-acak, yang terdiri dari waktu akses ditambah waktu tambahan yang diperlukan sebelum akses kedua dapat dimulai. Waktu tambahan ini mungkin diperlukan agar transien mati pada garis sinyal atau untuk membuat ulang data jika dibaca secara destruktif. Waktu siklus memori berkaitan dengan bus sistem, bukan prosesor.
- 3) Laju transfer (*transfer rate*): Ini adalah kecepatan di mana data dapat ditransfer ke dalam atau keluar dari unit memori. Untuk memori akses-acak, itu sama dengan $1/(waktu_siklus)$. Untuk memori non-akses acak, hubungan berikut berlaku:

$$T_n = T_A + \frac{n}{R} \quad (1)$$

dimana

T_n = Rata-rata waktu untuk membaca atau menulis n bit

T_A = Waktu akses rata-rata

n = Jumlah bit

R = Kecepatan transfer, dalam bit per detik (bps)

F. Physical Type

Berbagai jenis memori fisik telah digunakan, namun yang paling umum digunakan adalah memori semikonduktor, memori permukaan magnetik (digunakan untuk disk dan tape), serta optik dan magneto-optik.

Beberapa karakteristik fisik penyimpanan data menjadi sangat penting. Dalam memori yang tidak stabil, informasi dapat menguap (*volatile*) secara alami atau hilang ketika daya listrik dimatikan. Dalam memori yang tidak mudah menguap (*nonvolatile*), informasi yang tersimpan akan menetap tanpa kerusakan hingga sengaja diubah; tidak diperlukan daya listrik untuk menyimpan informasi. Memori permukaan magnetik adalah *nonvolatile*. Memori semikonduktor (memori pada sirkuit terintegrasi) dapat *volatile* atau *nonvolatile*. Memori semikonduktor *nonvolatile* tidak dapat diubah, kecuali dengan menghancurkan unit penyimpanannya. Memori semikonduktor jenis ini dikenal sebagai *Read Only Memory* (ROM). Karena kebutuhan, memori non-erasable yang praktis juga harus dapat bersifat *nonvolatile*.

2.3. Cache Memory

Untuk kinerja sistem memori komputer, program komputer tidak mengakses memori secara acak. Sebaliknya, sebagian besar program membatasi sebagian besar referensi memori untuk instruksi dan data ke area memori kecil, setidaknya selama rentang waktu terbatas tertentu. Perilaku program komputer ini menggunakan prinsip lokalitas referensi. Sederhananya, prinsip ini menyatakan bahwa program cenderung mengakses kode dan data yang baru saja diakses, atau yang berada di dekat kode atau data yang baru saja diakses. Prinsip ini menjelaskan mengapa memori *cache* yang relatif kecil dapat berdampak besar pada kinerja sistem memori. Suatu sistem bisa saja memiliki *cache* 0,1% atau kurang dari ukuran memori utama, namun harus dipastikan *cache* berisi 0,1% informasi valid yang kemungkinan besar akan digunakan dalam waktu dekat.

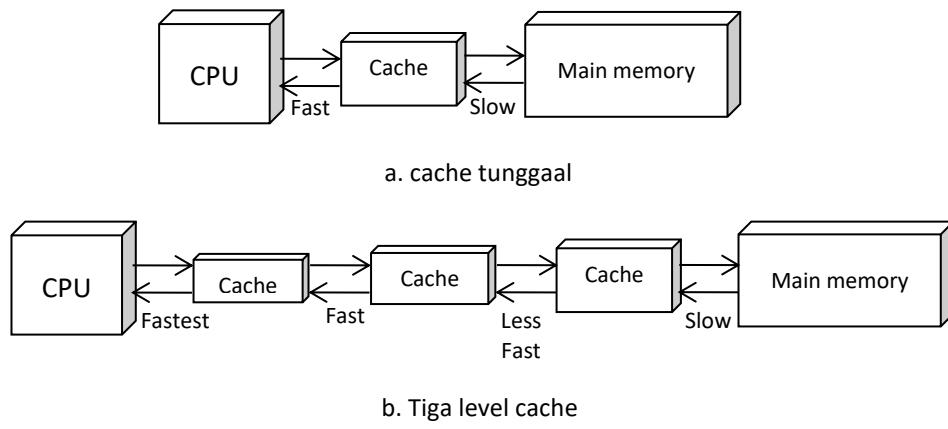
Aspek lokalitas rujukan mencakup temporal, spasial, dan sekuensial. Lokalitas temporal (terkait waktu) mengatakan bahwa jika lokasi memori tertentu diakses satu kali, ada kemungkinan besar untuk diakses kembali dalam waktu singkat. Lokalitas spasial berarti bahwa lokasi yang dekat dengan lokasi yang baru diakses juga kemungkinan besar akan dirujuk. Lokalitas sekuensial adalah bentuk spesifik dari lokalitas spasial; lokasi memori yang alamatnya secara berurutan mengikuti lokasi yang direferensikan sangat mungkin diakses dalam waktu dekat. Memori *cache* adalah teknik yang lebih umum yang dapat memanfaatkan ketiga bentuk lokalitas tersebut.

Aspek lokalitas referensi ini dengan mudah diilustrasikan oleh berbagai praktik pemrograman dan struktur data yang umum. Misalnya kode program, biasanya dijalankan secara berurutan. Penggunaan yang luas dari program *loop* dan *subrutin* berkontribusi pada lokalitas temporal. Vektor, array, string, tabel, stack, antrian, dan struktur data umum lainnya hampir selalu disimpan di lokasi memori yang berdekatan dan biasanya direferensikan dalam *loop* program. Bahkan item data skalar biasanya dikelompokkan bersama dalam blok umum atau segmen data oleh penyusun. Jelasnya, program dan kumpulan data yang berbeda menunjukkan tipe dan jumlah lokalitas yang berbeda; untungnya, hampir semua menunjukkan lokalitas sampai taraf tertentu.

Cache memory adalah memori berukuran kecil dan berkecepatan tinggi yang berfungsi untuk menyimpan sementara instruksi dan/atau data yang diperlukan oleh prosesor. Dapat dikatakan bahwa *cache memory* ini adalah memori internal prosesor. *Cache memory* dirancang untuk menggabungkan waktu akses memori yang mahal, memori berkecepatan tinggi dikombinasikan dengan ukuran memori besar yang lebih murah. Konsep tersebut diilustrasikan pada Gambar 2.2.a. Ada memori utama yang relatif besar dan lambat bersama dengan memori *cache* yang lebih kecil dan lebih cepat.

Cache memory berisi salinan bagian dari memori utama. Saat prosesor mencoba membaca *words* memori, pemeriksaan dilakukan untuk menentukan apakah *words* itu ada dalam *cache*. Jika ada, maka *words* tersebut dikirim ke prosesor. Jika tidak, blok memori utama, yang terdiri dari sejumlah *words* tetap, disalin ke dalam *cache* dan kemudian *words* tersebut dikirim ke prosesor. Karena fenomena lokalitas referensi, ketika satu blok data diambil ke dalam *cache* untuk memenuhi satu referensi memori, ada kemungkinan bahwa akan ada referensi kembali ke lokasi memori yang sama atau *words* lain dalam blok tersebut.

Gambar 2.2.b menggambarkan penggunaan beberapa level *cache*. L2 *cache* lebih lambat dan biasanya lebih besar dari *cache* L1, dan *cache* L3 lebih lambat dan biasanya lebih besar dari *cache* L2.

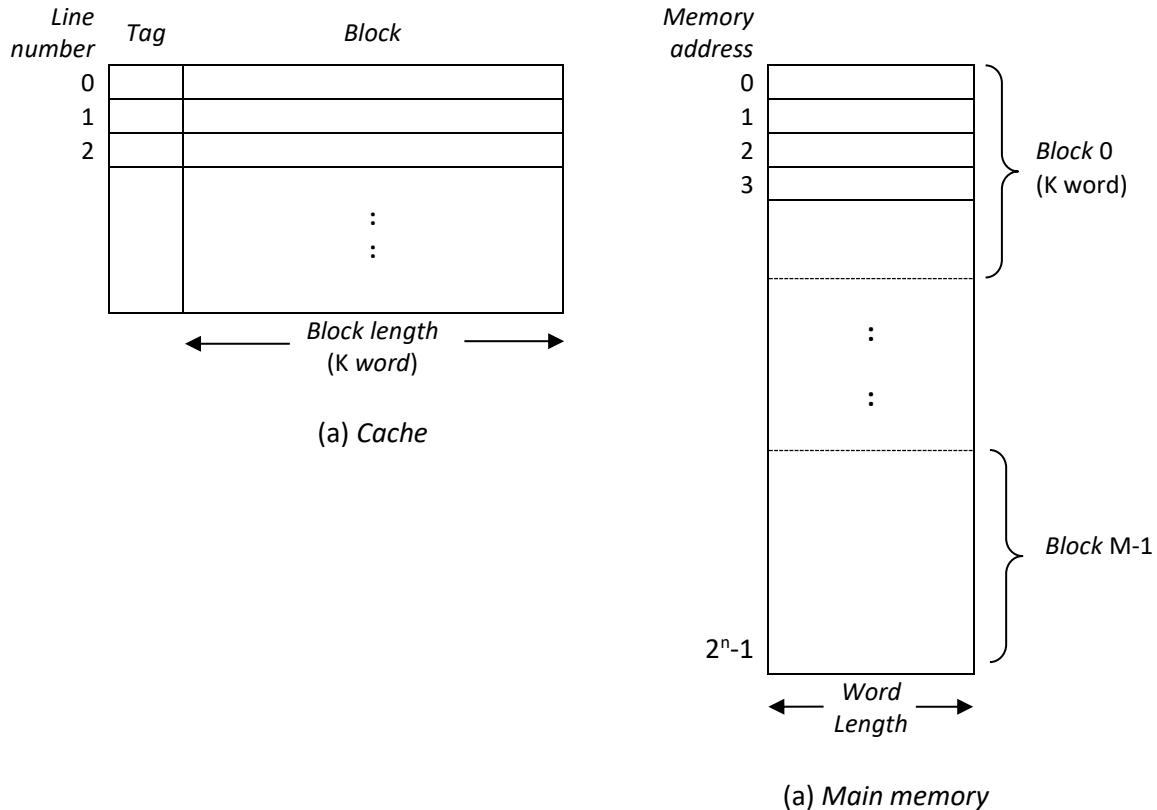


Sumber: (Stallings, 2016)

Gambar 2.2. Organisasi *Cache* dan *Main Memory*

Perbedaan arsitektur Harvard vs. von Neumann awalnya berlaku untuk memori utama. Namun, sebagian besar sistem komputer modern menerapkan arsitektur Harvard yang dimodifikasi, di mana *cache* L1 mengimplementasikan arsitektur Harvard, dan hierarki memori lainnya menerapkan arsitektur von Neumann. Oleh karena itu, dalam sistem modern, perbedaan arsitektur Harvard dan von Neumann sebagian besar diterapkan pada desain *cache* L1. Ini menjadi alasan desain *cache* terpisah juga disebut desain *cache* Harvard dan desain *cache* terpadu juga disebut von Neumann.

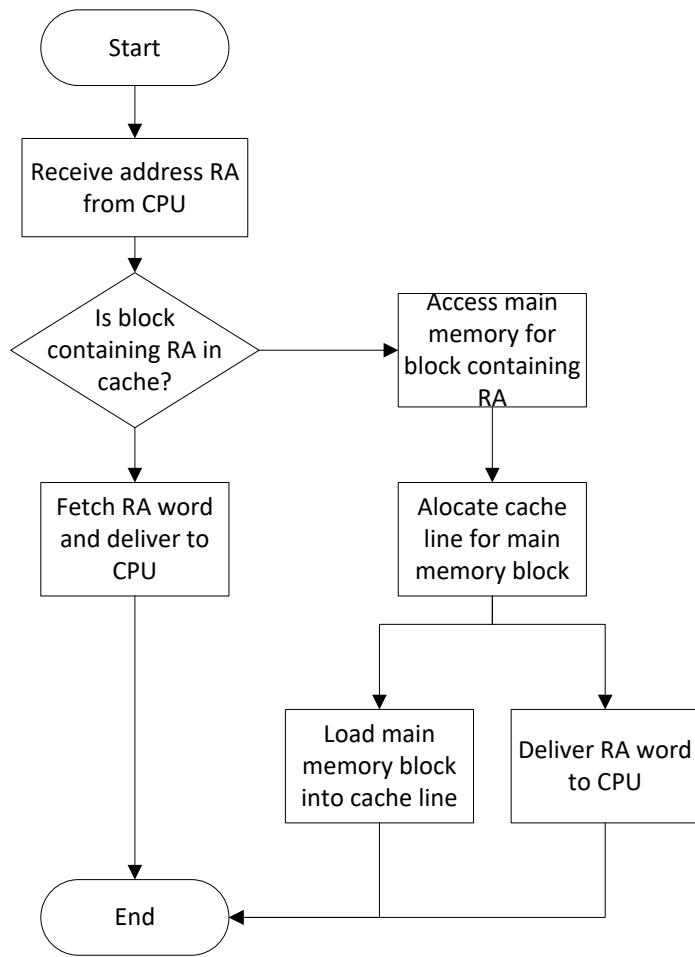
Gambar 2.3 menunjukkan struktur sistem *cache* dan memori utama. Memori utama terdiri dari hingga 2^n *word* yang dapat dialamatkan, dengan setiap *word* memiliki alamat n-bit yang unik. Untuk keperluan pemetaan, memori ini dianggap terdiri dari sejumlah blok *words* yang panjangnya masing-masing adalah K. Artinya, ada $M = 2^n/K$ blok di memori utama. *Cache* terdiri dari blok m, disebut garis. Setiap baris berisi *words* K, ditambah tag beberapa bit. Setiap baris juga termasuk bit kontrol (tidak ditampilkan), seperti bit untuk menunjukkan apakah baris telah dimodifikasi sejak dimuat ke dalam *cache*. Panjang garis, tidak termasuk tag dan bit kontrol, adalah ukuran garis. Ukuran garis mungkin sekecil 32 bit, dengan setiap "*word*" menjadi satu *byte*; dalam hal ini ukuran garis adalah 4 *byte*. Jumlah garis jauh lebih sedikit dari jumlah blok memori utama ($m < M$). Kapan saja, beberapa bagian dari blok memori berada dalam barisan dalam *cache*. Jika *word* dalam blok memori dibaca, blok itu ditransfer ke salah satu baris *cache*. Karena ada lebih banyak blok daripada garis, garis individual tidak dapat secara unik dan permanen didedikasikan untuk blok tertentu. Dengan demikian, setiap baris menyertakan tag yang mengidentifikasi blok tertentu mana yang sedang disimpan. Tag biasanya merupakan sebagian dari alamat memori utama, seperti yang dijelaskan kemudian di bagian ini.



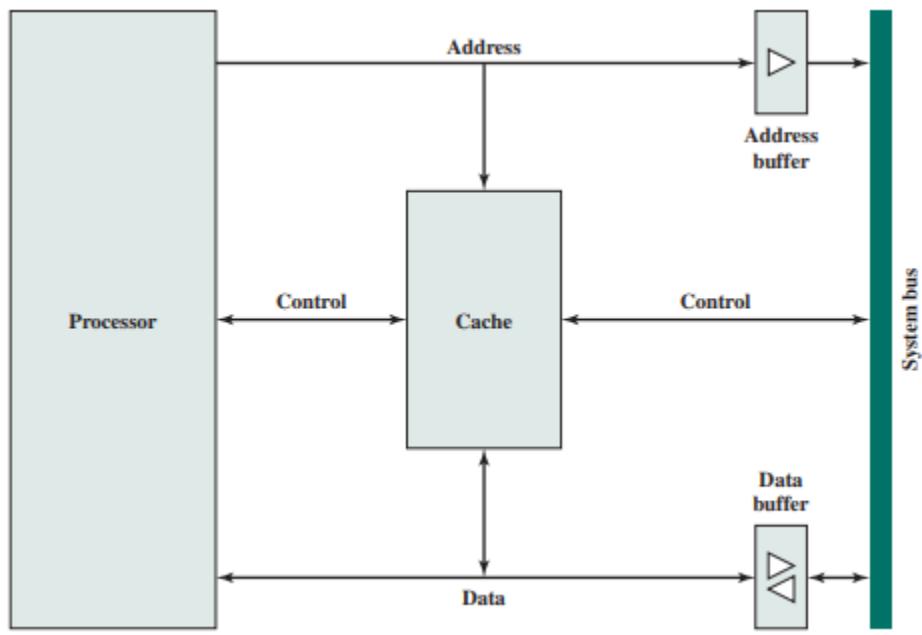
Sumber: (Stallings, 2016)

Gambar 2.3. Struktur Cache dan Main Memory

Gambar 2.4 menggambarkan operasi baca pada *cache memory*. Prosesor menghasilkan *Read Addressing* (RA) dari sebuah *word* untuk dibaca. Jika *word* itu terkandung dalam *cache*, maka dikirim ke prosesor. Jika tidak ada, maka blok yang berisi *word* itu dimuat ke dalam *cache*, dan *word* itu dikirim ke prosesor. Gambar 2.4 menunjukkan dua operasi terakhir yang terjadi secara paralel dan mencerminkan organisasi yang ditunjukkan pada Gambar 2.5, yang merupakan tipikal organisasi *cache* kontemporer. Dalam organisasi ini, *cache* terhubung ke prosesor melalui jalur data, kontrol, dan alamat. Jalur data dan alamat juga terhubung ke *buffer* data dan alamat, yang melampirkan *valid address* memori utama ke bus sistem. Ketika *hit cache* terjadi, *buffer* data dan alamat dinonaktifkan dan komunikasi hanya terjadi antara prosesor dan *cache*, tanpa lalu lintas bus sistem. Ketika terjadi *cache miss*, alamat yang diinginkan dimuat ke bus sistem dan data dikembalikan melalui *buffer* data ke *cache* dan prosesor. Di organisasi lain, *cache* secara fisik ditempatkan di antara prosesor dan memori utama untuk semua data, alamat, dan jalur kontrol. Dalam kasus terakhir ini, *word* yang diinginkan pertama kali dibaca ke dalam *cache* dan kemudian ditransfer dari *cache* ke prosesor.



Gambar 2.4. Siklus Operasi Baca pada Cache



Sumber: (Stallings, 2016)

Gambar 2.5. Organisasi Cache yang Umum

Ada beberapa elemen dasar yang berfungsi untuk mengklasifikasikan dan membedakan arsitektur *cache*. Tabel 2.3 mencantumkan elemen-elemen dasar tersebut.

Tabel 2.3. Klasifikasi Karakteristik Cache

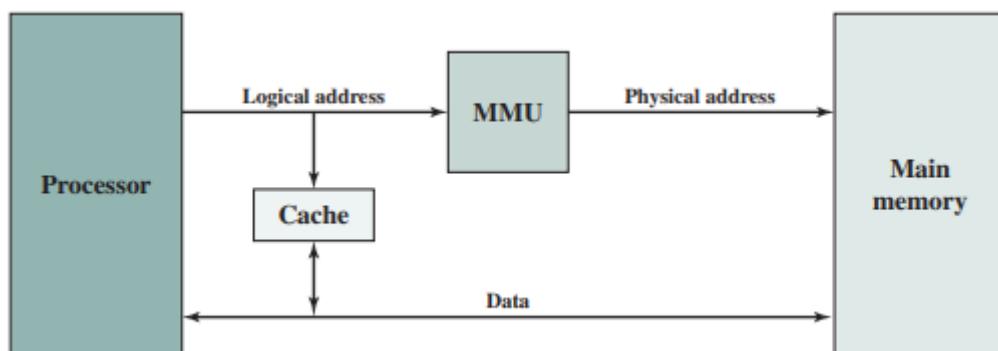
Cache address	Write policy
Logical	Write through
Physical	Write back
Cache size	Line size
Mapping function	Number of cache
Direct	Single or two level
Assosiatif	Unified or split
Set assosiatif	
Replacement algorithm	
Least Recently Used (LRU)	
First in First Out (FIFO)	
Least Frequency Used (LFU)	
Random	

Sumber: (Stallings, 2016)

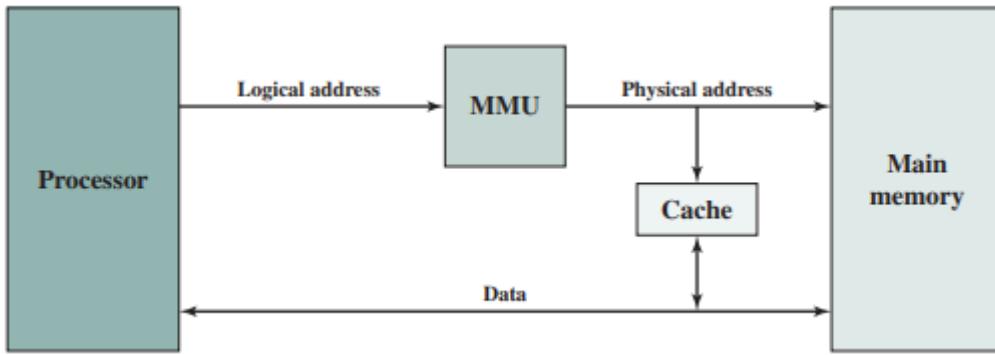
A. Cache Address

Hampir semua prosesor mendukung memori virtual, yaitu fasilitas yang memungkinkan program menangani memori dari sudut pandang logika, tanpa memperhatikan jumlah memori utama tersedia secara fisik. Ketika memori virtual digunakan, bidang alamat pada instruksi mesin berisi alamat virtual. Untuk membaca dan menulis ke dan dari memori utama, perangkat keras *Memory Management Unit* (MMU) menerjemahkan setiap alamat virtual ke alamat fisik di memori utama. Ketika alamat virtual digunakan, perancang sistem dapat memilih untuk menempatkan *cache* antara prosesor dan MMU atau antara MMU dan memori utama (Gambar 2.6). *Cache* logika, juga dikenal sebagai *cache virtual*, menyimpan data menggunakan alamat virtual. Prosesor mengakses *cache* secara langsung, tanpa melalui MMU. *Cache* fisik menyimpan data menggunakan alamat fisik memori utama.

Satu keuntungan dari *cache* logika adalah kecepatan akses *cache* lebih cepat dari pada *cache* fisik, karena *cache* logika dapat merespons sebelum MMU melakukan terjemahan alamat. Kelemahannya adalah berkaitan dengan fakta bahwa sebagian besar sistem memori virtual memasok setiap aplikasi dengan bidang alamat memori virtual yang sama. Artinya, setiap aplikasi melihat memori virtual yang dimulai pada alamat 0. Dengan demikian, alamat virtual yang sama dalam dua aplikasi berbeda mengacu pada dua alamat fisik yang berbeda.



(a) Logical Cache



(b) Physical Cache

Sumber: (Stallings, 2016)

Gambar 2.6. Logika dan Fisik Cache

B. Cache Size

Elemen kedua dalam klasifikasi memori pada Tabel 2.4 adalah ukuran *cache*. Yang diingin dari sebuah *cache* adalah kecepatan yang melebih *main memory*. Kebutuhan kapasitas *cache* yang besar, mengakibatkan semakin besar jumlah gerbang yang terlibat dalam mengatasi *cache*. Hasilnya adalah bahwa *cache* berkapasitas besar cenderung sedikit lebih lambat daripada yang kecil, bahkan ketika dibangun dengan teknologi sirkuit terpadu yang sama dan diletakkan di tempat yang sama pada *chip* dan papan sirkuit. Area *chip* dan papan yang tersedia juga membatasi ukuran *cache*. Karena kinerja *cache* sangat sensitif terhadap sifat beban kerja, tidak mungkin untuk mencapai ukuran *cache* "optimal" tunggal. Tabel 2.4 merangkum *cache* beserta ukurannya untuk beberapa jenis prosesor.

Tabel 2.4. Ukuran *Cache* untuk Beberapa Sistem Komputer

Processor	Type	Year of Introduction	L1 Cache	L2 Cache	L3 Cache
IBM 360/85	Mainframe	1968	16–32 kB	—	—
PDP-11/70	Minicomputer	1975	1 kB	—	—
VAX 11/780	Minicomputer	1978	16 kB	—	—
IBM 3033	Mainframe	1978	64 kB	—	—
IBM 3090	Mainframe	1985	128–256 kB	—	—
Intel 80486	PC	1989	8 kB	—	—
Pentium	PC	1993	8 kB/8 kB	256–512 kB	—
PowerPC 601	PC	1993	32 kB	—	—
PowerPC 620	PC	1996	32 kB/32 kB	—	—
PowerPC G4	PC/server	1999	32 kB/32 kB	256 kB to 1 MB	2 MB
IBM S/390 G6	Mainframe	1999	256 kB	8 MB	—
Pentium 4	PC/server	2000	8 kB/8 kB	256 kB	—
IBM SP	High-end server/ supercomputer	2000	64 kB/32 kB	8 MB	—
CRAY MTab	Supercomputer	2000	8 kB	2 MB	—
Itanium	PC/server	2001	16 kB/16 kB	96 kB	4 MB
Itanium 2	PC/server	2002	32 kB	256 kB	6 MB
IBM POWER5	High-end server	2003	64 kB	1.9 MB	36 MB
CRAY XD-1	Supercomputer	2004	64 kB/64 kB	1 MB	—
IBM POWER6	PC/server	2007	64 kB/64 kB	4 MB	32 MB
IBM z10	Mainframe	2008	64 kB/128 kB	3 MB	24–48 MB
Intel Core i7 EE 990	Workstation/ server	2011	6 * 32 kB/ 32 kB	1.5 MB	12 MB

IBM zEnterprise 196	Mainframe/ server	2011	24 * 64 kB/ 128 kB	24 * 1.5 MB	24 MB L3 192 MB L4
------------------------	-------------------	------	--------------------	-------------	--------------------

Catatan:

- a. Dua nilai yang dipisahkan oleh slash merujuk ke instruksi dan *cache* data.
- b. Kedua *cache* hanya instruksi; tidak ada *cache* data.

Sumber: (Stallings, 2016)

C. Fungsi Pemetaan

Karena ada lebih sedikit baris *cache* dibandingkan dengan blok memori utama, maka diperlukan algoritma untuk memetakan blok memori utama kedalam baris *cache*. Sebaliknya, diperlukan juga cara untuk menentukan blok memori utama mana yang sedang memakai baris *cache*. Pilihan fungsi pemetaan menentukan bagaimana *cache* diatur. Tiga teknik pemetaan yang dapat digunakan adalah: *direct*, *asosiatif*, dan *set-asosiatif*.

Contoh kasus: Sebuah cache dapat menampung 64 Kbyte. Data ditransfer dari memori utama ke cache dalam blok 4 bytes. Memori utama terdiri dari 16 MB, dengan setiap byte dapat dialamatkan secara langsung dengan alamat 24-bit ($2^{24} = 16M$). Jadi, untuk tujuan pemetaan, dapat diasumsikan bahwa memori utama terdiri dari 4M blok masing-masing 4 byte.

1. Direct Mapping

Teknik paling sederhana, yang dikenal sebagai pemetaan langsung, memetakan setiap blok memori utama menjadi hanya satu garis *cache* yang mungkin. Pemetaan dinyatakan sebagai

$$i = j \bmod m$$

dimana

i = nomor baris *cache*

j = nomor blok memori utama

m = jumlah baris dalam *cache*

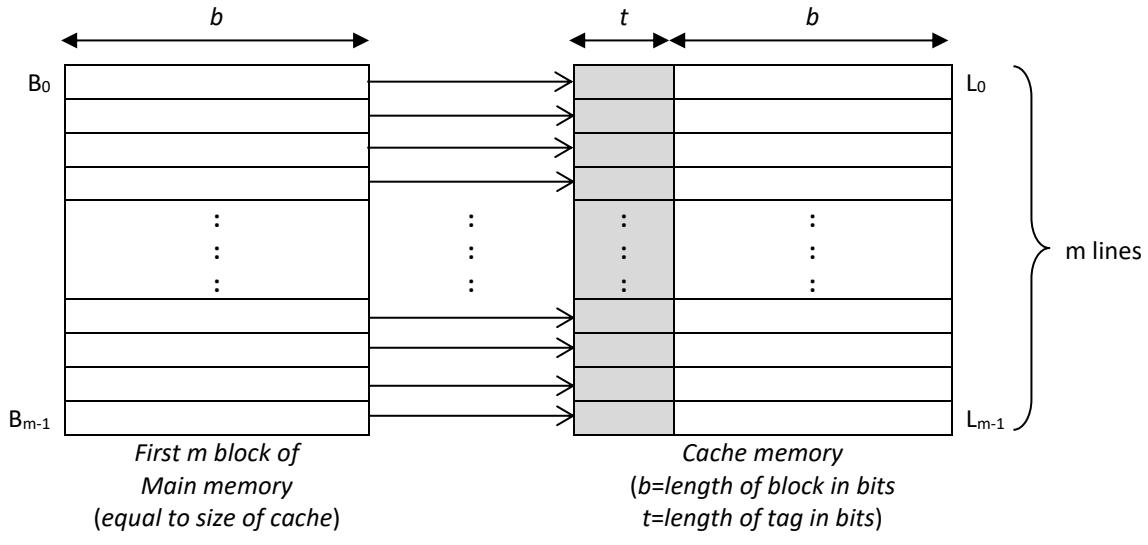
Gambar 2.7a menunjukkan pemetaan untuk m blok pertama dari memori utama. Setiap blok memori utama dipeetakan menjadi satu baris *cache* yang unik. Blok ke-m berikutnya memori utama dipertakan ke dalam *cache* dengan cara yang sama; yaitu, blok B_m dari memori utama dipetakan ke jalur L₀ *cache*, blok B_{m+1} peta ke jalur L₁, dan seterusnya.

Misalnya jika *main memory* memiliki 64 blok (j=0, 1, ..., 63). *Cache* memiliki baris sebanyak m=4 (i=0, 1, 2, 3). Maka setiap baris pada *cache* akan ditempati blok pada *main memory* sebagai berikut

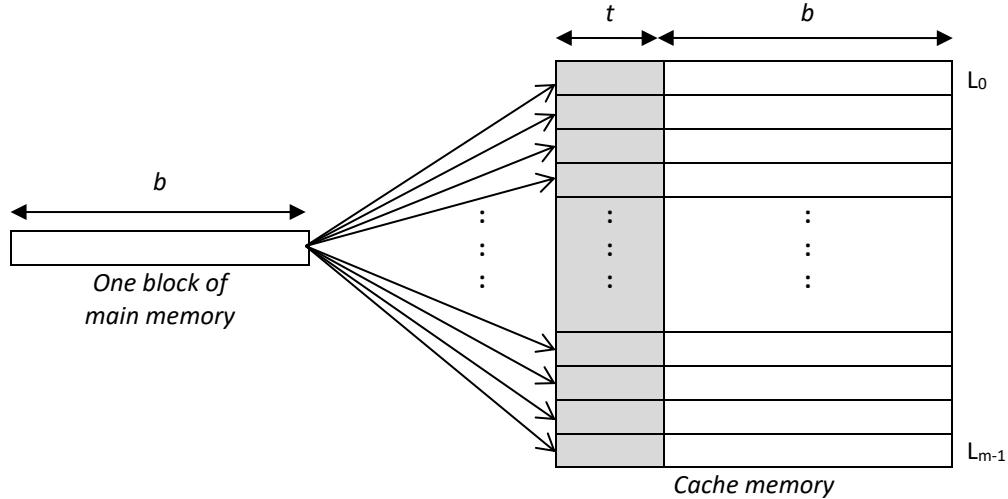
- *Cache* baris 0 dapat ditempati blok *main memory*: 0, 4, 8, 12,..., 60.
- *Cache* baris 1 dapat ditempati blok *main memory*: 1, 5, 9, 13,..., 61.
- *Cache* baris 2 dapat ditempati blok *main memory*: 2, 6, 10, 14,..., 62.
- *Cache* baris 3 dapat ditempati blok *main memory*: 3, 7, 11, 15,..., 63.

Fungsi pemetaan mudah diimplementasikan menggunakan alamat memori utama. Dimana Pada cara ini, address pada *main memory* dibagi 3 bidang atau bagian, yaitu:

1. *Tag identifier*, disimpan pada baris *cache* bersama dengan blok yang dipetakan dari *main memory*
2. *Line number identifier*, berisi informasi nomor fisik (bukan logika) baris (*line*) pada *chace*
3. *Word identifier (offset)*, berisi informasi mengenai lokasi *word* atau unit *addressable* lainnya dalam line tertentu pada *cache*.



(a) Direct mapping



(b) Associative mapping

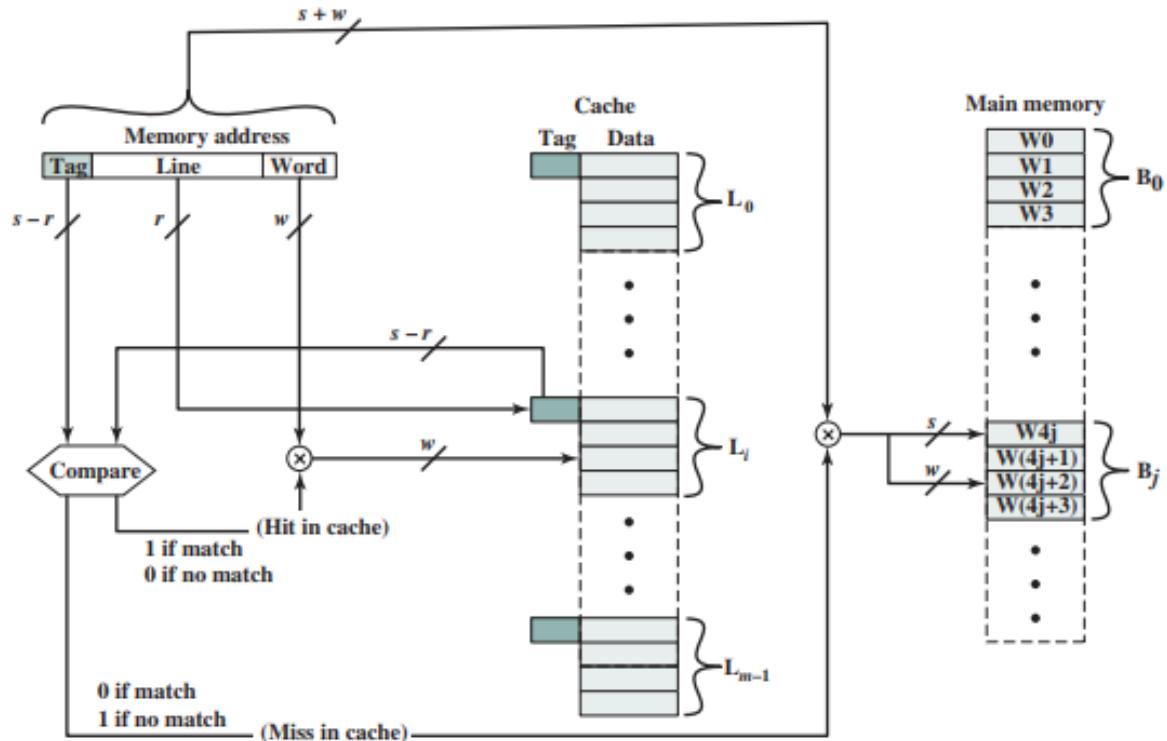
Sumber: (Stallings, 2016)

Gambar 2.7. Mapping dari Main Memory ke Cache: Direct dan Associative

Gambar 2.7(a) menggambarkan mekanisme umum dari *direct mapping cache*. Untuk keperluan akses *cache*, setiap alamat memori utama dapat dilihat terdiri dari tiga bidang. Bit-bit paling signifikan mengidentifikasi sebuah *word* atau *byte* unik dalam blok memori utama; pada kebanyakan mesin kontemporer, alamatnya berada pada level *byte*. Bit yang tersisa menentukan salah satu blok memori utama 2^s . Logika *cache* mengartikan bit-bit ini sebagai *tag* bit-bit s-r (bagian paling signifikan) dan bidang baris bit-bit r. Bidang yang terakhir ini mengidentifikasi salah satu dari baris $m = 2^r$ *cache*. Berikut adalah rangkumannya:

- Panjang Alamat = $(s + w)$ bit
- Jumlah unit yang dapat dialamatkan = 2^{s+w} *word* atau *byte*
- Ukuran blok = ukuran garis = 2^w *word* atau *byte*
- Jumlah blok dalam memori utama = $(2^{s+w}/2^w) = 2^s$
- Jumlah baris dalam *cache* = $m = 2^r$
- Ukuran *cache* = 2^{r+w} *word* atau *byte*

- Ukuran tag = $(s - r)$ bit



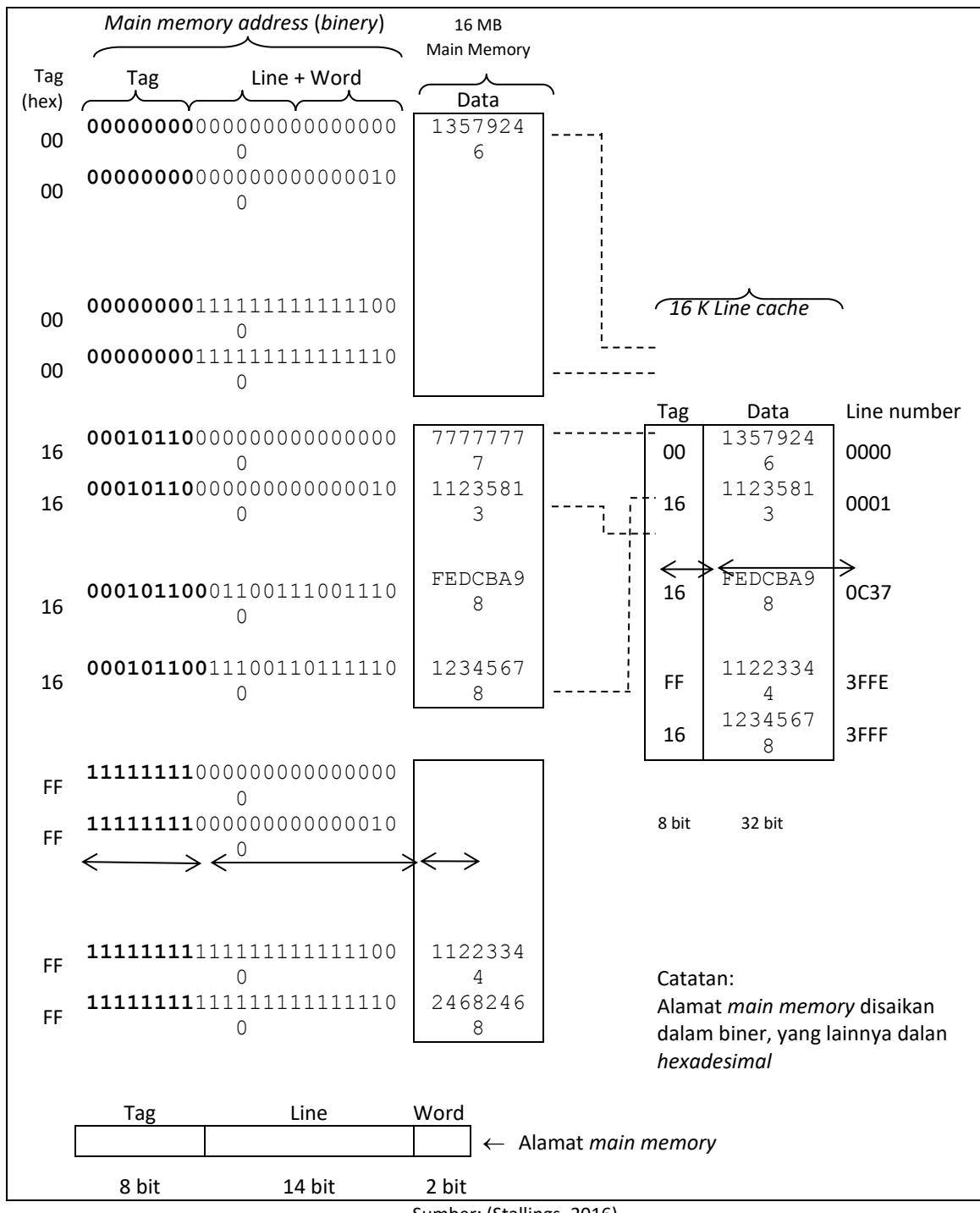
Gambar 2.8. Organisasi Direct Mapping Cache

Gambar 2.8 menunjukkan contoh menggunakan pemetaan langsung. Misalnya: $m = 16K = 2^{14}$, dan $i = j$ modulo 2^{14} . Pemetaan menjadi

<i>Cache line</i>	Memulai Alamat pada Block Memori
0	000000, 010000, ..., FF0000
1	000004, 010004, ..., FF0004
:	:
:	:
$2^{14}-1$	00FFFC, 01FFFC, ..., FFFF0C

Tidak ada dua blok yang memetakan ke nomor baris yang sama memiliki nomor tag yang sama. Jadi, blok dengan alamat mulai 000000, 010000, ..., FF0000 masing-masing memiliki nomor tag 00, 01, ..., FF secara berurutan.

Mengacu kembali ke Gambar 2.4, operasi baca berfungsi sebagai berikut: sistem *cache* disajikan dengan alamat 24-bit. Nomor baris 14-bit digunakan sebagai indeks ke dalam *cache* untuk mengakses baris tertentu. Jika nomor tag 8-bit cocok dengan nomor tag yang saat ini disimpan di baris itu, maka nomor word 2-bit digunakan untuk memilih salah satu dari 4 byte di baris itu. Jika tidak, bidang tag+line 22-bit digunakan untuk mengambil *block* dari memori utama. Alamat aktual yang digunakan untuk pengambilan adalah 22-bit tag+line yang digabungkan dengan dua 0 bit, sehingga 4 byte diambil mulai dari batas *block*. Gambar 2.9 memberikan contoh pemetaan data dari *main memory* yang dipetakan ke dalam *cache* dengan direct mapping.



Sumber: (Stallings, 2016)

Gambar 2.9. Contoh *Direct Mapping*

Efek pemetaan ini adalah bahwa blok memori utama dipetakan ke baris *cache* sebagai berikut:

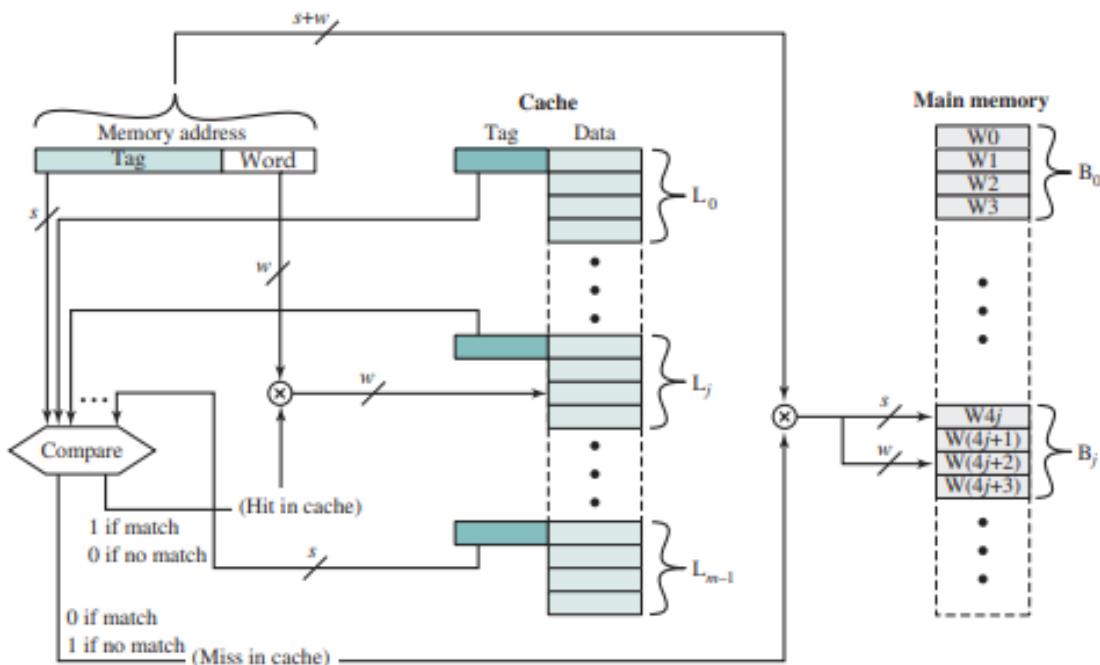
<i>Cache line</i>	<i>Main memory blocks assigned</i>
0	0, m, 2m, ..., $2^s - m$
1	1, m+1, 2m+1, ..., m+1
:	:
m-1	m-1, 2m-1, 3m-1, ..., $2^s - 1$

Dengan demikian, penggunaan sebagian dari alamat sebagai nomor baris menyediakan pemetaan unik dari setiap blok memori utama ke dalam *cache*. Ketika sebuah blok benar-benar dibaca ke dalam baris yang ditugaskan, perlu untuk menandai data untuk membedakannya dari blok lain yang dapat masuk ke dalam baris itu. Bit $s - r$ yang paling signifikan melayani tujuan ini.

Teknik pemetaan langsung sederhana dan murah untuk diterapkan. Kerugian utamanya adalah bahwa ada lokasi *cache* tetap untuk setiap blok yang diberikan. Dengan demikian, jika suatu program terjadi untuk merujuk *words* berulang kali dari dua blok berbeda yang memetakan ke dalam baris yang sama, maka blok-blok itu akan terus ditukar dalam *cache*, dan *hit ratio* akan rendah (sebuah fenomena yang dikenal sebagai *thrashing*).

2. Assosiatif Mapping

Gambar 2.10 menggambarkan logika organiasi *assosiatif mapping*. Pemetaan asosiatif mengatasi kelemahan *direct mapping* dengan mengizinkan setiap blok memori utama untuk dimuat ke setiap baris *cache* (Gambar 2.10b). Dalam hal ini, logika kontrol *cache* mengartikan alamat memori hanya sebagai bidang *Tag* dan *Word*. Bidang *Tag* secara unik mengidentifikasi blok memori utama. Untuk menentukan apakah suatu blok ada dalam *cache*, logika kontrol *cache* harus secara bersamaan memeriksa setiap tag baris untuk suatu kecocokan.



Sumber: (Stallings, 2016)

Gambar 2.10. Organisasi *Associative Mapping*

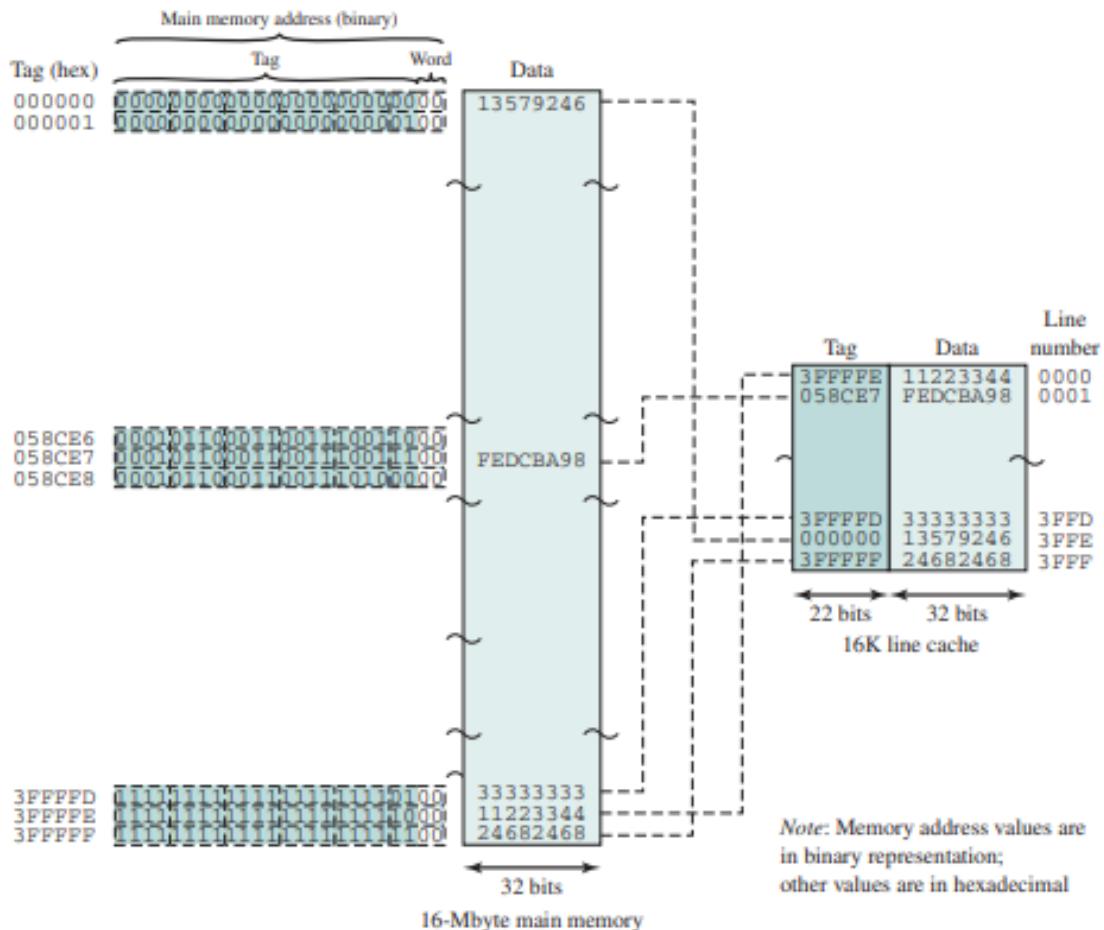
Gambar 2.11 menunjukkan contoh menggunakan pemetaan asosiatif. Alamat memori utama terdiri dari tag 22-bit dan nomor byte 2-bit. Tag 22-bit harus disimpan dengan blok data 32-bit untuk setiap baris dalam *cache*. Perhatikan bahwa itu adalah 22 bit paling kiri (paling signifikan) dari alamat yang membentuk tag. Dengan demikian, alamat heksadesimal 24-bit 16339C memiliki tag 22-bit 058CE7. Ini mudah dilihat dalam notasi biner:

<i>memory address</i>	0001	0110	0011	0011	1001	1100	(binary)
	1	6	3	3	9	C	(hex)
<i>tag (leftmost 22 bits)</i>	00	0101	1000	1100	1110	0111	(binary)
	0	5	8	C	E	7	(hex)

Perhatikan bahwa tidak ada bidang dalam alamat yang sesuai dengan nomor baris, sehingga jumlah baris dalam *cache* tidak ditentukan oleh format alamat. Untuk merangkum:

- Panjang Alamat = $(s + w)$ bit
- Jumlah unit yang dapat dialamatkan = 2^{s+w} word atau byte
- Ukuran blok = ukuran garis = 2^w word atau byte
- Jumlah blok dalam memori utama = $(2^{s+w})/2^w = 2^s$
- Jumlah baris dalam *cache* = tidak ditentukan
- Ukuran tag = s bit

Dengan pemetaan asosiatif, ada fleksibilitas untuk blok mana yang harus diganti ketika blok baru dibaca ke dalam *cache*. Algoritma penggantian, yang akan dibahas nanti di bagian ini, dirancang untuk memaksimalkan rasio hit. Kerugian utama pemetaan asosiatif adalah sirkuit kompleks yang diperlukan untuk memeriksa tag semua garis *cache* secara paralel.



Sumber: (Stallings, 2016)

Gambar 2.11. Contoh Associative Mapping

3. Set-Associative Mapping

Set-associative mapping adalah perpaduan yang menunjukkan kekuatan dari pendekatan direct mapping dan asosiatif mapping, sekaligus mengurangi kelemahan mereka. Dalam hal ini, *cache* terdiri dari sejumlah set, yang masing-masing terdiri dari sejumlah baris. Hubungannya adalah

$$m = n * k$$

$$i = j \text{ modulo } n$$

dimana

i = jumlah set *cache*

v = jumlah set

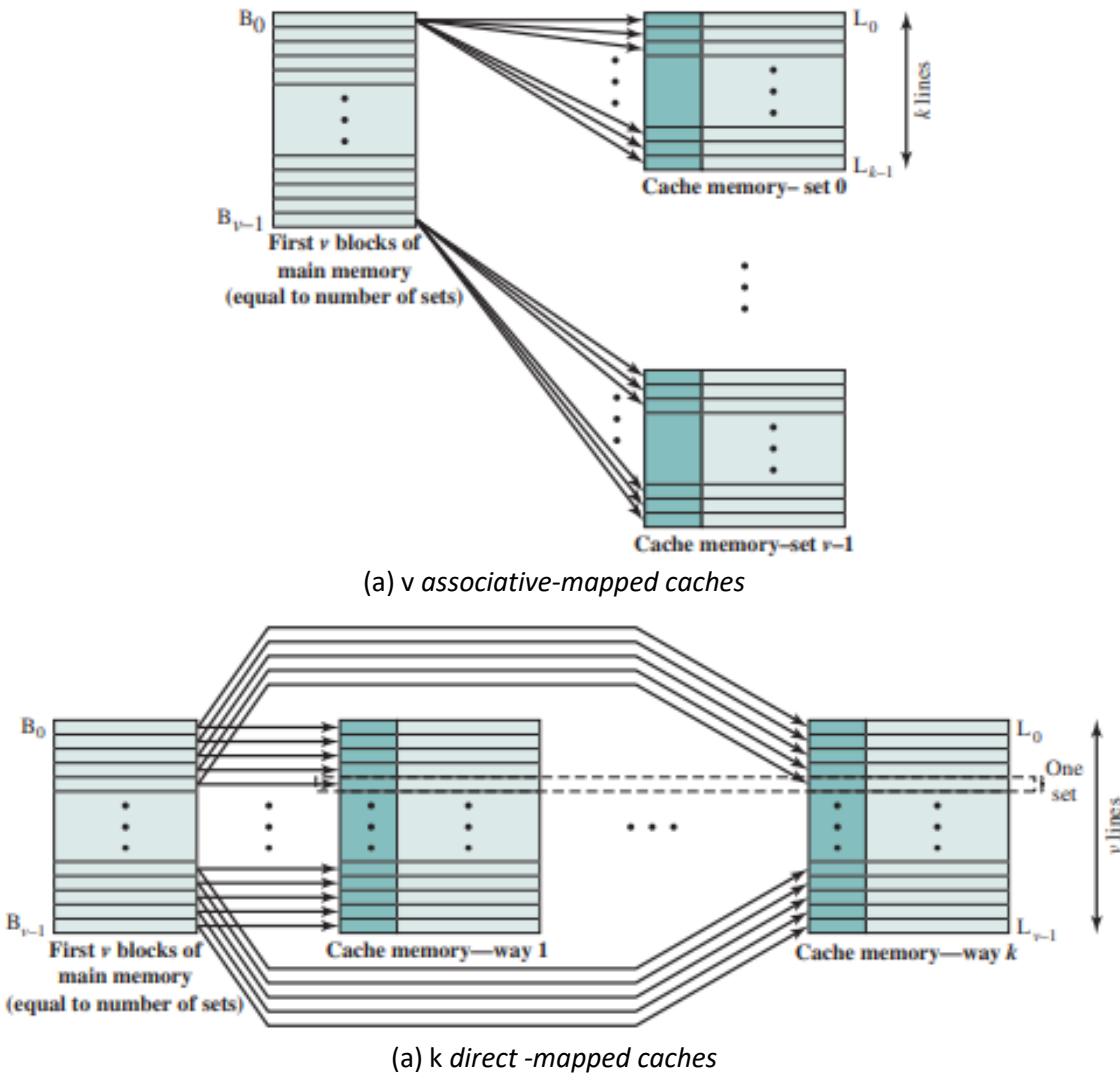
j = nomor blok memori utama

k = jumlah baris dalam setiap set

m = jumlah baris dalam *cache*

Ini disebut sebagai *k-way set-associative mapping*. Dengan *set-associative mapping*, *block B* dapat dipetakan ke salah satu garis set *j*. Gambar 2.12a mengilustrasikan pemetaan ini untuk *n* blok pertama memori utama. Seperti halnya pemetaan asosiatif, setiap *word* memetakan ke dalam beberapa baris *cache*. Untuk pemetaan *set-associatif*, setiap *word* memetakan ke semua baris

cache di set tertentu, sehingga memori utama memblokir B_0 peta ke set 0, dan seterusnya. Dengan demikian, *cache set-asosiatif* dapat diimplementasikan secara fisik sebagai *cache asosiatif*. Juga dimungkinkan untuk mengimplementasikan *cache set-asosiatif* sebagai *cache pemetaan langsung*, seperti yang ditunjukkan pada Gambar 2.12b.

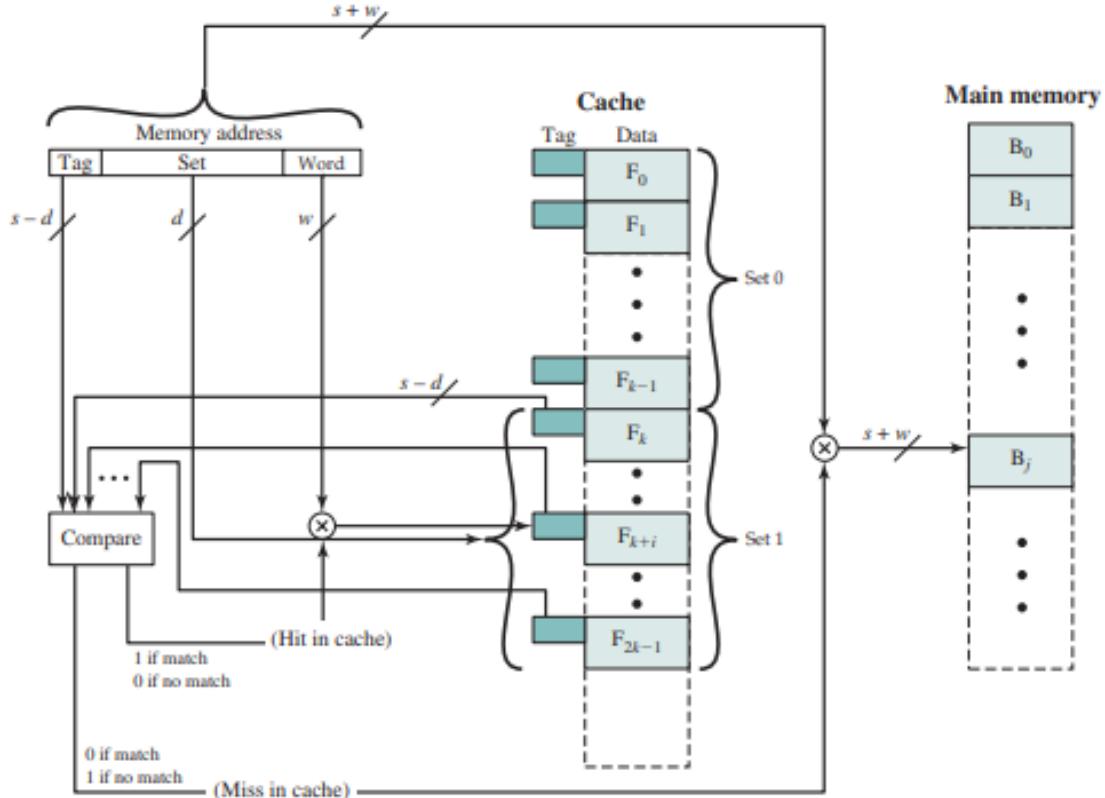


Sumber: (Stallings, 2016)

Gambar 2.12. *Mapping* dari *main memory* ke *cache*: *k-way Set Assosiative*

Setiap *cache* yang dipetakan langsung disebut sebagai cara (*way*), yang terdiri dari n baris. N baris pertama dari memori utama langsung dipetakan ke dalam n baris dari setiap *way*; kelompok berikutnya dari n baris memori utama dipetakan dengan cara yang sama, dan seterusnya. Implementasi yang dipetakan langsung biasanya digunakan untuk derajat asosiasi yang kecil (nilai-nilai kecil k) sedangkan implementasi yang dipetakan secara asosiatif biasanya digunakan untuk tingkat asosiasi yang lebih tinggi.

Untuk pemetaan *set-asosiatif*, logika kontrol *cache* mengartikan alamat memori sebagai tiga bidang: *Tag*, *Set*, dan *Word*. Bit set d menentukan satu dari *Set* $n = 2^d$. Bit-bit dari bidang *Tag* dan *Set* menentukan salah satu dari 2^s blok memori utama. Gambar 2.13 menggambarkan logika kontrol *cache*. Dengan pemetaan asosiatif penuh, *tag* dalam alamat memori cukup besar dan harus dibandingkan dengan *tag* setiap baris dalam *cache*.



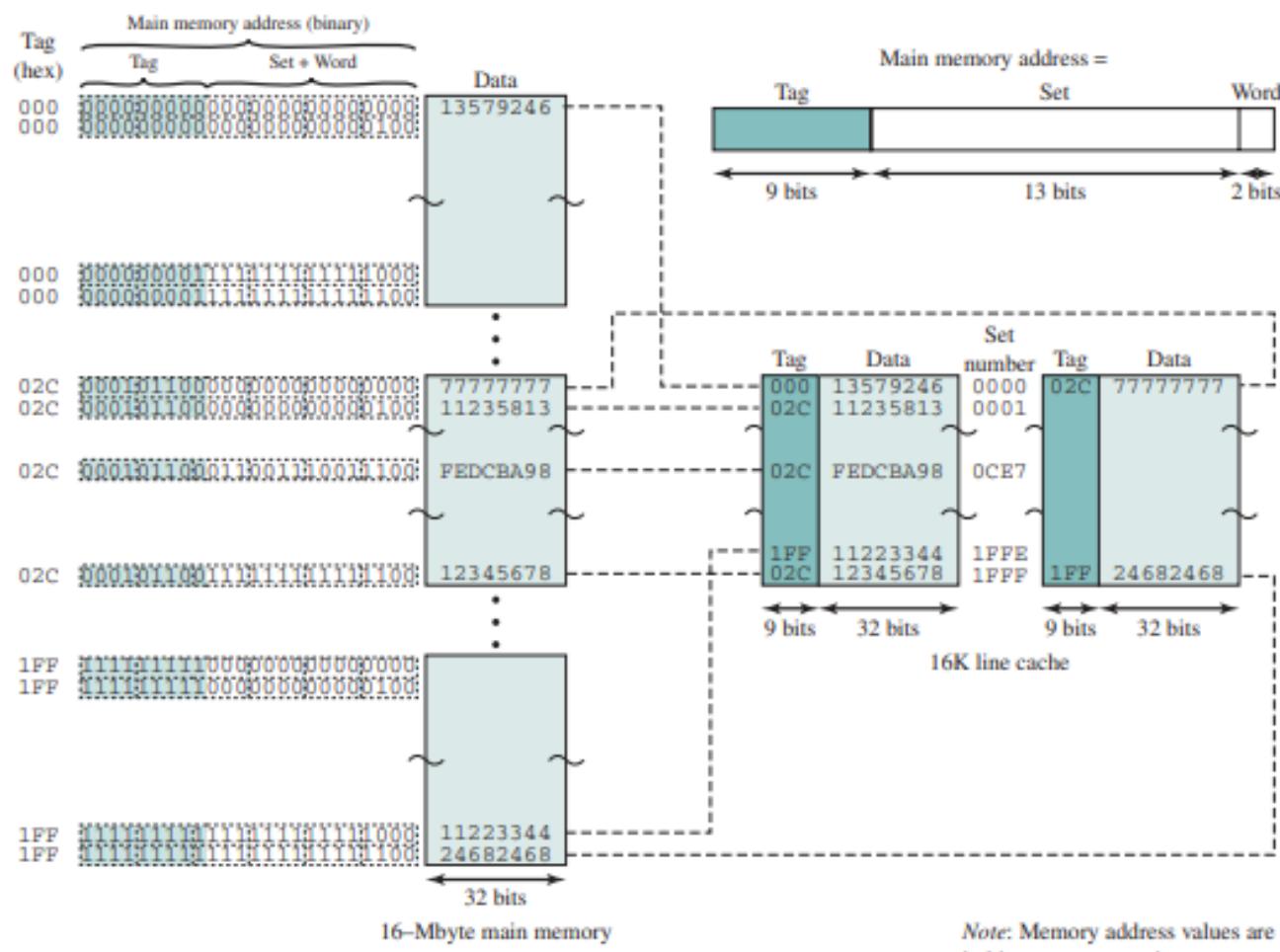
Sumber: (Stallings, 2016)

Gambar 2.13. Organisasi Set Associative K-way

Dengan pemetaan set-asosiatif k-way, tag dalam alamat memori jauh lebih kecil dan hanya dibandingkan dengan tag k dalam satu set. Sebagai rangkuman:

- Panjang Alamat = $(s + w)$ bit
- Jumlah unit yang dapat dialamatkan = 2^{s+w} word atau byte
- Ukuran blok = ukuran garis = 2^w word atau byte
- Jumlah blok dalam memori utama = $(2^{s+w})/2^w = 2^s$
- Jumlah baris dalam set = k
- Jumlah set = $n = 2^d$
- Jumlah baris dalam cache = $m = kn = k * 2^d$
- Ukuran cache = $k * 2^{d+w}$ word atau byte
- Ukuran tag = $(s - d)$ bit

Gambar 2.14 menunjukkan contoh penggunaan pemetaan set-asosiatif dengan dua garis di setiap set, disebut sebagai two way set-asosiatif. Nomor set 13-bit mengidentifikasi satu set unik dari dua baris dalam cache. Ini juga memberikan jumlah blok di memori utama, modulo 213. Ini menentukan pemetaan blok menjadi garis. Dengan demikian, blok 000000, 008000, ..., FF8000 dari peta memori utama ke dalam set cache 0. Setiap blok itu dapat dimuat ke salah satu dari dua baris di set. Tidak ada dua blok yang memetakan ke set cache yang sama memiliki nomor tag yang sama. Untuk operasi baca, angka set 13-bit digunakan untuk menentukan set dua garis yang harus diperiksa. Kedua baris dalam set diperiksa untuk mencocokkan dengan nomor tag dari alamat yang akan diakses.



Sumber: (Stallings, 2016)

Gambar 2.14. Contoh Two-Way Set Associative Mapping

D. Replacement Algorithm

Untuk melakukan pemindahan informasi yang dipindahkan saat mendesain *cache*, perlu juga membangun perangkat keras kontrol untuk memilih entri mana yang akan dipindahkan ketika *cache* penuh dan terjadi miss, artinya perlu memasukkan baris yang berisi informasi yang diinginkan. Dalam *cache* yang dipetakan langsung, hanya ada satu tempat yang dapat dituju item tertentu, jadi ini sederhana, tetapi dalam cache set-asosiatif atau asosiatif penuh, ada beberapa tempat potensial untuk memuat baris tertentu, jadi harus memiliki beberapa cara untuk memutuskan di antara mereka. Kriteria utama untuk ini, seperti dalam aspek lain dari desain *cache*, adalah algoritma harus sederhana sehingga dapat diterapkan di perangkat keras dengan sedikit penundaan. Untuk memaksimalkan rasio *hit*, juga diinginkan agar algoritma menjadi efektif; artinya, untuk memilih pengganti saluran yang tidak akan digunakan lagi untuk waktu yang lama. Salah satu strategi penggantian yang mungkin adalah algoritma *Least Frequently Used* (LFU). Algoritma ini memilih untuk mengganti baris yang sejauh ini telah memberikan hasil *hit* paling sedikit (menerima klik paling sedikit), dengan alasan bahwa baris tersebut kemungkinan akan tetap menjadi baris yang paling jarang digunakan jika dibiarkan tetap di *cache*. Baris yang sering terkena *hit* "diberi hadiah" dengan diizinkan untuk tetap berada di *cache*, dengan harapan akan terus berguna di masa mendatang. Salah satu potensi masalah dengan pendekatan ini adalah bahwa garis yang telah dimuat baru-baru ini mungkin belum memiliki jumlah penggunaan yang tinggi sehingga mungkin tergeser meskipun memiliki potensi untuk digunakan lebih banyak di masa mendatang. Masalah utama, bagaimanapun, adalah dengan kompleksitas perangkat keras yang dibutuhkan. LFU membutuhkan penghitung yang akan dibangun untuk setiap entri (baris) di *cache* untuk mengikuti berapa kali itu telah diakses; nilai hitungan ini harus dibandingkan (dan yang dipilih disetel ulang ke nol) setiap kali diperlukan untuk mengganti baris. Karena kerumitan perangkat keras ini, LFU sangat tidak praktis sebagai algoritma penggantian *cache*.

Algoritma pengganti lain yang dapat mencapai hasil yang mirip dengan LFU dengan kompleksitas perangkat keras yang lebih sedikit mencakup *Least Recently Used* (LRU) dan *First In First Out* (FIFO). Algoritma LRU menggantikan garis yang terkena *hit* paling lama, terlepas dari berapa kali telah digunakan. FIFO menggantikan baris "terlama", yaitu baris yang paling lama disimpan di *cache*. Masing-masing pendekatan ini, dengan caranya sendiri, berupaya untuk menggantikan entri yang memiliki lokalitas temporal paling sedikit yang terkait dengannya dengan harapan bahwa entri itu tidak akan dibutuhkan lagi segera.

E. Write Policy

Operasi *cache* yang menggunakan salah satu dari tiga pemetaan yang telah didiskusikan sebelumnya, cukup mudah selama akses memori terbatas hanya untuk membaca. Hal ini dimungkinkan dalam kasus *cache* khusus instruksi, karena kode yang memodifikasi sendiri tidak lazim terjadi. Namun, saat mendesain *cache* data dalam sistem dengan arsitektur *cache* terpisah (*split cache architecture* atau *modified Harvard*), atau *unified cache* architecture (Princeton atau instruksi/data campuran), penulisan ke lokasi *cache* harus selalu diizinkan. Jika konten baris tersebut baru saja dibaca, maka dapat dengan mudah ditimpas oleh data baru. Tetapi jika ada lokasi di baris itu yang telah dimodifikasi, harus dimastikan memori utama mencerminkan konten yang diperbarui sebelum memuat data baru di tempatnya.

Write-through cache adalah pendekatan paling sederhana untuk menjaga konten memori utama tetap konsisten dengan *cache*. Setiap kali dilakukan penulis ke suatu lokasi dalam *cache*, juga dilakukan penulisan ke lokasi memori utama yang sesuai. Kedua penulisan ini biasanya dapat dimulai pada waktu yang sama, tetapi penulisan memori utama membutuhkan waktu lebih lama dan dengan demikian menentukan waktu keseluruhan untuk operasi tulis.

Satu keuntungan dari metode ini adalah relatif mudah untuk dibangun ke dalam perangkat keras. Selain itu, karena adanya write-through, memori utama selalu memiliki data terbaru, identik dengan konten *cache*. Karena pembacaan berikutnya dari lokasi yang sama akan masuk ke *cache* dan tetap mendapatkan data terbaru. Namun, jika ada perangkat lain (misalnya, DMA atau prosesor I/O) dalam sistem, selalu memperbarui memori utama dengan data terbaru dapat sedikit menyederhanakan banyak hal.

write-through cache lebih kompleks untuk diterapkan, tetapi dapat meningkatkan kinerja jika penulisan sering dilakukan. Pada penulisan yang dilakukan pada *cache*, hanya lokasi *cache* yang diperbarui. Memori utama hanya ditulis jika baris yang telah dimodifikasi dipindahkan dari *cache* untuk memberi ruang bagi baris baru yang akan dimuat. Hal ini membutuhkan penambahan bit ke setiap tag untuk menunjukkan apakah baris terkait telah ditulis atau tidak. Bit ini sering disebut "bit tidak konsisten" (*inconsistent bit*), atau disebut juga dengan "*dirty bit*". Lokasi *cache* yang telah ditulis dapat disebut "*dirty cell*" atau "*dirty word*". Jika bit ini 0, maka baris ini hanya dibaca, dan perangkat keras kontrol *cache* akan mengetahui bahwa baris baru dapat dimuat di atas informasi yang ada. Jika dirty bit adalah 1, maka baris tersebut (atau setidaknya lokasi yang dimodifikasi di dalamnya) harus disalin atau "ditulis kembali (*write-back*)" ke memori utama sebelum ditimpas oleh informasi baru. Jika ini tidak dilakukan, maka penulisan tersebut seolah-olah tidak pernah terjadi; memori utama tidak akan pernah diperbarui, dan sistem akan beroperasi secara tidak benar.

Keuntungan dan kerugian dari *write-back* strategy adalah kebalikan dari *write-through cache*. Menggunakan pendekatan *write-back* umumnya akan memaksimalkan kinerja karena *write hits* hampir sama bermanfaatnya dengan *read hits*. Jika diperlukan menulis ke lokasi di-*cache* sebanyak 10 kali, misalnya, hanya satu penulisan ke memori utama yang dibutuhkan, 9 dari 10 *write hit* tidak memerlukan akses memori utama dan karenanya tidak membutuhkan lebih banyak waktu daripada *read hits*. Dengan pendekatan ini, data di memori utama bisa menjadi "*basically stale*", yaitu tidak ada jaminan bahwa apa yang ada di memori utama cocok dengan aktivitas terbaru CPU. Ini adalah masalah potensial yang harus dideteksi dan ditangani jika perangkat lain dalam sistem mengakses memori.

Selain itu, logika yang diperlukan untuk melakukan *write-backs* lebih kompleks daripada yang diperlukan untuk melakukan *write-throughs*. Dalam hal ini tidak hanya membutuhkan sedikit tambahan untuk setiap tag untuk melacak baris yang diperbarui; namun membutuhkan logika untuk memeriksa bit-bit ini dan memulai operasi line *write-back* jika diperlukan. Untuk melakukan *write-backs*, pengontrol harus menahan operasi baca untuk mengisi baris baru hingga *write-backs* selesai, atau harus menyangga informasi yang dipindahkan di lokasi sementara dan menuliskannya kembali setelah pengisian baris selesai.

2.4. Read Only Memory (ROM)

Read Only Memory (ROM) adalah memori yang berisi pola data permanen (residen) yang tidak dapat diubah. ROM tidak mudah menguap (*nonvolatile*), artinya tidak ada sumber daya yang diperlukan untuk mempertahankan nilai bit dalam memori. Meskipun ROM dapat dicara, namun tidak diijinkan untuk menulis data baru ke dalamnya. Aplikasi ROM yang penting adalah pemrograman mikro serta aplikasi potensial lainnya seperti: *library subroutines* untuk fungsi yang sering digunakan, Program sistem, serta tabel fungsi.

Keunggulan ROM adalah bahwa data atau program secara permanen berada dalam memori utama dan tidak perlu dimuat dari perangkat penyimpanan sekunder. ROM dibuat seperti *chip Intergrated Circuit (IC)* lainnya, dimana data ditransfer ke dalam *chip* sebagai bagian dari proses fabrikasi. Ini menyajikan dua masalah: biaya penyimpanan data yang relatif besar, baik satu atau ribuan salinan

ROM tertentu yang dibuat, dan tidak boleh ada kesalahan data (jika satu bit salah, seluruh *batch* ROM harus dibuang).

A. Programmable-ROM (PROM)

Ketika hanya sejumlah kecil ROM dengan konten memori tertentu diperlukan, alternatif yang lebih murah adalah *Programmable* ROM (PROM), yaitu ROM yang dapat diprogram atau ditulis. Seperti halnya ROM, PROM bersifat *nonvolatile* dan dapat ditulis hanya sekali. Untuk PROM, proses penulisan dilakukan secara elektrik dan dapat dilakukan oleh pabrik atau user. Peralatan khusus diperlukan untuk proses penulisan atau "pemrograman". PROM memberikan fleksibilitas dan kenyamanan.

B. EPROM

Erasable Programmable ROM (EPROM) adalah ROM yang dapat diprogram dan dihapus secara optik, dibaca dan ditulis secara elektrik, seperti halnya PROM. Namun, sebelum operasi penulisan, semua sel penyimpanan harus dihapus ke keadaan awal yang sama dengan paparan *chip* yang dikemas untuk radiasi ultraviolet. Penghapusan dilakukan dengan menyinari sinar ultraviolet yang intens melalui jendela yang dirancang ke dalam *chip* memori. Proses penghapusan ini dapat dilakukan berulang kali; setiap penghapusan bisa memakan waktu hingga 20 menit untuk. Dengan demikian, EPROM dapat diubah beberapa kali dan, seperti ROM dan PROM, menyimpan datanya secara virtual tanpa batas. Untuk jumlah penyimpanan yang sebanding, EPROM lebih mahal daripada PROM, tetapi memiliki keunggulan dari beberapa kemampuan pembaruan.

A ₇	1	24	V _{cc}
A ₆	2	23	A ₉
A ₅	3	22	A ₈
A ₄	4	21	V _{pp}
A ₃	5	20	CS
A ₂	6	19	A ₁₀
A ₁	7	18	PD/PG
A ₀	8	17	M
O ₀	9	16	O ₇
O ₁	10	15	O ₆
O ₂	11	14	O ₅
GND	12	13	O ₄

A₀-A₁₁ = Pin alamat
PD/PG=Mat Daya/Program
CS=Chip Select (aktif rendah)
O₀-O₇=Output
Sumber: (Brey, 2009)

Gambar 2.15. Pin-out EPROM 2716

Gambar 2.15 mengilustrasikan konfigurasi pin pada IC 2716 EPROM produk Intel, yang mewakili sebagian besar EPROM. Peranti ini memiliki *input* alamat sepanjang 11 bit, dan data sepanjang 8 bit. Melalui masukan alamat 11 bit, memori ini memiliki kapasitas 2K x 8 bit (2 KB). Tabel 2.5 merangkung Seri EPROM 27XXX .

Tabel 2.5. Seri EPROM 27XXX

Seri	Kapasitas
2704	512 x 8 bit
2708	1K x 8 bit
2716	2K x 8 bit
2732	4K x 8 bit
2764	8K x 8 bit
27128	16K x 8 bit
27256	32K x 8 bit
27512	64K x 8 bit
271024	128K x 8 bit

Sumber: (Brey, 2009)

C. EEPROM

Erasable Programmable ROM (EPROM) dapat dihapus secara optik, dapat dibaca dan ditulis secara elektrik, seperti halnya dengan PROM. Namun, sebelum operasi penulisan, semua sel penyimpanan harus dihapus ke keadaan awal yang sama dengan paparan *chip* yang dikemas untuk radiasi ultraviolet. Penghapusan dilakukan dengan menyinari sinar ultraviolet yang intens melalui jendela yang dirancang ke dalam *chip* memori. Proses penghapusan ini dapat dilakukan berulang kali; setiap penghapusan dapat dilakukan hingga 20 menit. Dengan demikian, EPROM dapat diubah beberapa kali dan, seperti ROM dan PROM, menyimpan datanya secara virtual tanpa batas. Untuk jumlah penyimpanan yang sebanding, EPROM lebih mahal daripada PROM, tetapi memiliki keunggulan dari beberapa kemampuan pembaruan.

Bentuk yang lebih menarik dari kebanyakan ROM adalah *Electrically Erasable PROM* (EEPROM) yang dapat dihapus secara elektrik. Ini adalah memori yang sebagian besar dapat dibaca, dapat ditulis kapan saja tanpa menghapus konten sebelumnya dimana hanya *byte* atau *byte* yang di-update yang diperbarui. Operasi tulis memakan waktu lebih lama dari operasi membaca, dengan urutan beberapa ratus mikrodetik per *byte*. EEPROM menggabungkan keunggulan nonvolatilitas dengan fleksibilitas yang dapat diperbarui di tempat, menggunakan kontrol bus biasa, alamat, dan jalur data. EEPROM lebih mahal dari EPROM dan juga kurang padat, mendukung lebih sedikit bit per *chip*.

2.5. Flash Memory

Bentuk lain dari memori semikonduktor adalah *flash memory* (dinamakan demikian karena kecepatannya memprogram kembali) atau sering disebut sebagai *Read-Mostly Memory* (RMM). Flash memory sering disebut juga dengan EEPROM (*Electrically Erasable Programmable ROM*), EAPROM (*Electrically Alterable Programmable ROM*), atau NVRAM (*Non-Volatile RAM*).

Pertama kali diperkenalkan pada pertengahan 1980-an, *flash memory* adalah peralihan antara EPROM dan EEPROM baik dalam hal biaya maupun fungsionalitas. Seperti EEPROM, memori flash menggunakan teknologi penghapus listrik. Seluruh *flash memory* dapat dihapus dalam satu atau beberapa detik, yang jauh lebih cepat daripada EPROM. Selain itu, dimungkinkan untuk menghapus hanya blok memori daripada seluruh *chip*. *Flash memory* mendapatkan namanya karena *microchip* diatur sehingga sebagian sel memori dihapus dalam satu tindakan atau "flash." Namun, memori flash tidak memberikan penghapusan tingkat *byte*. Seperti EPROM, *flash memory* hanya menggunakan satu transistor per bit, dan karenanya mencapai kepadatan tinggi (dibandingkan dengan EEPROM) dari EPROM.

2.6. Random Access Memory (RAM)

Satu karakteristik pembeda dari RAM adalah bahwa dimungkinkan untuk membaca data dari memori dan untuk menulis data baru ke dalam memori dengan mudah dan cepat. Baik membaca dan menulis dicapai melalui penggunaan sinyal listrik.

Karakteristik lain yang membedakan RAM adalah *volatile*. RAM harus dilengkapi dengan catu daya konstan. Jika daya terputus, maka data hilang. Dengan demikian, RAM hanya dapat digunakan sebagai penyimpanan sementara. Dua bentuk tradisional RAM yang digunakan dalam komputer adalah DRAM dan SRAM.

DRAM dan SRAM keduanya adalah *volatile*; yaitu, daya harus terus-menerus dipasok ke memori untuk mempertahankan nilai bit. Sel memori dinamis lebih sederhana dan lebih kecil dari sel memori statis. Jadi, DRAM lebih padat (sel lebih kecil = lebih banyak sel per satuan luas) dan lebih murah daripada SRAM yang sesuai. Di sisi lain, DRAM membutuhkan sirkuit penyegaran (refresh circuitry) pendukung. Untuk memori yang lebih besar, biaya tetap dari sirkuit penyegaran lebih dari dikompensasi oleh biaya variabel yang lebih kecil dari sel-sel DRAM. Dengan demikian, DRAM cenderung disukai untuk kebutuhan memori yang besar. Poin terakhir adalah bahwa SRAM agak lebih cepat dari DRAM. Karena karakteristik relatif ini, SRAM digunakan untuk memori *cache* (baik *on-chip* maupun *off-chip*), dan DRAM digunakan untuk memori utama.

Tabel 2.6 menunjukkan type memori semikonduktor yang umum digunakan pada sistem komputer berserta karakteristiknya.

Tabel 2.6. Type Memori Semikonduktor

Type Memori	Kategori	Penghapusan	Mekanisme Tulis	Volatilitas
RAM	Memori baca-tulis	<i>Electrically byte level</i>	<i>Electrically</i>	<i>Volatile</i>
ROM	Memori hanya baca	Tidak mungkin	<i>Mask</i>	<i>Nonvolatile</i>
PROM				
EPROM	Read mostly memory	Sinar ultraviolet, chip level	<i>Electrically</i>	<i>Nonvolatile</i>
EEPROM		<i>Electrically, byte level</i>		
Flash Memory		<i>Electrically, block level</i>		

Sumber: (Brey, 2009)

D. Dynamic RAM (DRAM).

Dinamic RAM (DRAM) dibuat dengan sel memori yang menyimpan data sebagai muatan pada kapasitor. Ada atau tidak adanya muatan dalam kapasitor diartikan sebagai biner 1 atau 0. Karena kapasitor memiliki kecenderungan alami untuk dikeluarkan, RAM dinamis memerlukan penyegaran muatan berkala untuk menjaga penyimpanan data pada sel memori yang hanya dapat menampung data 2 sampai 4 ms. Setelah 2 sampai 4 ms, isi sel memori yang terbuat dari kapasitor akan kehilangan muatannya, sehingga harus ditulis kembali (*refresh*). Istilah dinamis mengacu pada kecenderungan muatan yang tersimpan ini bocor, bahkan dengan daya yang terus menerus diterapkan. Baris alamat harus diaktifkan ketika nilai bit dari sel ini dibaca atau ditulis. Transistor bertindak sebagai saklar yang ditutup (memungkinkan arus mengalir) jika tegangan diterapkan ke jalur alamat dan terbuka (tidak ada arus mengalir) jika tidak ada tegangan pada garis alamat. Meskipun sel DRAM digunakan untuk menyimpan bit tunggal (0 atau 1), itu pada dasarnya adalah perangkat analog. Kapasitor dapat menyimpan nilai muatan apa pun dalam rentang waktu tertentu, nilai ambang menentukan apakah muatan ditafsirkan sebagai 1 atau 0.

DRAM TMS4464 adalah sebuah DRAM 64K x 4 bit, yang dapat menyimpan 256 KB data. Gambar 2.16 menunjukkan susunan 16 pin pada memori ini. Dari gambar tersebut dapat diliha bahwa memori ini memiliki 8 bit *input* alamat, satu saluran *input* data, dan satu saluran *output* data. Untuk menjangkau 256 K lokasi data, memori ini seharusnya memiliki 16 bit alamat. Namun dengan cara melakukan dua kali pembacaan alamat melalui 8 bit *input* yang tersedia maka memori ini mendapatkan alamat valid 16 bit. Operasi ini membutuhkan dua pin khusus, yaitu: *Column Address Strobe (CAS)* dan *Row Address Strobe (RAS)*, yang keduanya merupakan sinyal aktif rendah. Pertama, A0 sampai A7 ditempatkan pada pin-pin alamat dan di-strobe ke dalam *latch* baris internal *RAS* sebagai *row address*. Berikutnya,

bit alamat A8 sampai A15 ditempatkan pada delapan pin alamat yang sama dan di-strobe kedalam sebuah latch kolom internal oleh **CAS**. Alamat 16 bit yang berada di dalam latch internal ini menjadi alamat valid 16 bit yang menentukan salah satu lokasi memori 4 bit. Perhatikan, bahwa **CAS** juga menunjukkan fungsi dari *input* seleksi *chip* ke DRAM. Gambar 2.17 mengilustrasikan bagaimana alamat valid dimulplex pada DRAM TMS4464.

A ₈	1	16	GND
D _{IN}	2	15	<u>CAS</u>
<u>WR</u>	3	14	D _{OUT}
RAS	4	13	A ₆
A ₀	5	12	A ₃
A ₁	6	11	A ₄
A ₂	7	10	A ₅
V _{CC}	8	9	A ₇

A₀-A₁₁ = Pin alamat

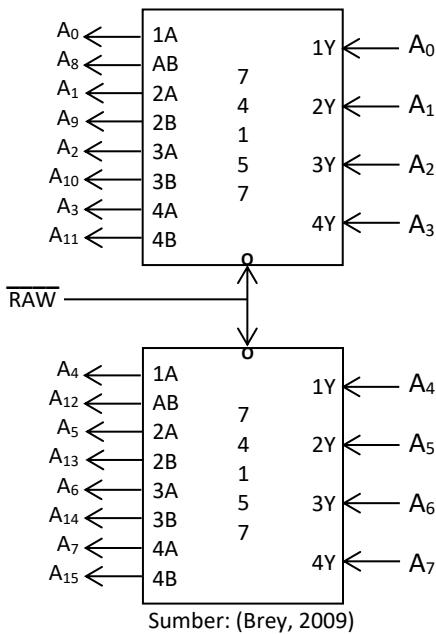
D_{IN} = Data masuk

D_{OUT} = Data keluar

WR = Write enable (aktif rendah)

Sumber: (Brey, 2009)

Gambar 2.16. Pin-out DRAM TMS4464



Sumber: (Brey, 2009)

Gambar 2.17. Multiplekser alamat untuk DRAM TMS4464

E. Static RAM (SRAM)

Sebaliknya, *Static RAM* (SRAM) adalah perangkat digital yang menggunakan elemen logika yang sama yang digunakan dalam prosesor. Dalam SRAM, nilai-nilai biner disimpan menggunakan konfigurasi gerbang logika flip-flop tradisional. RAM statis akan menyimpan datanya sementara (selama daya dipasok ke sana), dan digunakan untuk kebutuhan kapasitas memori yang relatif kecil.

SRAM TMS4016 adalah salah satu jenis SRAM 2K x 8 Bit. Memori ini memiliki 11 *input* alamat, 8 pin *input/output* data. Gambar 2.18 mengilustrasikan pin yang terdapat pada *chip* SRAM TMS4016.

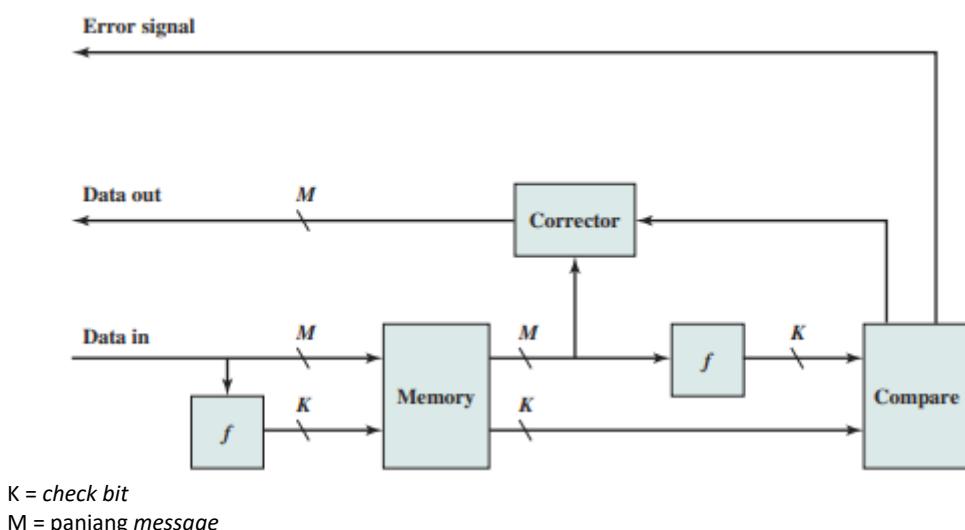
A ₇	1	24	V _{cc}
A ₆	2	23	A ₉
A ₅	3	22	A ₈
A ₄	4	21	W̄
A ₃	5	20	Ḡ
A ₂	6	19	A ₁₀
A ₁	7	18	S̄
A ₀	8	17	DQ ₇
DQ ₀	9	16	DQ ₆
DQ ₁	10	15	DQ ₅
DQ ₂	11	14	DQ ₄
V _{ss}	12	13	DQ ₃

A₀-A₁₁ = Pin alamat
 DQ₀-DQ₇ = Data masuk/keluar
 G = output enable
 S = Chip Select (aktif rendah)
 W = write enable
 Sumber: (Brey, 2009)

Gambar 2.18. Pin-out SRAM TMS4016

2.7. Deteksi Kesalahan

Kesalahan yang terjadi pada memori semikonduktor dapat dikategorikan sebagai *hard failure* dan *soft error*. *Hard failure* adalah cacat fisik permanen sehingga sel memori atau sel yang terpengaruh tidak dapat secara andal menyimpan data tetapi menjadi macet pada 0 atau 1 atau beralih secara tidak menentu antara 0 dan 1. *Hard failure* dapat disebabkan oleh penyalahgunaan lingkungan yang keras, cacat produksi, dan keausan. *Soft error* adalah peristiwa acak dan tidak merusak, yang mengubah isi dari satu atau lebih sel memori tanpa merusak fisik memori. *Soft error* dapat disebabkan oleh masalah catu daya atau partikel alfa. Partikel-partikel ini dihasilkan dari peluruhan radioaktif dan sangat umum karena inti radioaktif ditemukan dalam jumlah kecil di hampir semua bahan. Baik *hard failure* maupun *soft error* jelas tidak diinginkan, dan sebagian besar sistem memori utama modern menyertakan logika untuk mendekripsi dan memperbaiki kesalahan tersebut.



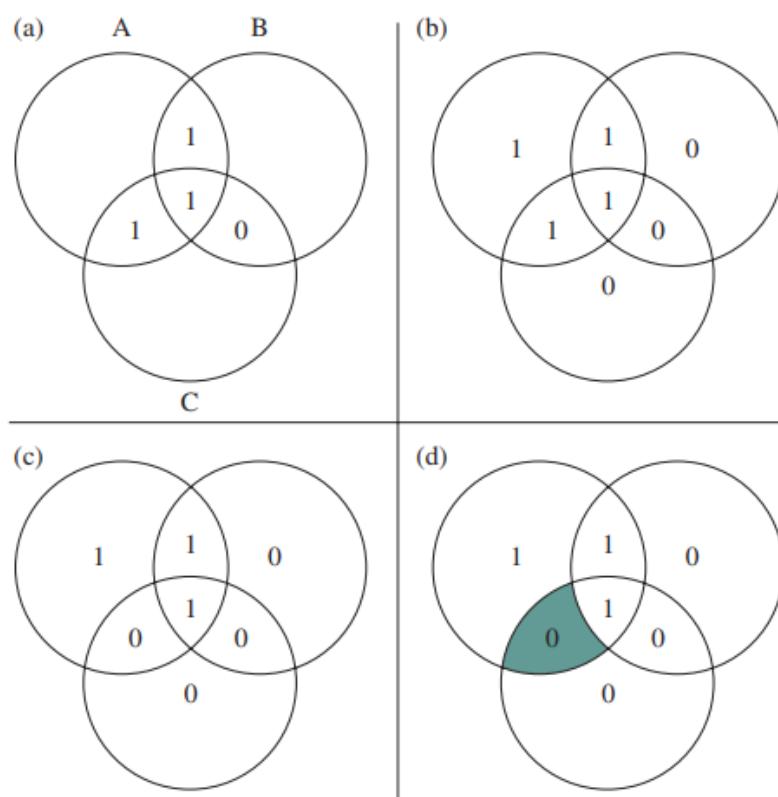
Sumber: (Stallings, 2016)

Gambar 2.19. Fungsi Error Correction Code

Gambar 2.19 menggambarkan secara umum bagaimana proses deteksi kesalahan dilakukan. Ketika data akan ditulis ke dalam memori, perhitungan, digambarkan sebagai fungsi f , dilakukan pada data untuk menghasilkan kode. Baik kode dan data disimpan. Dengan demikian, jika *word* data M-bit disimpan dan kodenya berukuran panjang K bit, maka ukuran sebenarnya dari word yang disimpan adalah $M+K$ bit

Ketika *word* yang disimpan sebelumnya dibacakan, kode tersebut digunakan untuk mendeteksi dan mungkin memperbaiki kesalahan. Seperangkat bit kode K baru dihasilkan dari bit data M dan dibandingkan dengan bit kode yang diambil. Perbandingan menghasilkan satu dari tiga hasil:

- 1) Tidak ada kesalahan yang terdeteksi. Bit data yang diambil dikirim.
 - 2) Kesalahan terdeteksi, dan ada kemungkinan untuk memperbaiki kesalahan. Bit data ditambah bit koreksi kesalahan dimasukkan ke korektor, yang menghasilkan set bit M yang dikoreksi untuk dikirim.
 - 3) Kesalahan terdeteksi, tetapi tidak mungkin untuk memperbaikinya. Kondisi ini dilaporkan.
-



Sumber: (Stallings, 2016)

Gambar 2.20. Hamming Code *Error Correction Code*

Kode yang beroperasi dengan cara ini disebut sebagai *Error Correction Code* (ECC). Kode ditandai oleh jumlah kesalahan bit dalam sebuah *word* yang dapat dikoreksi dan dideteksi. Kode koreksi kesalahan yang paling sederhana adalah Hamming Code yang dibuat oleh Richard Hamming di Bell Laboratories. Gambar 2.20 menunjukkan diagram Venn untuk menggambarkan penggunaan ECC pada *words* 4-bit ($M=4$). Dengan tiga lingkaran berpotongan, ada tujuh kompartemen. Terdapat 4 bit data pada kompartemen bagian dalam (Gambar 2.20a). Kompartemen yang tersisa diisi dengan apa yang disebut bit paritas. Setiap bit paritas dipilih sehingga jumlah total 1s dalam lingkarannya adalah genap (Gambar 2.20b). Dengan demikian, karena lingkaran A mencakup tiga data 1, bit paritas dalam lingkaran itu diatur ke 1. Sekarang, jika kesalahan mengubah salah satu dari bit data (Gambar 2.20c), itu mudah ditemukan.

Dengan memeriksa bit paritas, perbedaan ditemukan dalam lingkaran A dan lingkaran C tetapi tidak dalam lingkaran B. Hanya satu dari tujuh kompartemen dalam A dan C tetapi tidak B (Gambar 2.20d). Kesalahan karena itu dapat diperbaiki dengan mengubah bit itu. Konsep yang digunakan adalah dengan menggunakan kode yang dapat mendeteksi dan memperbaiki kesalahan bit tunggal dalam *words* 8-bit. Untuk memulai, tentukan berapa jumlah bit pada *error code* yang dibutuhkan. Mengacu pada Gambar 2.20, *block compare* menerima dua *input* (dua nilai K-bit). Bit demi bit dibandingkan dengan menggunakan eksklusif-OR dari dua *input* tersebut, hasilnya disebut *syndrome word*. Dengan demikian, setiap bit dari *syndrome word* dapat bernilai 0 atau 1 tergantung inputnya sama atau berbeda. Untuk mengingatkan kembali, eksklusif-OR dua input, akan menghasilkan output 0 jika dan hanya jika semua inputnya bernilai sama.

Metode hamming code bekerja dengan menyisipkan *check bit* K ke *message* M. Jumlah *check bit* yang disisipkan tergantung pada panjang *message*. Rumus untuk menghitung jumlah *check bit* yang akan disisipkan ke dalam *messege* 2^n bit adalah $c=n+1$, dimana c adalah jumlah *check bit* yang disisipkan.

Syndrome word dengan panjang K bit, memiliki kisaran antara 0 sampai 2^K-1 . Nilai 0 menunjukkan bahwa tidak ada kesalahan yang terdeteksi. Namun kesalahan dapat saja terjadi pada salah satu bit data M atau bit cek K, untuk mendeteksi terjadinya kesalahan bit, setidaknya harus memiliki kondisi dimana $2^K-1 \geq M \pm K$. Ketidaksetaraan ini memberikan jumlah bit yang diperlukan untuk memperbaiki kesalahan bit tunggal dalam *word* yang mengandung bit data M. Sebagai contoh, untuk *word* 8 bit ($M = 8$):

$$K = 3: 2^3-1 < 8 + 3$$

$$K = 4: 2^4-1 > 8 + 4$$

Dengan demikian, delapan bit data membutuhkan empat *check bit*. Tiga kolom pertama dari Tabel 2.7 mencantumkan jumlah *check bits* yang diperlukan untuk berbagai panjang *word* data. Pada kasus dimana *sindrom word* 4-bit untuk data *word* 8-bit, memiliki karakteristik berikut:

- 1) Jika *sindrom word* berisi semua 0, tidak ada kesalahan yang terdeteksi.
- 2) Jika *sindrom word* berisi satu dan hanya satu bit yang diatur ke 1, maka kesalahan telah terjadi di salah satu dari 4 bit periksa. Tidak diperlukan koreksi.
- 3) Jika *sindrom word* mengandung lebih dari satu bit yang ditetapkan ke 1, maka nilai numerik dari *sindrom word* menunjukkan posisi bit data dalam kesalahan. Bit data ini dibalik untuk koreksi.

Tabel 2.7. Penambahan lebar *word* dengan *Error Correction*

Data bits	<i>Single-Error Correction</i>		<i>Single-Error correction/ Double-Error Detection</i>	
	Check bits	% Increase	Check bits	% Increase
8	4	50	5	62.5
16	5	31.25	6	37.5
32	6	18.75	7	21.875
64	7	10.94	8	12.5
128	8	6.25	9	7.03
256	9	3.52	10	3.91

Sumber: (Stallings, 2016)

3. Eksternal Memori

3.1. Magnetik Disk

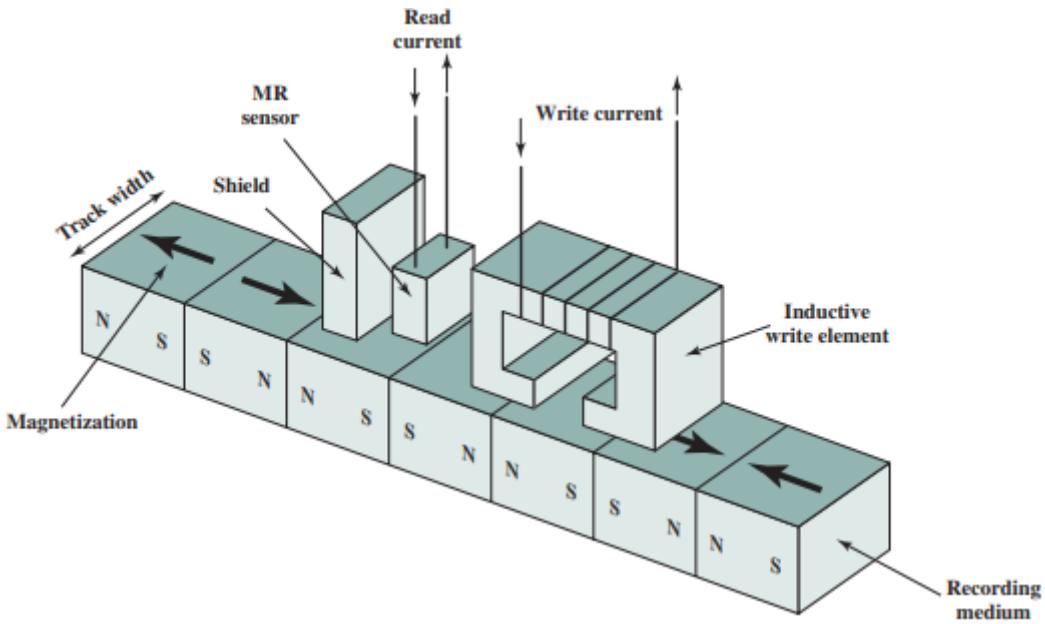
Disk adalah piringan bundar yang terbuat dari bahan nonmagnetik, yang disebut media, dilapisi *substrate* dengan bahan yang dapat dimagnetisasi. Secara tradisional, *substrate* terbuat dari bahan aluminium atau paduan aluminium. Selanjutnya, *substrate* dibuat dari bahan 68ptic (kaca) yang memiliki sejumlah kelebihan, yaitu:

- 1) Peningkatan keseragaman permukaan film magnetik untuk meningkatkan keandalan disk.
- 2) Pengurangan yang signifikan pada keseluruhan cacat permukaan untuk membantu mengurangi kesalahan baca-tulis.
- 3) Kemampuan untuk mendukung lower fly heights.
- 4) Kekakuan yang lebih baik untuk mengurangi dinamika disk.
- 5) Kemampuan yang lebih besar untuk menahan guncangan dan kerusakan.

Mekanisme Baca dan Tulis Magnetik Disk.

Pada magnetik disk, data direkam dan kemudian diambil dari disk melalui koil konduktor bernama *head*; dalam banyak sistem, ada dua *head*, *head* baca dan *head* tulis. Selama operasi membaca atau menulis, *head* bersifat stationer, sementara piring berputar di bawahnya. Mekanisme penulisan mengeksplorasi fakta bahwa listrik yang mengalir melalui koil menghasilkan medan magnet. Pulsa listrik dikirim ke *head* tulis, dan pola magnetik yang dihasilkan dicatat pada permukaan di bawah ini, dengan pola yang berbeda untuk arus positif dan negatif. *Head* tulis itu sendiri terbuat dari bahan yang mudah magnet dan dalam bentuk donat persegi panjang dengan celah di satu sisi dan beberapa putaran kawat pengantar di sisi yang berlawanan (Gambar 3.1). Arus listrik di kawat menginduksi medan magnet melintasi celah, yang pada gilirannya menarik medan kecil dari media perekaman. Membalikkan arah arus membalikkan arah magnetisasi pada media perekaman.

Mekanisme baca tradisional mengeksplorasi fakta bahwa medan magnet yang bergerak relatif terhadap koil menghasilkan arus listrik dalam koil. Ketika permukaan disk berputar di bawah *head*, itu menghasilkan arus dengan polaritas yang sama dengan yang sudah direkam. Struktur *head* untuk membaca dalam hal ini pada dasarnya sama dengan atau menulis dan oleh karena itu *head* yang sama dapat digunakan untuk keduanya. *Head* tunggal semacam itu digunakan dalam sistem floppy disk dan pada sistem disk yang lebih lama kaku.



Sumber: (Stallings, 2016)

Gambar 3.1. Magnetisasi Read/Inductive Write pada Head

Sistem disk kaku kontemporer menggunakan mekanisme baca berbeda, membutuhkan *head* baca terpisah, untuk kenyamanan diposisikan dekat dengan *head* tulis. *Head* baca terdiri dari sensor *Magnetoresistive Read* (MR) yang memiliki hambatan listrik tergantung pada arah magnetisasi medium yang bergerak di bawahnya. Dengan melewatkannya arus melalui sensor MR, perubahan resistansi terdeteksi sebagai sinyal tegangan. Desain MR memungkinkan operasi frekuensi tinggi, yang setara dengan kepadatan penyimpanan yang lebih besar dan kecepatan operasi.

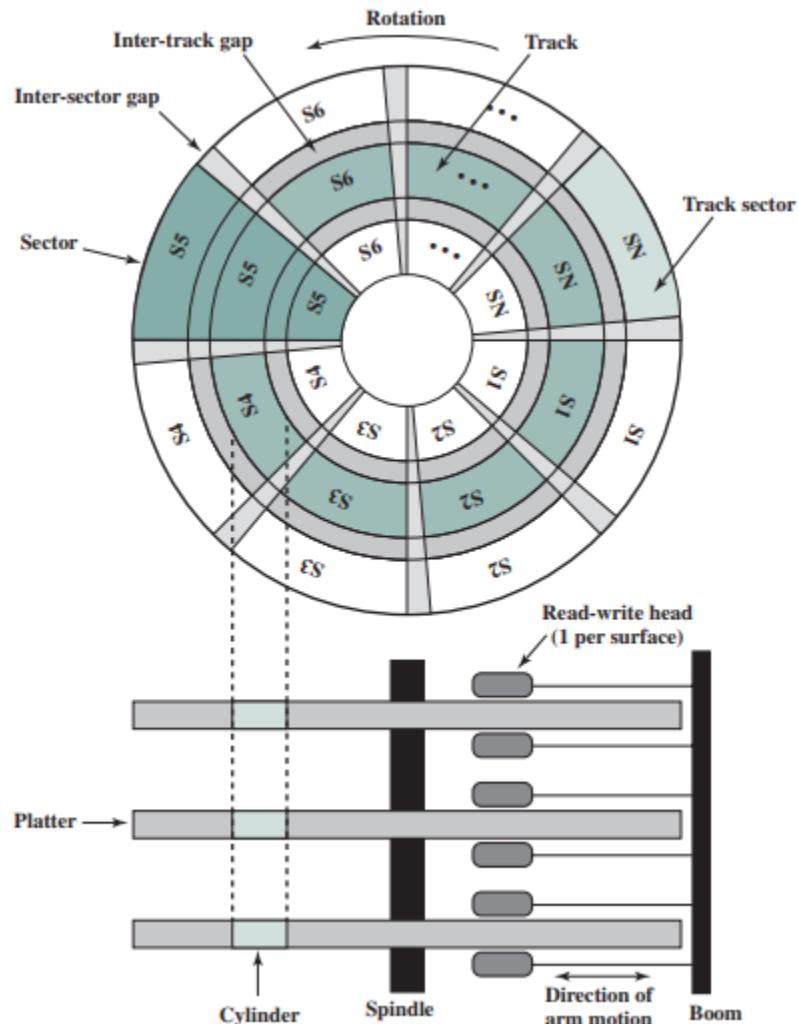
Organisasi dan Pemformatan Data

Head adalah perangkat yang relatif kecil yang mampu membaca dari atau menulis ke bagian piring yang berputar di bawahnya. Ini memunculkan pengorganisasian data pada piringan dalam satu set cincin konsentris, yang disebut *track*. Setiap *track* memiliki lebar yang sama dengan *head*. Ada ribuan *track* per permukaan.

Gambar 3.2 menggambarkan tata letak data ini. *Track* yang berdekatan dipisahkan oleh celah *inter-track*. Ini mencegah, atau setidaknya meminimalkan, kesalahan karena ketidaksejajaran *head* atau hanya gangguan medan magnet. Data ditransfer ke dan dari disk di *sector*. Biasanya ada ratusan sektor per *track*, dan ini mungkin memiliki panjang tetap atau variabel. Dalam kebanyakan sistem kontemporer, sektor dengan panjang tetap digunakan, dengan 512 byte adalah ukuran sektor yang hampir universal. Untuk menghindari penerapan persyaratan presisi yang tidak masuk akal pada sistem, sektor-sektor yang berdekatan dipisahkan oleh celah intersektor.

Sedikit di dekat pusat disk yang berputar melewati titik tetap (seperti *head* baca-tulis) lebih lambat daripada sedikit di luar. Oleh karena itu, beberapa cara harus ditemukan untuk mengimbangi variasi dalam kecepatan sehingga *head* dapat membaca semua bit pada kecepatan yang sama. Ini dapat dilakukan dengan mendefinisikan jarak variabel antara bit informasi yang direkam di lokasi pada disk, dengan cara bahwa *track* terluar memiliki sektor dengan jarak yang lebih besar. Informasi tersebut kemudian dapat dipindai pada kecepatan yang sama dengan memutar disk pada kecepatan tetap, yang dikenal sebagai *constant angular velocity* (CAV). Gambar 3.3a menunjukkan tata letak disk menggunakan CAV. Disk dibagi menjadi beberapa sektor berbentuk pie dan menjadi serangkaian *track* konsentris. Keuntungan menggunakan CAV adalah bahwa setiap blok data dapat langsung diatasi dengan jalur dan sektor. Untuk memindahkan *head* dari lokasi saat ini ke alamat tertentu, hanya diperlukan gerakan pendek *head* ke *track* tertentu dan menunggu sebentar untuk sektor yang tepat

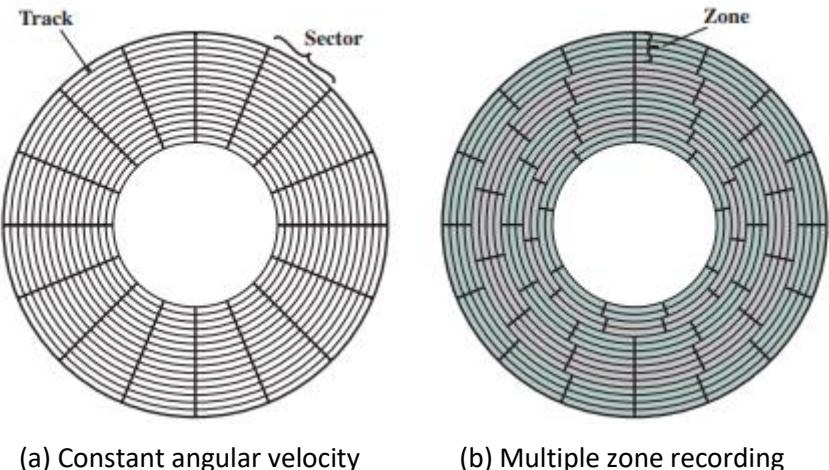
berputar di bawah *head*. Kekurangan CAV adalah bahwa jumlah data yang dapat disimpan pada *track* luar yang panjang adalah sama dengan apa yang dapat disimpan pada *track* dalam yang pendek.



Sumber: (Stallings, 2016)
Gambar 3.2. Layout Disk Data

Karena densitas, dalam bit per inch linier, meningkat dalam perpindahan dari *track* terluar ke *track* terdalam, kapasitas penyimpanan disk dalam sistem CAV langsung dibatasi oleh kepadatan perekaman maksimum yang dapat dicapai pada *track* terdalam. Untuk memaksimalkan kapasitas penyimpanan, akan lebih disukai untuk memiliki kerapatan bit linier yang sama di setiap *track*. Ini akan membutuhkan sirkuit rumit yang tidak dapat diterima. Sistem hard disk modern menggunakan teknik yang lebih sederhana, yang mendekati kerapatan bit yang sama per *track*, yang dikenal sebagai *Multiple Zone Recording* (MZR), di mana permukaannya dibagi menjadi sejumlah zona konsentris (pada umumnya 16 konsentris). Setiap zona berisi sejumlah *track* yang berdekatan, biasanya dalam ribuan. Dalam suatu zona, jumlah bit per *track* adalah konstan. Zona yang lebih jauh dari pusat mengandung lebih banyak bit (lebih banyak sektor) daripada zona yang lebih dekat ke pusat. Zona didefinisikan sedemikian rupa sehingga kerapatan bit linier kira-kira sama pada semua *track* disk. MZR memungkinkan kapasitas penyimpanan keseluruhan yang lebih besar dengan mengorbankan sirkuit yang sedikit lebih kompleks. Saat *head disk* bergerak dari satu zona ke zona lain, panjang (di sepanjang jalur) dari masing-masing bit berubah, menyebabkan perubahan waktu untuk membaca dan menulis.

Gambar 3.3b adalah tata letak MZR yang disederhanakan, dengan 15 *track* disusun dalam 5 zona. Dua zona terdalam masing-masing memiliki dua *track*, dengan masing-masing *track* memiliki sembilan sektor; zona berikutnya memiliki 3 *track*, masing-masing dengan 12 sektor; dan 2 zona terluar masing-masing memiliki 4 *track*, dengan masing-masing *track* memiliki 16 sektor.

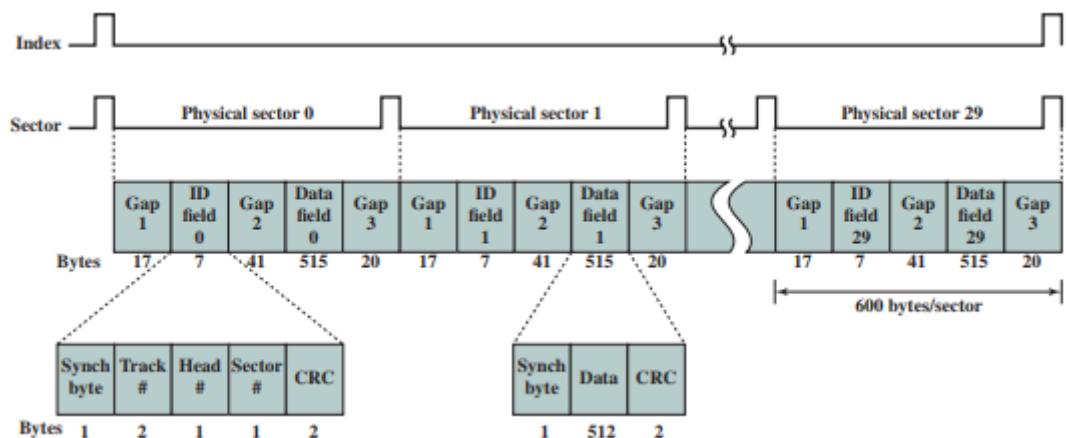


Sumber: (Stallings, 2016)

Gambar 3.3. Perbandingan Metode Layout Disk

Beberapa cara diperlukan untuk menemukan posisi sektor dalam suatu *track*. Harus ada titik awal di *track* dan cara mengidentifikasi awal dan akhir setiap sektor. Cara ini dilakukan dengan cara mengontrol data yang direkam pada disk. Dengan demikian, disk diformat dengan beberapa data tambahan yang hanya digunakan oleh drive disk dan tidak dapat diakses oleh pengguna.

Contoh format disk ditunjukkan pada Gambar 3.4. Dalam hal ini, setiap *track* berisi 30 sektor dengan panjang tetap masing-masing 600 byte. Setiap sektor menyimpan 512 byte data plus informasi kontrol yang berguna untuk pengontrol disk. Bidang ID adalah pengidentifikasi atau alamat unik yang digunakan untuk menemukan sektor tertentu. Byte SYNCH adalah pola bit khusus yang membatasi awal bidang. Nomor *track* mengidentifikasi *track* di permukaan. Nomor *head* mengidentifikasi *head*, karena disk ini memiliki beberapa permukaan. Bidang ID dan data masing-masing berisi kode pendekripsi kesalahan.



Sumber: (Stallings, 2016)

Gambar 3.4. Format Winchester Disk (Seagate ST506)

Karakter Fisik

Tabel 3.1 mencantumkan karakteristik utama yang membedakan antara berbagai jenis disk magnetik. Pertama, *head* dapat dipasang atau digerakkan sehubungan dengan arah radial piringan. Dalam disk *fixed-head*, ada satu *head read-write* per *track*. Semua *head* dipasang pada lengan kaku yang memanjang di semua *track*; sistem seperti itu jarang terjadi saat ini. Di *disk head* bergerak, hanya ada satu *head baca-tulis*. Sekali lagi, *head* dipasang pada lengan. Karena *head* harus dapat diposisikan di atas *track* apa pun, lengan dapat diperpanjang atau ditarik untuk tujuan ini.

Disk itu sendiri dipasang di *disk drive*, yang terdiri dari lengan, poros yang memutar disk, dan elektronik yang diperlukan untuk *input* dan *output* data biner. Disk yang tidak dapat dilepas dipasang secara permanen di drive disk; hard disk di komputer pribadi adalah disk yang tidak dapat dilepas. Disk yang dapat dilepas dapat dihapus dan diganti dengan disk lain. Keuntungan dari tipe yang terakhir adalah bahwa jumlah data yang tidak terbatas tersedia dengan jumlah sistem disk yang terbatas. Selanjutnya, disk tersebut dapat dipindahkan dari satu sistem komputer ke yang lain. *Floppy disk* dan *ZIP cartridge disks* adalah contoh dari *removable disk*.

Tabel 3.1. Karakteristik Fisik dari Disk System

Head Motion	Plates
<i>Fixed head (satu head per track)</i>	<i>Single platter</i>
<i>Movable head (satu head per surface)</i>	<i>Multiple platter</i>
Disk Portability	Head Mechanism
<i>Nonremovable disk</i>	<i>Contact (floppy)</i>
<i>Removable disk</i>	<i>Fixed gap</i>
Sides	<i>Aerodinamyc gap (winchester)</i>
<i>Single side</i>	
<i>Double side</i>	

Sumber: (Stallings, 2016)

Untuk sebagian besar disk, lapisan yang dapat di magnetkan diaplikasikan pada kedua sisi platter, yang kemudian disebut sebagai sisi ganda. Beberapa sistem disk yang lebih murah menggunakan disk satu sisi. Beberapa disk drive menampung banyak piring yang disusun secara vertikal, hanya sebagian kecil dari satu inch. Disk *multi-platter* menggunakan *head* yang dapat digerakkan, dengan satu *head read-write* per permukaan platter. Semua *head* diperbaiki secara mekanis sehingga semua berada pada jarak yang sama dari pusat disk dan bergerak bersama. Dengan demikian, setiap saat, semua *head* diposisikan di atas *track* yang berjarak sama dari pusat disk. Himpunan semua *track* dalam posisi relatif yang sama pada plat disebut sebagai silinder. Ini diilustrasikan pada Gambar 3.3. Akhirnya, mekanisme *head* menyediakan klasifikasi disk menjadi tiga jenis.

Secara tradisional, *head baca-tulis* telah diposisikan jarak tetap di atas piring, memungkinkan celah udara. Di sisi lain ekstrim adalah mekanisme *head* yang benar-benar datang ke dalam kontak fisik dengan media selama operasi baca atau tulis. Mekanisme ini digunakan dengan floppy disk, yang merupakan piringan kecil yang fleksibel dan jenis disk yang paling murah.

Tipe disk ketiga, berkaitan dengan hubungan antara kepadatan data dan ukuran celah udara (*air gap*). *Head* harus menghasilkan atau merasakan medan elektromagnetik yang cukup besar untuk menulis dan membaca dengan benar.

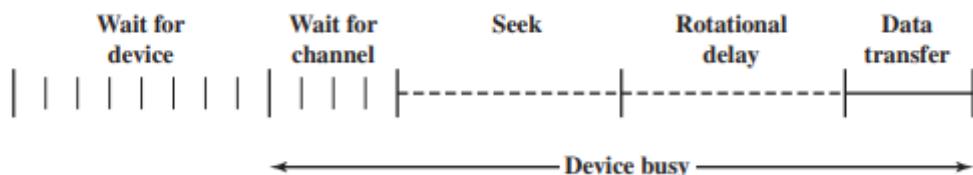
Semakin sempit *head*, semakin dekat dengan permukaan *platter* agar berfungsi. Ukuran *head* yang kecil, diikuti dengan ukuran *track* yang kecil, dan akan memberikan kepadatan data yang lebih besar. Namun, semakin dekat *head* ke disk, semakin besar risiko kesalahan. Disk Winchester dikembangkan untuk mendapatkan teknologi yang lebih baik. *Head Winchester* digunakan dalam rakitan drive tersegel yang hampir bebas dari kontaminasi dari luar. Mereka dirancang untuk beroperasi lebih dekat ke permukaan disk daripada *head* disk konvensional yang kaku, sehingga memungkinkan kepadatan

data yang lebih besar. *Head* sebenarnya adalah foil aerodinamik yang bersandar ringan pada permukaan platter ketika disk tidak bergerak. Tekanan udara yang dihasilkan oleh piringan berputar cukup untuk membuat foil naik di atas permukaan. Sistem nonkontak yang dihasilkan dapat direkayasa untuk menggunakan *head* yang lebih sempit yang beroperasi lebih dekat ke permukaan platter daripada *head* disk konvensional.

Parameter Kinerja Disk

Rincian aktual dari operasi I/O disk tergantung pada sistem komputer, sistem operasi, dan sifat saluran I/O serta perangkat keras pengontrol disk. Diagram timing umum transfer I/O disk ditunjukkan pada Gambar 3.5.

Saat disk drive beroperasi, disk berputar dengan kecepatan konstan. Untuk membaca atau menulis, *head* harus diposisikan di *track* dan awal sektor yang diinginkan pada *track* itu. Pemilihan *track* melibatkan gerakan *head* dalam sistem *head-moveable* atau secara elektronik memilih satu *head* pada sistem *fixed-head*.



Sumber: (Stallings, 2016)

Gambar 3.5. Pewaktu I/O Transfer pada Disk

Pada sistem *moveable head*, waktu yang diperlukan untuk memposisikan *head* di *track* dikenal sebagai *seek time*. Dalam kedua kasus, setelah *track* dipilih, pengontrol disk menunggu hingga sektor yang sesuai berputar untuk sejajar dengan *head*. Waktu yang diperlukan bagi awal sektor untuk mencapai *head* dikenal sebagai penundaan rotasi, atau latensi rotasi. Jumlah waktu pencarian, jika ada, dan penundaan rotasi sama dengan waktu akses, yang merupakan waktu yang diperlukan untuk masuk ke posisi untuk membaca atau menulis. Setelah *head* berada di posisi, operasi baca atau tulis dilakukan saat sektor bergerak di bawah *head*; ini adalah bagian transfer data dari operasi; waktu yang diperlukan untuk transfer adalah waktu transfer.

Selain waktu akses dan waktu transfer, ada beberapa penundaan antrian yang biasanya terkait dengan operasi I/O disk. Ketika suatu proses mengeluarkan permintaan I/O, itu harus terlebih dahulu menunggu dalam antrian untuk perangkat yang akan tersedia. Pada saat itu, perangkat ditugaskan untuk proses tersebut. Jika perangkat berbagi saluran I/O tunggal atau satu set saluran I/O dengan drive disk lain, maka mungkin ada menunggu tambahan untuk saluran yang akan tersedia. Pada titik itu, pencarian dilakukan untuk memulai akses disk.

Dalam beberapa sistem *high-end* untuk server, teknik yang digunakan dikenal sebagai *Rotational Positional Sensing* (RPS). Ini berfungsi sebagai berikut: Ketika perintah *search* telah dikeluarkan, saluran dilepaskan untuk menangani operasi I/O lainnya. Ketika *search* selesai, perangkat menentukan kapan data akan diputar di bawah *head*. Ketika sektor itu mendekati *search*, perangkat mencoba untuk membangun kembali jalur komunikasi kembali ke tuan rumah. Jika *Control Unit* atau saluran sibuk dengan I/O lain, maka upaya koneksi ulang gagal dan perangkat harus memutar satu revolusi penuh sebelum dapat mencoba untuk menyambung kembali, yang disebut RPS *miss*. Ini adalah elemen keterlambatan tambahan yang harus ditambahkan ke garis waktu Gambar 3.5.

Timing Seek

Timing seek adalah waktu yang diperlukan untuk memindahkan lengan disk ke *track* yang diperlukan. Ternyata ini adalah jumlah yang sulit untuk dijabarkan. Waktu pencarian terdiri dari dua komponen utama: waktu *startup* awal, dan waktu yang diambil untuk melintasi *track* yang harus dilintasi begitu

akses akses mencapai kecepatan. Sayangnya, waktu traversal bukan fungsi linier dari jumlah *track*, tetapi termasuk waktu penyelesaian (waktu setelah posisi *head* di atas *track* target sampai identifikasi *track* dikonfirmasi).

Banyak peningkatan berasal dari komponen disk yang lebih kecil dan lebih ringan. Beberapa tahun yang lalu, cakram khas berdiameter 14 inch (36 cm), sedangkan ukuran paling umum saat ini adalah 3,5 inch (8,9 cm), mengurangi jarak yang harus dilalui lengkap. Rata-rata waktu pencarian rata-rata pada hard disk kontemporer adalah di bawah 10 ms.

Penundaan rotasi

Disk, selain floppy disk, berputar dengan kecepatan mulai dari 3600 rpm (untuk perangkat genggam seperti kamera digital) hingga, pada tulisan ini, 20.000 rpm; pada kecepatan terakhir ini, ada satu revolusi per 3 ms. Jadi, secara rata-rata, penundaan rotasi adalah 1,5 ms.

Transfer Time

Waktu transfer ke atau dari disk tergantung pada kecepatan rotasi disk dengan cara berikut:

$$T = \frac{b}{rN}$$

dimana

T = waktu transfer

b = jumlah byte yang akan ditransfer

N = jumlah byte di *track*

r = kecepatan rotasi, dalam putaran per detik

Dengan demikian total rata-rata waktu baca atau tulis T_{total} dapat dinyatakan sebagai:

$$T_{Total} = T_s + \frac{1}{2r} + \frac{b}{rN}$$

dimana T_s adalah rata-rata waktu pencarian.

Timing Comparison

Dengan parameter yang ditentukan sebelumnya, dapat dilihat dua operasi I/O berbeda yang menggambarkan bahaya mengandalkan nilai rata-rata. Pertimbangkan disk dengan waktu pencarian rata-rata yang diiklankan 4 ms, kecepatan putaran 15.000 rpm, dan sektor 512-byte dengan 500 sektor per *track*. Misalkan diinginkan membaca file yang terdiri dari 2500 sektor dengan total 1,28 Mbytes. Untuk memperkirakan total waktu transfer, pertama-tama, diasumsikan bahwa file tersebut disimpan seluruh mungkin pada disk. Artinya, file menempati semua sektor pada 5 *track* yang berdekatan (5 *track* * 500 sektor/*track* = 2500 sektor). Ini dikenal sebagai organisasi berurutan. Sekarang, waktu untuk membaca *track* pertama adalah sebagai berikut:

Average seek	4	ms
Rotational delay	2	ms
Read 1 sectors	<u>0.008</u>	ms
	6.008	ms

Total time = 2500 * 6.008 = 15,020 ms = 15.02 seconds

3.2. Redundant Array of Independent Disk (RAID)

Dengan penggunaan beberapa disk, ada berbagai cara di mana data dapat diorganisir dan dapat menambahkan redundansi untuk meningkatkan keandalan. Ini bisa menyulitkan pengembangan

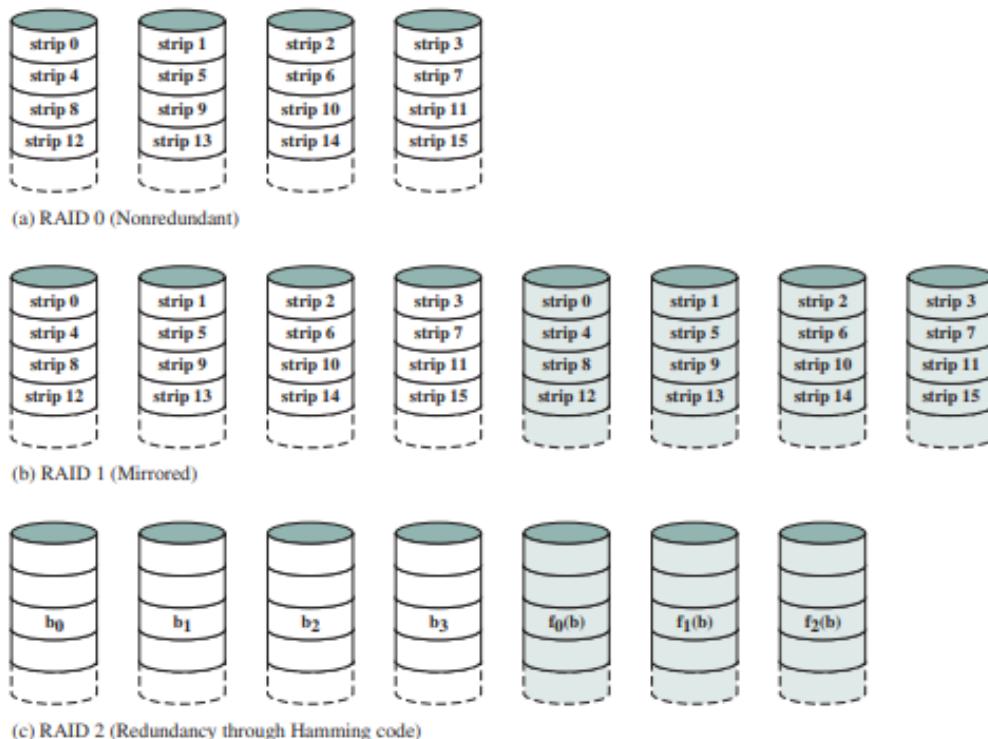
skema basis data yang dapat digunakan pada sejumlah *platform* dan sistem operasi. Skema RAID (*Redundant Array of Independent Disk*) terdiri dari tujuh level, nol hingga enam. Level-level ini tidak menyiratkan hubungan hierarkis tetapi menunjuk arsitektur desain yang berbeda yang memiliki tiga karakteristik umum:

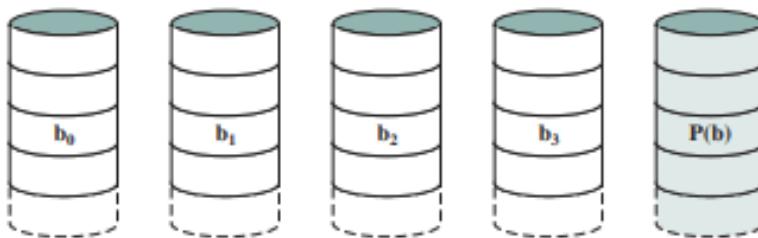
1. RAID adalah sekumpulan drive disk fisik yang dilihat oleh sistem operasi sebagai drive logika tunggal.
2. Data didistribusikan di seluruh drive fisik array dalam skema yang dikenal sebagai striping, dijelaskan selanjutnya.
3. Kapasitas disk berlebihan digunakan untuk menyimpan informasi paritas, yang menjamin pemulihan data jika terjadi kegagalan disk.

Rincian karakteristik kedua dan ketiga berbeda untuk tingkat RAID yang berbeda. RAID 0 dan RAID 1 tidak mendukung karakteristik ketiga.

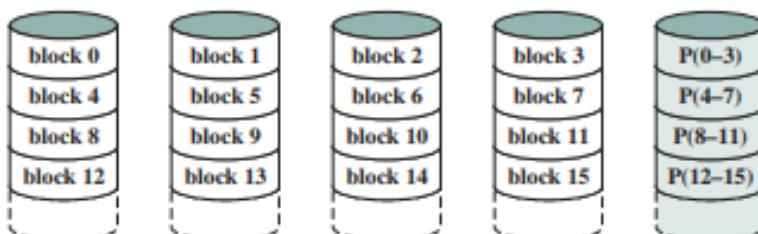
Istilah RAID awalnya muncul dalam makalah yang ditulis oleh sekelompok peneliti di University of California di Berkeley. RAID menggunakan beberapa disk drive dan mendistribusikan data sedemikian rupa sehingga memungkinkan akses simultan ke data dari banyak drive tersebut, sehingga meningkatkan kinerja I/O dan memungkinkan peningkatan kapasitas yang lebih mudah.

Kontribusi unik yang ditawarkan dari RAID adalah untuk mengatasi secara efektif kebutuhan redundansi. Meskipun memungkinkan beberapa *head* dan aktuator beroperasi secara simultan mencapai tingkat I/O dan transfer yang lebih tinggi, penggunaan beberapa perangkat meningkatkan kemungkinan kegagalan. Untuk mengimbangi penurunan keandalan ini, RAID menggunakan informasi paritas tersimpan yang memungkinkan pemulihan data yang hilang karena kegagalan disk.

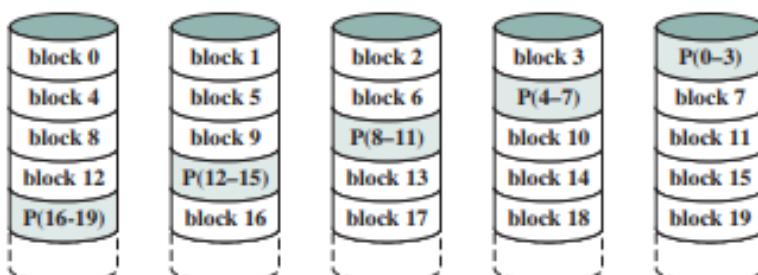




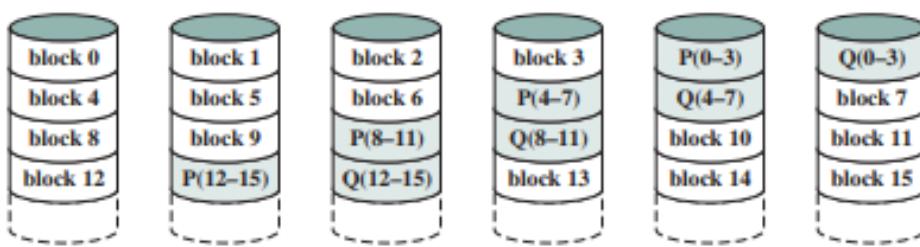
(d) RAID 3 (Bit-interleaved parity)



(e) RAID 4 (Block-level parity)



(f) RAID 5 (Block-level distributed parity)



(g) RAID 6 (Dual redundancy)

Sumber: (Stallings, 2016)
Gambar 3.6. Level RAID

Tabel 3.2 merangkum secara ringkas mengenai kinerja semua level RAID. Dalam tabel, kinerja I/O ditampilkan baik dalam hal kapasitas transfer data, atau kemampuan untuk memindahkan data, dan tingkat permintaan I/O, atau kemampuan untuk memenuhi permintaan I/O. Setiap titik kuat tingkat RAID disorot oleh naungan yang lebih gelap. Gambar 3.6 mengilustrasikan penggunaan tujuh skema RAID untuk mendukung kapasitas data yang membutuhkan empat disk tanpa redundansi. Angka-angka menyoroti tata letak data pengguna dan data yang berlebihan dan menunjukkan persyaratan penyimpanan relatif dari berbagai tingkatan. Dari tujuh level RAID yang dijelaskan, hanya empat yang umum digunakan: level RAID 0, 1, 5, dan 6.

Tabel 3.2. Level RAID

Level	Category	Description	Disks Required	Data Availability	Large I/O Data Transfer Capacity	Small I/O Request Rate
0	Striping	Nonredundant	N	Lower than single disk	Very high	Very high for both read and write
1	Mirroring	Mirrored	2N	Higher than RAID 2, 3, 4, or 5; lower than RAID 6	Higher than single disk for read; similar to single disk for write	Up to twice that of a single disk for read; similar to single disk for write
2	Parallel access	Redundant via Hamming code	N+m	Much higher than single disk; comparable to RAID 3, 4, or 5	Highest of all listed alternatives	Approximately twice that of a single disk
3		Bit-interleaved parity	N+1	Much higher than single disk; comparable to RAID 2, 4, or 5	Highest of all listed alternatives	Approximately twice that of a single disk
4	Independent access	Block-interleaved parity	N+1	Much higher than single disk; comparable to RAID 2, 3, or 5	Similar to RAID 0 for read; significantly lower than single disk for write	Similar to RAID 0 for read; significantly lower than single disk for write
5		Block-interleaved distributed parity	N+1	Much higher than single disk; comparable to RAID 2, 3, or 4	Similar to RAID 0 for read; lower than single disk for write	Similar to RAID 0 for read; generally lower than single disk for write
6		Block-interleaved dual distributed parity	N+2	Highest of all listed alternatives	Similar to RAID 0 for read; lower than RAID 5 for write	Similar to RAID 0 for read; significantly lower than RAID 5 for write

Note: N = number of data disks; m proportional to log N

Sumber: (Stallings, 2016)

Sumber: (Stallings, 2016)

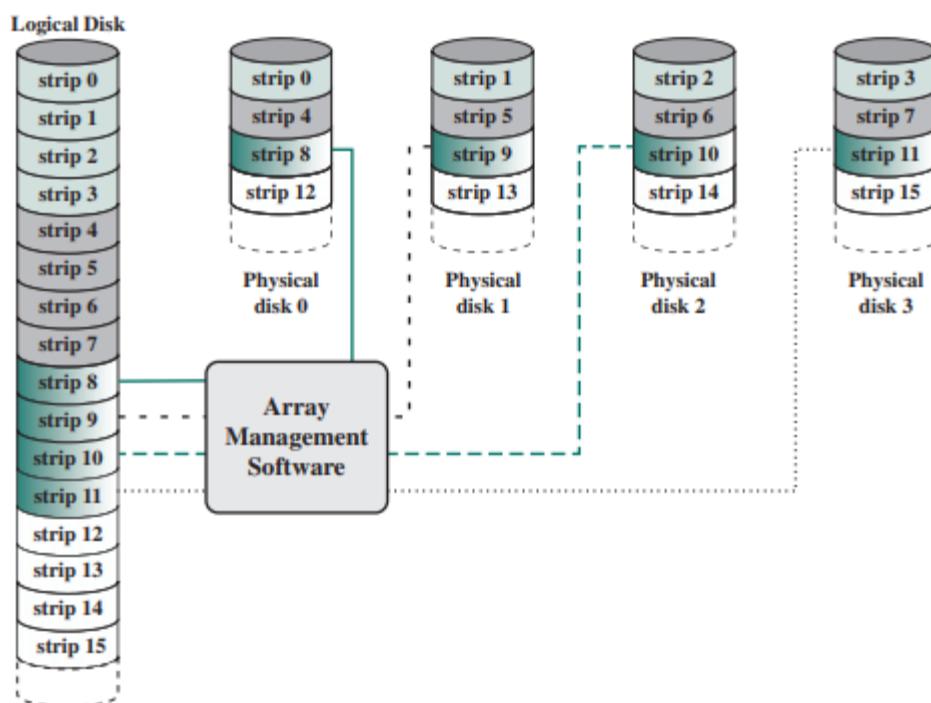
Level RAID 0

Level RAID 0 bukan anggota sejati keluarga RAID karena tidak termasuk redundansi untuk meningkatkan kinerja. Namun, ada beberapa aplikasi, seperti beberapa pada superkomputer di mana kinerja dan kapasitas menjadi perhatian utama dan biaya rendah lebih penting daripada peningkatan keandalan.

Untuk RAID 0, data pengguna dan sistem didistribusikan di semua disk dalam array. Ini memiliki keunggulan penting dibandingkan penggunaan disk besar tunggal: Jika permintaan I/O dua-duanya menunggu untuk dua blok data yang berbeda, maka ada kemungkinan besar bahwa blok yang diminta berada pada disk yang berbeda. Dengan demikian, dua permintaan dapat dikeluarkan secara paralel, mengurangi waktu antrian I/O.

Tetapi RAID 0, seperti halnya semua level RAID, melangkah lebih jauh dari sekadar mendistribusikan data melalui array disk: Data dilucuti di seluruh disk yang tersedia. Semua data pengguna dan sistem secara logika disimpan pada satu disk (disk logika). Disk logika dibagi menjadi beberapa strip; strip ini dapat berupa blok fisik, sektor, atau unit lain. Strip dipetakan dengan *round robin* ke disk fisik berturut-turut dalam array RAID. Satu set strip berturut-turut yang memetakan secara tepat satu strip untuk setiap anggota array disebut sebagai strip. Dalam sebuah array n-disk, strip logika pertama dan n secara fisik disimpan sebagai strip pertama pada masing-masing disk, membentuk strip pertama; strip kedua didistribusikan sebagai strip kedua pada setiap disk; dan seterusnya. Keuntungan dari tata letak ini adalah bahwa jika satu permintaan I/O terdiri dari beberapa strip yang berdekatan secara logika, maka hingga n strip untuk permintaan tersebut dapat ditangani secara paralel, sangat mengurangi waktu transfer I/O.

Gambar 3.7 menunjukkan penggunaan *Array Management Software* untuk memetakan antara ruang disk logika dan fisik pada RAID 0. Perangkat lunak ini dapat dijalankan di subsistem disk atau di komputer host.



Sumber: (Stallings, 2016)

Gambar 3.7. Data Mapping untuk Array RAID Level 0

Raid 0 Untuk Kapasitas Transfer Data Tinggi

Kinerja salah satu level RAID tergantung secara kritis pada pola permintaan sistem host dan tata letak data. Agar aplikasi mendapatkan tingkat transfer yang tinggi, dua persyaratan harus dipenuhi. Pertama, kapasitas transfer yang tinggi harus ada di sepanjang jalur antara memori host dan drive disk individual. Persyaratan kedua adalah aplikasi harus membuat permintaan I/O yang menggerakkan array disk secara efisien. Persyaratan ini dipenuhi jika permintaan akses data pada umumnya adalah untuk sejumlah besar data yang berdekatan secara logika, dibandingkan dengan ukuran strip. Dalam hal ini, permintaan I/O tunggal melibatkan transfer data paralel dari beberapa disk, meningkatkan kecepatan transfer efektif dibandingkan dengan transfer disk tunggal.

Raid 0 Untuk Tingkat Permintaan I/O Tinggi

Dalam lingkungan yang berorientasi transaksi, pengguna biasanya lebih mementingkan waktu respons daripada kecepatan transfer. Untuk permintaan I/O individu untuk sejumlah kecil data, waktu I/O didominasi oleh gerakan *head disk* (*seek time*) dan pergerakan disk (*rotasi latensi*).

Dalam lingkungan transaksi, mungkin ada ratusan permintaan I/O per detik. Array disk dapat memberikan tingkat eksekusi I/O yang tinggi dengan menyeimbangkan beban I/O di banyak disk. Perimbangan beban yang efektif hanya dilakukan jika ada beberapa permintaan I/O yang beredar. Ini, pada gilirannya, menyiratkan bahwa ada beberapa aplikasi independen atau aplikasi berorientasi transaksi tunggal yang mampu beberapa permintaan I/O asinkron. Kinerja juga akan dipengaruhi oleh ukuran strip. Jika ukuran strip relatif besar, sehingga satu permintaan I/O hanya melibatkan akses disk tunggal, maka beberapa permintaan I/O menunggu dapat ditangani secara paralel, mengurangi waktu antrian untuk setiap permintaan.

Level RAID 1

RAID 1 berbeda dari level RAID 2 hingga 6, dalam hal perhitungan redundansi. Dalam skema RAID lainnya, beberapa bentuk perhitungan paritas digunakan untuk memperkenalkan redundansi, sedangkan dalam RAID 1, redundansi dicapai dengan cara sederhana untuk menduplikasi semua data. Seperti ditunjukkan pada Gambar 3.6b, striping data digunakan, seperti pada RAID 0. Namun dalam kasus ini, setiap strip logika dipetakan ke dua disk fisik yang terpisah sehingga setiap disk dalam array memiliki disk mirror yang berisi data yang sama. RAID 1 juga dapat diimplementasikan tanpa striping data, meskipun ini kurang umum.

Ada sejumlah aspek positif pada organisasi RAID 1, yaitu:

1. Permintaan baca dapat dilayani oleh salah satu dari dua disk yang berisi data yang diminta, yang mana saja yang melibatkan waktu pencarian minimum ditambah latensi rotasi.
2. Permintaan penulisan mengharuskan kedua strip yang sesuai diperbarui, tetapi ini dapat dilakukan secara paralel. Dengan demikian, kinerja penulisan ditentukan oleh lebih lambat dari keduanya menulis (yaitu, yang melibatkan waktu pencarian yang lebih besar ditambah latensi rotasi). Namun, tidak ada "tulis penalti" dengan RAID 1. Level RAID 2 hingga 6 melibatkan penggunaan bit paritas. Oleh karena itu, ketika satu strip diperbarui, perangkat lunak manajemen array harus terlebih dahulu menghitung dan memperbarui bit paritas serta memperbarui strip aktual yang dimaksud.
3. Pemulihan dari kegagalan itu sederhana. Ketika drive gagal, data masih dapat diakses dari drive kedua.

Kekurangan utama RAID 1 adalah biaya, karena membutuhkan dua kali ruang disk dari disk logika yang didukungnya. Karena itu, konfigurasi RAID 1 kemungkinan akan terbatas pada drive yang menyimpan perangkat lunak dan data sistem dan file yang sangat penting lainnya. Dalam kasus ini, RAID 1 menyediakan salinan *real-time* dari semua data sehingga jika terjadi kegagalan disk, semua data penting dapat segera tersedia.

Dalam lingkungan yang berorientasi transaksi, RAID 1 dapat mencapai tingkat permintaan I/O yang tinggi jika sebagian besar permintaan dibaca. Dalam situasi ini, kinerja RAID 1 dapat mendekati dua kali lipat dari RAID 0. Namun, jika sebagian besar dari permintaan I/O adalah permintaan tulis, maka mungkin tidak ada peningkatan kinerja yang signifikan atas RAID 0. RAID 1 juga dapat memberikan peningkatan kinerja melalui RAID

O untuk aplikasi intensif transfer data dengan persentase baca yang tinggi. Peningkatan terjadi jika aplikasi dapat membagi setiap permintaan baca sehingga kedua anggota disk berpartisipasi.

Tingkat RAID 2

Level RAID 2 dan 3 menggunakan teknik akses paralel. Dalam array akses paralel, semua disk anggota berpartisipasi dalam pelaksanaan setiap permintaan I/O. Biasanya, poros dari masing-masing drive disinkronkan sehingga setiap *head* disk berada pada posisi yang sama pada setiap disk pada waktu tertentu.

Seperti dalam skema RAID lainnya, striping data digunakan. Dalam kasus RAID 2 dan 3, strip sangat kecil, seringkali sekecil byte atau word tunggal. Dengan RAID 2, kode koreksi kesalahan dihitung pada bit yang sesuai pada setiap disk data, dan bit kode disimpan dalam posisi bit yang sesuai pada beberapa disk paritas. Biasanya, kode Hamming digunakan, yang mampu memperbaiki kesalahan bit tunggal dan mendeteksi kesalahan bit ganda.

Meskipun RAID 2 membutuhkan lebih sedikit disk daripada RAID 1, itu masih agak mahal. Jumlah disk redundan sebanding dengan log dari jumlah disk data. Pada sekali baca, semua disk diakses secara bersamaan. Data yang diminta dan kode koreksi kesalahan terkait dikirimkan ke pengontrol array. Jika ada kesalahan bit tunggal, pengontrol dapat mengenali dan memperbaiki kesalahan secara instan, sehingga waktu akses baca tidak melambat. Pada satu penulisan, semua disk data dan disk paritas harus diakses untuk operasi penulisan.

RAID 2 hanya akan menjadi pilihan yang efektif di lingkungan di mana banyak kesalahan disk terjadi. Mengingat keandalan yang tinggi dari masing-masing disk dan drive disk, RAID 2 berlebihan dan tidak diimplementasikan.

Tingkat RAID 3

RAID 3 diatur dengan cara yang mirip dengan RAID 2. Perbedaannya adalah bahwa RAID 3 hanya membutuhkan disk redundan tunggal, tidak peduli seberapa besar array disk. RAID 3 menggunakan akses paralel, dengan data didistribusikan dalam strip kecil. Alih-alih kode koreksi kesalahan, bit paritas sederhana dihitung untuk set bit individual di posisi yang sama pada semua disk data.

Redundansi

Dalam hal terjadi kegagalan drive, drive paritas diakses dan data direkonstruksi dari perangkat yang tersisa. Setelah drive yang gagal diganti, data yang hilang dapat dipulihkan pada drive baru dan operasi dilanjutkan.

Rekonstruksi data sederhana. Pertimbangkan array lima drive di mana X0 hingga X3 berisi data dan X4 adalah disk paritas. Paritas untuk bit engan dihitung sebagai berikut:

$$X4(i) = X3(i) \oplus X2(i) \oplus X1(i) \oplus X0(i)$$

di mana \oplus adalah fungsi eksklusif-OR.

Misalkan drive X1 gagal. Jika menambahkan $X4(i) \oplus X1(i)$ ke kedua sisi persamaan sebelumnya, maka didapat:

$$X1(i) = X4(i) \oplus X3(i) \oplus X2(i) \oplus X0(i)$$

Dengan demikian, isi setiap strip data pada X1 dapat dibuat ulang dari isi strip yang sesuai pada disk yang tersisa dalam array. Prinsip ini berlaku untuk level RAID 3 hingga 6.

Jika terjadi kegagalan disk, semua data masih tersedia dalam apa yang disebut sebagai mode tereduksi. Dalam mode ini, untuk dibaca, data yang hilang dibuat kembali dengan cepat menggunakan perhitungan eksklusif-OR. Ketika data ditulis ke array RAID 3 yang dikurangi, konsistensi paritas harus dipertahankan untuk regenerasi nanti. Kembali ke operasi penuh mengharuskan disk yang gagal diganti dan seluruh konten disk yang gagal diregenerasi pada disk yang baru.

Kinerja (Performance)

Karena data bergris kecil, RAID 3 dapat mencapai kecepatan transfer data yang sangat tinggi. Setiap permintaan I/O akan melibatkan transfer data paralel dari semua disk data. Untuk transfer besar, peningkatan kinerja terutama terlihat. Di sisi lain, hanya satu permintaan I/O yang dapat dieksekusi sekaligus. Jadi, dalam lingkungan yang berorientasi transaksi, kinerja menderita.

Level RAID 4

Level RAID 4 hingga 6 memanfaatkan teknik akses independen. Dalam array akses independen, setiap disk anggota beroperasi secara independen, sehingga permintaan I/O yang terpisah dapat dipenuhi secara paralel. Karena itu, array akses independen lebih cocok untuk aplikasi yang memerlukan tingkat permintaan I/O yang tinggi dan relatif kurang cocok untuk aplikasi yang memerlukan tingkat transfer data yang tinggi.

Seperti dalam skema RAID lainnya, striping data digunakan. Dalam kasus RAID 4 hingga 6, strip relatif besar. Dengan RAID 4, strip paritas bit demi bit dihitung pada strip yang sesuai pada setiap disk data, dan bit paritas disimpan dalam strip yang sesuai pada disk paritas.

RAID 4 melibatkan penalti tulis ketika permintaan penulisan I/O ukuran kecil dilakukan. Setiap kali penulisan terjadi, perangkat lunak manajemen array harus memperbarui tidak hanya data pengguna tetapi juga bit paritas yang sesuai. Pertimbangkan array lima drive di mana X0 hingga X3 berisi data dan X4 adalah disk paritas. Misalkan penulisan dilakukan yang hanya melibatkan strip pada disk X1. Awalnya, untuk setiap bit i, memiliki hubungan berikut:

$$X4(i) = X3(i) \oplus X2(i) \oplus X1(i) \oplus X0(i) \quad (6.2)$$

Setelah pembaruan, dengan bit yang berpotensi diubah ditunjukkan oleh simbol utama:

$$\begin{aligned} X4'(i) &= X3(i) \oplus X2(i) \oplus X1'(i) \oplus X0(i) \\ &= X3(i) \oplus X2(i) \oplus X1'(i) \oplus X0(i) \oplus X1(i) \oplus X1(i) \\ &= X3(i) \oplus X2(i) \oplus X1(i) \oplus X0(i) \oplus X1(i) \oplus X1'(i) \\ &= X4(i) \oplus X1(i) \oplus X1'(i) \end{aligned}$$

Rangkaian persamaan sebelumnya diturunkan sebagai berikut. Baris pertama menunjukkan bahwa perubahan X1 juga akan mempengaruhi X4 paritas disk. Di baris kedua, ditambahkan istilah $\oplus X1(i)$ $\oplus X1'(i)$. Karena eksklusif-OR dari kuantitas apa pun dengan dirinya sendiri adalah 0, ini tidak mempengaruhi persamaan. Namun, itu adalah kenyamanan yang digunakan untuk membuat baris ketiga, dengan pemesanan ulang. Akhirnya, Persamaan (6.2) digunakan untuk mengganti empat istilah pertama dengan X4(i).

Untuk menghitung paritas baru, perangkat lunak manajemen array harus membaca strip pengguna lama dan strip paritas lama. Kemudian dapat memperbarui dua strip ini dengan data baru dan paritas yang baru dihitung. Jadi, setiap strip menulis melibatkan dua membaca dan dua menulis.

Dalam kasus ukuran I/O yang lebih besar yang melibatkan strip pada semua disk drive, paritas mudah dihitung dengan perhitungan hanya menggunakan bit data baru. Dengan demikian, drive paritas dapat diperbarui secara paralel dengan drive data dan tidak ada tambahan baca atau tulis. Dalam setiap kasus, setiap operasi penulisan harus melibatkan disk paritas, yang karenanya dapat menjadi hambatan.

Level RAID 5

RAID 5 diatur dengan cara yang mirip dengan RAID 4. Perbedaannya adalah bahwa RAID 5 mendistribusikan strip paritas di semua disk. Alokasi tipikal adalah skema round-robin, seperti yang diilustrasikan dalam Gambar 3.6f. Untuk array n-disk, strip paritas berada pada disk yang berbeda untuk n strip pertama, dan pola kemudian berulang. Distribusi strip paritas di semua drive menghindari potensi I/O botol-leher yang ditemukan di RAID 4.

Level RAID 6

RAID 6 diperkenalkan dalam makalah selanjutnya oleh para peneliti Berkeley. Dalam skema RAID 6, dua perhitungan paritas yang berbeda dilakukan dan disimpan dalam blok terpisah pada disk yang berbeda. Dengan demikian, array RAID 6 yang data penggunanya memerlukan N disk terdiri dari N + 2 disk.

Gambar 3.6g menggambarkan skema tersebut. P dan Q adalah dua algoritma pemeriksaan data yang berbeda. Salah satu dari keduanya adalah perhitungan eksklusif-OR yang digunakan dalam RAID 4 dan 5. Tetapi yang lainnya adalah algoritma pemeriksaan data independen. Ini memungkinkan untuk meregenerasi data bahkan jika dua disk yang berisi data pengguna gagal.

Keuntungan dari RAID 6 adalah ia menyediakan ketersediaan data yang sangat tinggi. Tiga disk harus gagal dalam interval MTTR (berarti waktu untuk memperbaiki) untuk menyebabkan data hilang. Di sisi lain, RAID 6 menimbulkan penalti tulis yang besar, karena setiap penulisan memengaruhi dua blok paritas. Tabel 3.3 adalah ringkasan perbandingan dari tujuh level RAID.

Tabel 3.3. Kelebihan dan Kekurangan RAID 0 sampai 6

Level	Kelebihan	Kekurangan	Penerapan
0	Performa I/O sangat tinggi dengan mendistribusikan beban I/O ke banyak saluran dan drive. Tidak melibatkan kalkulasi paritas yang <i>overhead</i> . Desain sangat sederhana. Mudah diimplementasikan	Kegagalan hanya satu drive akan mengakibatkan semua data dalam array hilang	Produksi dan pengeditan video Pengeditan gambar Pre-press applications Aplikasi apa pun yang membutuhkan bandwidth tinggi
1	100% redundansi data, berarti tidak perlu membangun kembali jika terjadi kegagalan disk, cukup salin ke disk pengganti Dalam keadaan tertentu, RAID 1 dapat mengalami beberapa kegagalan drive secara bersamaan Merupakan desain subsistem RAID storage yang paling sederhana	<i>Overhead</i> disk tertinggi dari semua jenis RAID (100%) — tidak efisien	Accounting Payroll Financial Aplikasi apa pun yang membutuhkan ketersediaan yang sangat tinggi
2	Kecepatan transfer data yang sangat tinggi mungkin Semakin tinggi kecepatan transfer data yang diperlukan, semakin baik rasio disk data ke disk ECC Desain pengontrol yang relatif sederhana dibandingkan dengan RAID level 3, 4, & 5	Rasio yang sangat tinggi dari disk ECC terhadap disk data dengan ukuran <i>word</i> yang lebih kecil — tidak efisien Biaya tingkat awal sangat tinggi — membutuhkan persyaratan kecepatan transfer yang sangat tinggi untuk membenarkan (<i>justify</i>)	Tidak ada implementasi komersial, tidak layak secara komersial
3	Kecepatan transfer data baca sangat tinggi Kecepatan transfer tulis data sangat tinggi	Laju transaksi paling baik sama dengan satu drive disk (jika spindel disinkronkan) Desain pengontrol cukup kompleks	Produksi video dan streaming langsung Pengeditan gambar Penyuntingan video Prepress applications

	Kegagalan disk memiliki dampak yang tidak signifikan pada throughput Rasio rendah dari disk ECC (paritas) ke disk data berarti efisiensi tinggi		Aplikasi apa pun yang membutuhkan throughput tinggi
4	Tingkat transaksi data Baca sangat tinggi Rasio rendah dari disk ECC (paritas) ke disk data berarti efisiensi tinggi	Desain pengontrol yang cukup kompleks Nilai transaksi tulis terburuk dan kecepatan transfer agregat tulis sulit dan tidak efisien dalam <i>rebuild data</i> jika terjadi kegagalan disk	Tidak ada implementasi komersial/tidak layak secara komersial
5	Tingkat transaksi Baca data tertinggi Rasio rendah dari disk ECC (paritas) ke disk data, berarti efisiensi tinggi Kecepatan transfer agregat yang baik	Desain pengontrol paling kompleks Sulit untuk membangun kembali jika terjadi kegagalan disk (dibandingkan dengan RAID level 1)	File dan server aplikasi Server database Web, email, dan news Servers. Server intranet Level RAID paling serbaguna
6	Memberikan toleransi kesalahan data yang sangat tinggi dan dapat mempertahankan beberapa kegagalan drive secara bersamaan	Desain pengontrol yang lebih kompleks <i>Overhead</i> pengontrol untuk menghitung alamat paritas sangat tinggi	Solusi sempurna untuk aplikasi penting

Sumber: (Stallings, 2016)

3.3. Solid State Drives (SSD)

Salah satu perkembangan paling signifikan dalam arsitektur komputer dalam beberapa tahun terakhir adalah meningkatnya penggunaan *Solid State Drive* (SSD) untuk melengkapi atau bahkan mengganti *Hard Disk Drive* (HDD), baik sebagai memori sekunder internal maupun eksternal. Istilah *solid state* mengacu pada sirkuit elektronik yang dibangun dengan semikonduktor. SSD adalah perangkat memori yang dibuat dengan komponen solid state yang dapat digunakan sebagai pengganti drive hard disk.

SSD Dibandingkan dengan HDD

Karena biaya SSD berbasis flash telah turun dan kinerja serta kepadatan bit meningkat, SSD menjadi semakin kompetitif dengan HDD. Tabel 3.4 menunjukkan ukuran perbandingan pada saat penulisan ini. SSD memiliki keunggulan berikut dibandingkan HDD:

- performance *input/output operations per second* (IOPS) yang tinggi: Secara signifikan meningkatkan subsistem I/O kinerja.
- Daya tahan (*Durability*): Tidak terlalu rentan terhadap guncangan dan getaran fisik.
- Masa pakai lebih lama (*longer lifespan*): SSD tidak rentan terhadap keausan mekanis.
- Konsumsi daya yang lebih rendah: SSD menggunakan daya yang jauh lebih kecil daripada HDD ukuran sebanding.
- Kemampuan berjalan yang lebih tenang dan lebih dingin: Lebih sedikit ruang yang dibutuhkan, biaya energi yang lebih rendah, dan perusahaan yang lebih hijau.
- Waktu akses dan tingkat latensi yang lebih rendah: Lebih dari 10 kali lebih cepat daripada disk pemintalan dalam HDD.

Selain antarmuka ke sistem host, SSD berisi komponen-komponen berikut:

- Pengontrol: Menyediakan antarmuka tingkat perangkat SSD dan eksekusi firmware.
- Mengatasi: Logika yang menjalankan fungsi pemilihan di seluruh komponen memori flash.

- Buffer/*cache* data: Komponen memori RAM kecepatan tinggi yang digunakan untuk pencocokan kecepatan dan untuk meningkatkan throughput data.
- Koreksi kesalahan: Logika untuk deteksi dan koreksi kesalahan.

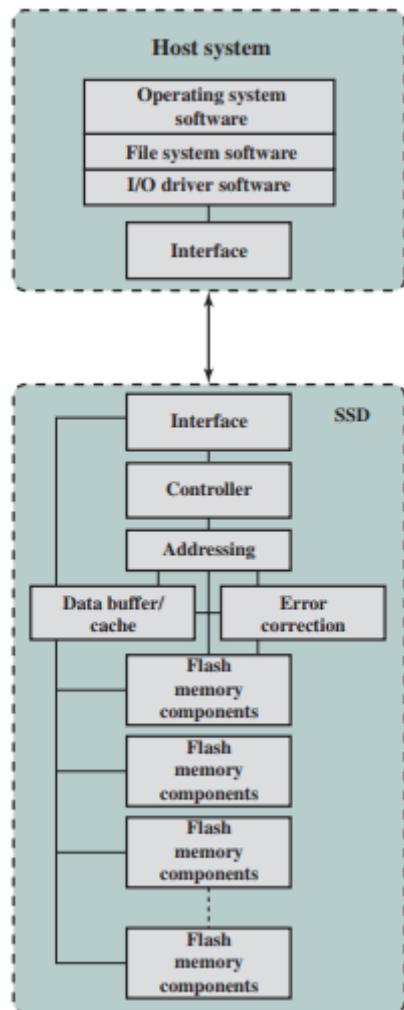
Komponen memori flash: Masing-masing *chip* NAND flash.

Tabel 3.4. Perbandingan SSD dan Disk Drive

	NAND Flash Drives	Seagate Laptop Internal HDD
File copy/write speed	200-500 Mbps	50-120 Mbps
Pengambilan daya/masa pakai baterai	Pengeluaran daya yang lebih sedikit, rata-rata 2–3 watt, menghasilkan peningkatan daya baterai selama lebih dari 30 menit	Lebih banyak menghabiskan daya, rata-rata 6–7 watt dan karena itu menggunakan lebih banyak baterai
Kapasitas penyimpanan	Biasanya tidak lebih dari 512 GB untuk drive ukuran notebook; Maksimal 1 TB untuk desktop	Biasanya sekitar 500 GB dan maks 2 TB untuk drive ukuran notebook; Maksimal 4 TB untuk desktop
Cost	Perkiraan \$ 0,50 per GB untuk drive 1 TB	Perkiraan \$ 0,15 per GB untuk drive 4 TB

Sumber: (Stallings, 2016)

Gambar 3.8 mengilustrasikan pandangan umum komponen sistem arsitektur umum yang terkait dengan sistem SSD apa pun. Pada sistem host, sistem operasi memanggil perangkat lunak sistem file untuk mengakses data pada disk. Sistem file, pada gilirannya, memanggil perangkat lunak driver I/O. Perangkat lunak driver I/O menyediakan akses host ke produk SSD tertentu. Komponen antarmuka pada Gambar 3.8 mengacu pada antarmuka fisik dan listrik antara prosesor host dan perangkat periferal SSD. Jika perangkat adalah hard drive internal, antarmuka yang umum adalah PCIe. Untuk perangkat eksternal, satu antarmuka umum adalah USB.



Sumber: (Stallings, 2016)
Gambar 3.8. Arsitektur *Solid State Drive*

4. Input/Output

Selain prosesor dan satu set modul memori, elemen kunci ketiga dari sistem komputer adalah satu set modul I/O. Setiap modul terhubung ke bus sistem atau saklar pusat dan mengontrol satu atau beberapa perangkat periferal. Modul I/O bukan hanya satu set konektor mekanis yang menghubungkan perangkat ke bus sistem. Sebaliknya, modul I/O berisi logika untuk melakukan fungsi komunikasi antara periferal dan bus system.

Alasan mengapa tidak dapat menghubungkan periferal langsung ke bus sistem adalah sebagai berikut:

- 1) Ada beragam periferal dengan berbagai metode operasi. Akan tidak praktis untuk memasukkan logika yang diperlukan dalam prosesor untuk mengontrol berbagai perangkat.
- 2) Kecepatan transfer data periferal seringkali jauh lebih lambat dari pada memori atau prosesor. Jadi, tidak praktis menggunakan bus sistem berkecepatan tinggi untuk berkomunikasi langsung dengan periferal.
- 3) Di sisi lain, kecepatan transfer data beberapa periferal lebih cepat daripada memori atau prosesor. Sekali lagi, ketidakcocokan akan menyebabkan inefisiensi jika tidak dikelola dengan baik.
- 4) Perangkat sering menggunakan format data dan panjang *word* yang berbeda dari komputer yang dilampirkan.
- 5) Dengan demikian, modul I/O diperlukan. Modul ini memiliki dua fungsi utama, yaitu: antarmuka ke prosesor dan memori melalui bus sistem atau saklar pusat, dan antarmuka ke satu atau beberapa perangkat periferal oleh tautan data yang disesuaikan.

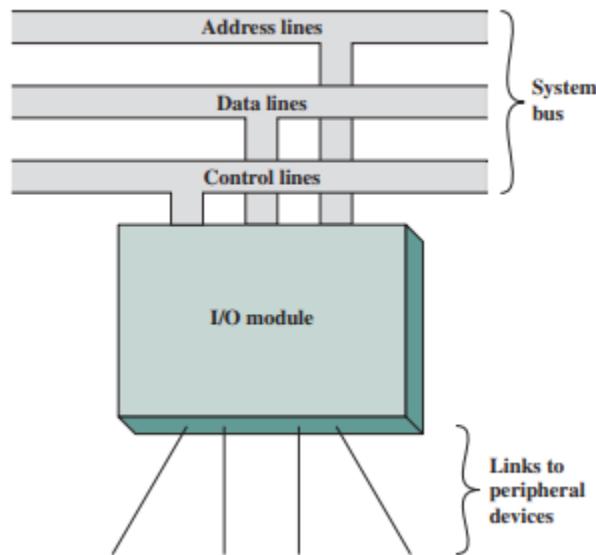
4.1. External Dievices

Operasi I/O dicapai melalui bermacam-macam perangkat eksternal (*external devices*) yang menyediakan sarana pertukaran data antara lingkungan eksternal dan komputer. Perangkat eksternal terhubung ke komputer melalui modul I/O (Gambar 4.1). Tautan ini digunakan untuk bertukar kontrol, status, dan data antara modul I/O dan perangkat eksternal. Perangkat eksternal yang terhubung ke modul I/O sering disebut sebagai perangkat periferal atau, sederhana, periferal.

Secara luas perangkat eksternal dapat diklasifikasikan dalam tiga kategori:

- 1) *Human readable*: Cocok untuk berkomunikasi dengan user;
- 2) *Mechine readable*: Cocok untuk berkomunikasi dengan peralatan;
- 3) *Communication*: Cocok untuk berkomunikasi dengan perangkat jarak jauh.

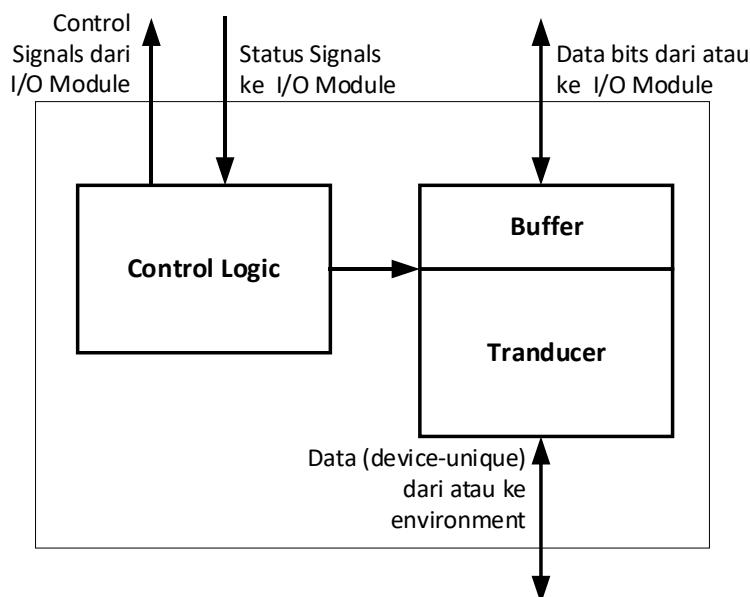
Contoh perangkat *human readable* adalah *Video Dispaly Terminal* (VDT) dan printer. Contoh perangkat *mechine readable* adalah disk magnetik dan sistem pita, serta sensor dan aktuator, seperti yang digunakan dalam aplikasi robotika. Dari sudut pandang struktural, perangkat-perangkat tersebut dikendalikan oleh modul I/O.



Sumber: (Stallings, 2016)
Gambar 4.1. Model Generik dari Modul I/O

Perangkat komunikasi (*Communication devices*) memungkinkan komputer untuk bertukar data dengan perangkat jarak jauh, dapat berupa *human readable*, seperti terminal, perangkat yang dapat digerakkan dengan mesin, atau bahkan komputer lain.

Dalam istilah yang sangat umum, sifat perangkat eksternal ditunjukkan pada Gambar 4.2. Antarmuka ke modul I/O adalah dalam bentuk sinyal kontrol, data, dan status. Sinyal kontrol menentukan fungsi yang akan dilakukan perangkat, seperti mengirim data ke modul I/O (*INPUT* atau *READ*), menerima data dari modul I/O (*OUTPUT* atau *WRITE*), melaporkan status, atau melakukan beberapa fungsi kontrol tertentu ke perangkat (misalnya: posisi *head disk*). Data dalam bentuk sekumpulan bit untuk dikirim atau diterima dari modul I/O. Sinyal status menunjukkan keadaan perangkat. Contohnya adalah *READY/NOT READY* untuk menunjukkan apakah perangkat siap untuk transfer data.



Sumber: (Stallings, 2016)
Gambar 4.2. Diagram Blok External Device

Logika kontrol yang terkait dengan perangkat mengontrol operasi perangkat sebagai respons terhadap arah dari modul I/O. Transduser mengubah data dari listrik ke bentuk energi lain selama *output* dan dari bentuk lain menjadi listrik selama *input*. Biasanya, buffer dikaitkan dengan transduser untuk sementara waktu menahan data yang ditransfer antara modul I/O dan lingkungan eksternal. Ukuran buffer 8 hingga 16 bit adalah umum untuk perangkat serial, sedangkan perangkat berorientasi blok seperti pengontrol drive disk mungkin memiliki buffer yang jauh lebih besar.

A. Keyboard dan Monitor

Cara paling umum dari interaksi komputer/pengguna adalah pengaturan keyboard/monitor. Pengguna memberikan *input* melalui keyboard, *input* tersebut kemudian dikirim ke komputer dan juga dapat ditampilkan pada monitor. Selain itu, monitor menampilkan data yang disediakan oleh komputer.

Unit dasar pertukaran adalah karakter, dimana setiap karakter mewakili kode, biasanya 7 atau 8 bit panjangnya. Kode teks yang paling umum digunakan adalah *International Reference Alphabet* (IRA). Setiap karakter dalam kode ini diwakili oleh kode biner 7-bit yang unik; dengan demikian, 128 karakter yang berbeda dapat direpresentasikan. Karakter terdiri dari dua jenis: cetak dan kontrol. Karakter yang dapat dicetak adalah karakter alfabet, numerik, dan khusus yang dapat dicetak di atas kertas atau ditampilkan di layar. Beberapa karakter kontrol berhubungan dengan mengontrol pencetakan atau tampilan karakter; contohnya adalah *carriage return*. Karakter kontrol lainnya berkaitan dengan prosedur komunikasi (lebih lanjut akan dibahas pada subbab 8.2.B).

Untuk *input* keyboard, ketika pengguna menekan tombol, ini menghasilkan sinyal elektronik yang ditafsirkan oleh transduser di keyboard dan diterjemahkan ke dalam pola bit kode IRA yang sesuai. Pola bit ini kemudian ditransmisikan ke modul I/O di komputer. Di komputer, teks dapat disimpan dalam kode IRA yang sama. Pada keluaran, karakter kode IRA ditransmisikan ke perangkat eksternal dari modul I/O. Transduser pada perangkat menginterpretasikan kode ini dan mengirimkan sinyal elektronik yang diperlukan ke perangkat *output* untuk menampilkan karakter yang ditunjukkan atau melakukan fungsi kontrol yang diminta.

Cara paling umum dari interaksi komputer/pengguna adalah pengaturan keyboard/monitor. Pengguna memberikan *input* melalui keyboard, *input* tersebut kemudian dikirim ke komputer dan juga dapat ditampilkan pada monitor. Selain itu, monitor menampilkan data yang disediakan oleh komputer. Unit dasar pertukaran adalah karakter. Terkait dengan setiap karakter adalah kode, biasanya 7 atau 8 bit panjangnya. Kode teks yang paling umum digunakan adalah International Reference Alphabet (IRA). Setiap karakter dalam kode ini diwakili oleh kode biner 7-bit yang unik; dengan demikian, 128 karakter yang berbeda dapat direpresentasikan. Karakter terdiri dari dua jenis: cetak dan kontrol. Karakter yang dapat dicetak adalah karakter alfabet, numerik, dan khusus yang dapat dicetak di atas kertas atau ditampilkan di layar. Beberapa karakter kontrol berhubungan dengan mengontrol pencetakan atau tampilan karakter; contohnya adalah carriage return. Karakter kontrol lainnya berkaitan dengan prosedur komunikasi.

Untuk *input* keyboard, ketika pengguna menekan tombol, ini menghasilkan sinyal elektronik yang ditafsirkan oleh transduser di keyboard dan diterjemahkan ke dalam pola bit kode IRA yang sesuai. Pola bit ini kemudian ditransmisikan ke modul I/O di komputer. Di komputer, teks dapat disimpan dalam kode IRA yang sama. Pada keluaran, karakter kode IRA ditransmisikan ke perangkat eksternal dari modul I/O. Transduser pada perangkat menginterpretasikan kode ini dan mengirimkan sinyal elektronik yang diperlukan ke perangkat *output* untuk menampilkan karakter yang ditunjukkan atau melakukan fungsi kontrol yang diminta.

B. Disk Drive

Disk drive berisi komponen elektronik yang membentuk fungsi untuk bertukar data, kontrol, dan sinyal status dengan modul I/O, ditambah komponen elektronik untuk mengendalikan mekanisme baca/tulis disk. Dalam disk dengan *fixed-head*, transduser mampu mengonversi antara pola magnetik pada permukaan disk yang bergerak dan bit dalam buffer perangkat (Gambar 4.2). Pada disk drive dengan *Movable head*, juga harus dapat menyebabkan lengan disk bergerak secara radial masuk dan keluar melintasi permukaan disk.

4.2. Modul I/O (I/O Modules)

Fungsi atau persyaratan utama untuk modul I/O adalah sebagai berikut:

- 1) Kontrol dan pengaturan waktu
- 2) Komunikasi prosesor
- 3) Komunikasi perangkat
- 4) Penyangga data (*data buffering*)
- 5) Deteksi kesalahan (*error detection*)

Selama periode waktu tertentu, prosesor dapat berkomunikasi dengan satu atau beberapa perangkat eksternal dalam pola yang tidak dapat diprediksi, tergantung pada kebutuhan program untuk I/O. Sumber daya internal, seperti memori utama dan bus sistem, penggunaannya harus dibagi untuk sejumlah kegiatan, termasuk I/O. Dengan demikian, fungsi I/O mencakup persyaratan kontrol dan waktu, untuk mengoordinasikan aliran lalu lintas data internal.

sumber daya dan perangkat eksternal. Misalnya, kontrol transfer data dari perangkat eksternal ke prosesor mungkin melibatkan urutan langkah-langkah berikut:

- 1) Prosesor menginterogasi modul I/O untuk memeriksa status perangkat yang terpasang.
- 2) Modul I/O mengembalikan status perangkat.
- 3) Jika perangkat operasional dan siap untuk mengirim, prosesor meminta transfer data, melalui perintah ke modul I/O.
- 4) Modul I/O memperoleh unit data (pada umumnya 8 bit atau kelipatannya) dari perangkat eksternal.
- 5) Data ditransfer dari modul I/O ke prosesor.

Jika sistem menggunakan bus, maka masing-masing interaksi antara prosesor dan modul I/O melibatkan satu atau lebih arbitrasi bus.

Skenario yang disederhanakan sebelumnya juga menggambarkan bahwa modul I/O harus berkomunikasi dengan prosesor dan dengan perangkat eksternal. Komunikasi prosesor melibatkan hal-hal berikut:

- 1) Decoding perintah: Modul I/O menerima perintah dari prosesor, biasanya dikirim sebagai sinyal pada bus kontrol. Misalnya, modul I/O untuk drive disk mungkin menerima perintah berikut: READ SECTOR, WRITE SECTOR, SEEK *track* number, dan SCAN record ID. Dua perintah terakhir masing-masing menyertakan parameter yang dikirim pada bus data.
- 2) Data: Data dipertukarkan antara prosesor dan modul I/O melalui bus data.
- 3) Pelaporan status: Karena periferal sangat lambat, penting untuk mengetahui status modul I/O. Misalnya, jika modul I/O diminta untuk mengirim data ke prosesor (baca), itu mungkin tidak siap untuk melakukannya karena masih bekerja pada perintah I/O sebelumnya. Fakta ini dapat dilaporkan dengan sinyal status. Sinyal status umum adalah SIBUK dan SIAP. Mungkin juga ada sinyal untuk melaporkan berbagai kondisi kesalahan.
- 4) Pengenalan alamat: Sama seperti setiap *word* dari memori memiliki alamat, demikian juga setiap perangkat I/O. Dengan demikian, modul I/O harus mengenali satu alamat unik untuk setiap perangkat yang dikontrolnya.

Di sisi lain, modul I/O harus dapat melakukan komunikasi perangkat. Komunikasi ini melibatkan perintah, informasi status, dan data (Gambar 4.3). Tugas penting dari modul I/O adalah buffering data. Buffering perlu dilakukan oleh modul I/O agar dapat bekerja dalam dua kecepatan transfer data, yaitu kecepatan transfer data prosesor dan memori yang sangat tinggi, dan kecepatan perangkat periferal yang sangat lamban. Data disangga dalam modul I/O dan kemudian dikirim ke perangkat periferal pada kecepatan periferal tersebut. Dalam arah yang berlawanan, data di-buffer agar tidak menyebabkan memori berada dalam operasi transfer yang lambat. Dengan demikian, modul I/O harus dapat beroperasi pada kecepatan perangkat maupun memori. Demikian pula, jika perangkat I/O beroperasi pada kecepatan yang lebih tinggi dari kecepatan akses memori, maka modul I/O melakukan operasi buffering yang diperlukan.

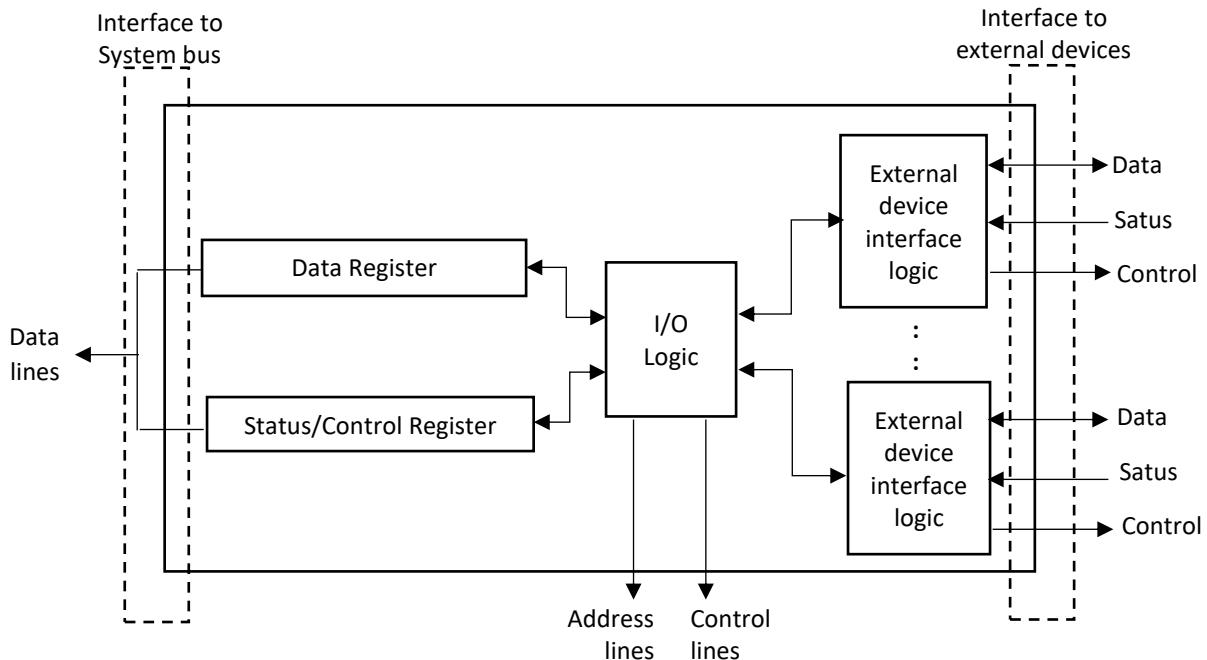
Akhirnya, modul I/O sering bertanggung jawab untuk deteksi kesalahan dan untuk selanjutnya melaporkan kesalahan ke prosesor. Satu kelas kesalahan mencakup kerusakan mekanis dan listrik yang dilaporkan oleh perangkat (misalnya: Kertas macet, *track disk buruk*). Kelas lain terdiri dari perubahan yang tidak disengaja ke pola bit karena ditransmisikan dari perangkat ke modul I/O. Beberapa bentuk kode pendekripsi kesalahan sering digunakan untuk mendekripsi kesalahan transmisi. Contoh sederhana adalah penggunaan bit paritas pada setiap karakter data. Misalnya, kode karakter IRA menempati 7 bit *byte*. Bit kedelapan diatur sehingga jumlah total 1s dalam *byte* adalah even (*even parity*) atau odd (*odd parity*). Ketika *byte* diterima, modul I/O memeriksa paritas untuk menentukan apakah kesalahan telah terjadi.

Struktur Modul I/O

Modul I/O sangat bervariasi dalam kompleksitas dan jumlah perangkat eksternal yang dikontrolnya. Gambar 4.3 menyediakan diagram blok umum dari modul I/O. Modul terhubung ke seluruh komputer melalui serangkaian jalur sinyal (misalnya alur bus sistem). Data yang ditransfer ke dan dari modul disangga dalam satu atau lebih register data. Mungkin juga ada satu atau lebih register status yang memberikan informasi status saat ini. Register status juga dapat berfungsi sebagai register kontrol, untuk menerima informasi kontrol terperinci dari prosesor. Logika dalam modul berinteraksi dengan prosesor melalui serangkaian garis kontrol. Prosesor menggunakan garis kontrol untuk mengeluarkan perintah ke modul I/O. Beberapa jalur kontrol dapat digunakan oleh modul I/O (misalnya untuk sinyal arbitrase dan status). Modul ini juga harus dapat mengenali dan menghasilkan alamat yang terkait dengan perangkat yang dikontrolnya. Setiap modul I/O memiliki alamat unik atau, jika mengontrol lebih dari satu perangkat eksternal, satu set alamat unik. Akhirnya, modul I/O berisi logika khusus untuk antarmuka dengan setiap perangkat yang dikontrolnya.

Modul I/O berfungsi untuk memungkinkan prosesor melihat berbagai perangkat dengan cara yang sederhana. Ada spektrum kemampuan yang dapat disediakan. Modul I/O dapat menyembunyikan detail pengaturan waktu, format, dan elektromekanik dari perangkat eksternal sehingga prosesor dapat berfungsi dalam hal perintah membaca dan menulis sederhana, dan mungkin membuka dan menutup perintah file. Dalam bentuknya yang paling sederhana, modul I/O mungkin masih menyisakan sebagian besar pekerjaan untuk mengendalikan perangkat yang terlihat oleh prosesor.

Modul I/O yang menanggung sebagian besar beban pemrosesan terperinci, menghadirkan antarmuka tingkat tinggi ke prosesor, biasanya disebut sebagai saluran I/O atau prosesor I/O. Modul I/O yang cukup primitif dan membutuhkan kontrol terperinci biasanya disebut sebagai pengontrol I/O atau pengontrol perangkat. Pengontrol I/O umumnya terlihat pada mikrokomputer, sedangkan saluran I/O digunakan pada mainframe.

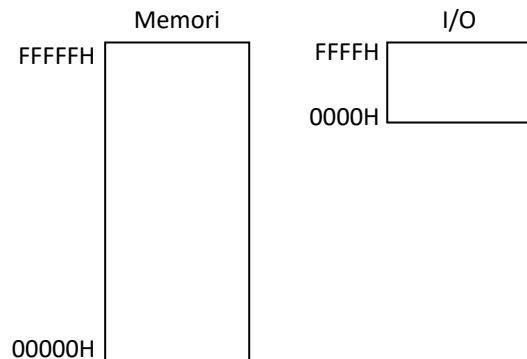


Sumber: (Stallings, 2016)
Gambar 4.3. Blok Diagram Modul I/O

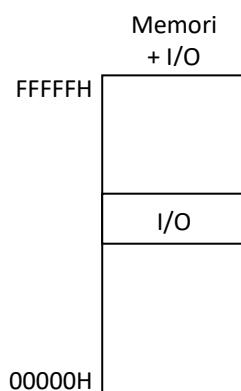
4.3. I/O Terisolasi dan I/O Terpeta-Memori

I/O terisolasi (isolated I/O) menggambarkan bagaimana sistem memisahkan ruang alamat memori dengan ruang alamat I/O. Gambar 4.4a mengilustrasikan ruang alamat I/O terisolasi pada rancangan prosesor keluarga Intel 80x86. Alamat yang terpisah ini disebut dengan port, terpisah dari memori. Hal ini akan memberikan keuntungan bagi user, dimana user dapat menggunakan ruang alamat memori secara maksimal tanpa harus mengurangi sebagianya untuk dipetakan kepada perangkat I/O. Kekurangan I/O terisolasi adalah transfer data antara modul I/O dengan memori harus menggunakan instruksi berbeda dari yang biasa digunakan untuk memori. Selain itu sistem harus menyediakan sinyal control terpisah bagi ruang I/O. Pada lingkungan personal computer (PC), alamat port 8 bit digunakan untuk mengakses perangkat (peripheral) yang berada pada papan sistem (motherboard), seperti timer dan antarmuka keyboard. Sementara port 16 bit digunakan port seri dan parallel, sistem video, serta disk drive.

I/O terpeta memory (memory-mapped I/O), ruang alamat untuk perangkat luar, diambil dari sebagian ruang alamat memori sistem. Keunggulan utama dari I/O terpeta memori adalah setiap instruksi transfer memori dapat digunakan untuk mengakses perangkat I/O. Kekurangannya adalah Sebagian ruang alamat memori sistem tidak dapat digunakan untuk mengakses memori, karena harus digunakan untuk memetakan ruang alamat perangkat I/O. Hal ini akan mengurangi kapasitas memori yang dapat digunakan oleh aplikasi. Keunggulan lainnya, I/O terpeta memori tidak membutuhkan sinyal control tambahan untuk mengakses perangkat I/O, sehingga dapat mengurangi jumlah sirkuit yang diperlukan untuk melakukan decoding.



(a) I/O Terisolasi



(b) I/O Terpeta Memori

Sumber: (Brey, 2009)

Gambar 4.4. Peta I/O dan Memori Pada Prosesor Keluarga Intel 80x86

4.4. Teknik Pelayanan I/O

Tiga teknik dimungkinkan untuk operasi I/O, yaitu: Programmed I/O, Interrupt driven I/O, dan Direct Memory I/O. Dengan I/O terprogram, data dipertukarkan antara prosesor dan modul I/O. Prosesor menjalankan program yang memberinya kendali langsung terhadap operasi I/O, termasuk membaca status perangkat, mengirim perintah baca atau tulis, dan mentransfer data. Ketika prosesor mengeluarkan perintah ke modul I/O, itu harus menunggu sampai operasi I/O selesai. Jika prosesor lebih cepat dari modul I/O, ini akan membuang waktu prosesor. Dengan I/O yang digerakkan oleh interupsi, prosesor mengeluarkan perintah I/O, terus menjalankan instruksi lain. Dengan I/O terprogram dan interupsi, prosesor bertanggung jawab untuk mengekstraksi data dari memori utama untuk *output* dan menyimpan data dalam memori utama untuk *input*. Alternatif lain dikenal sebagai *Direct Memory Access* (DMA). Dalam mode ini, pertukaran memori antara modul I/O dan memori utama dilakukan tanpa keterlibatan prosesor. Tabel 4.1 menunjukkan hubungan di antara ketiga teknik ini.

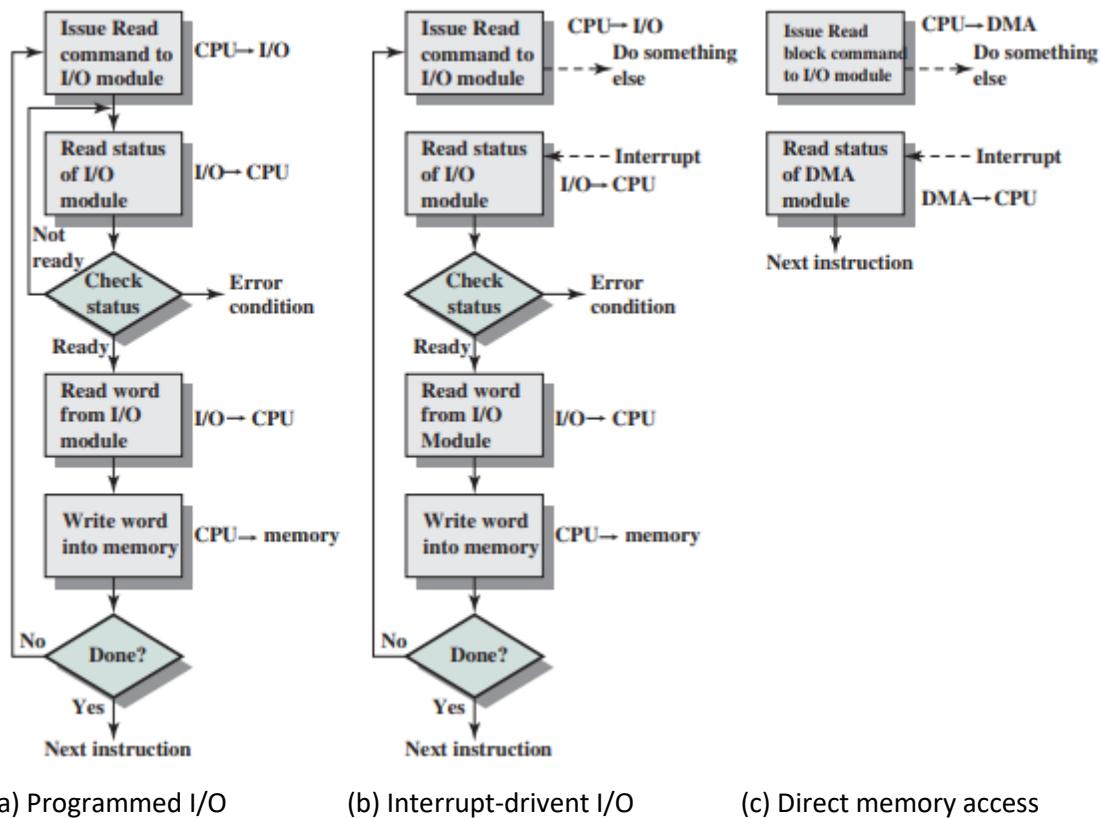
Tabel 4.1. Teknik Layanan I/O

	Tanpa <i>interrupt</i>	Dengan <i>interrupt</i>
Transfer I/O ke memori melalui prosesor	<i>Programmed I/O</i>	<i>Interrupt driven I/O</i>
Transfer I/O ke memori langsung	-	DMA

Sumber: (Brey, 2009)

A. Programmed I/O

Ketika prosesor menjalankan program dan menemukan instruksi yang berkaitan dengan I/O, maka prosesor akan menjalankan instruksi tersebut dengan mengeluarkan perintah ke modul I/O yang sesuai. Dengan metode I/O yang diprogram (*programmed I/O*), modul I/O akan melakukan tindakan yang diminta dan kemudian mengatur bit yang sesuai dalam register status I/O (Gambar 4.5a). Modul I/O tidak mengambil tindakan lebih lanjut untuk memperingatkan prosesor. Secara khusus, itu tidak mengganggu prosesor. Dengan demikian, merupakan tanggung jawab prosesor untuk secara berkala memeriksa status modul I/O sampai menemukan bahwa operasi selesai. Untuk menjelaskan teknik I/O yang diprogram, dapat dilakukan dari sudut pandang perintah I/O yang dikeluarkan oleh prosesor ke modul I/O, dan kemudian dari sudut pandang instruksi I/O yang dijalankan oleh prosesor.



Sumber: (Stallings, 2016)
Gambar 4.5. Tiga Teknik untuk Menginput Block Data

Perintah I/O

Untuk menjalankan instruksi terkait I/O, prosesor mengeluarkan alamat, menentukan modul I/O dan perangkat eksternal tertentu, dan perintah I/O. Ada empat jenis perintah I/O yang dapat diterima modul I/O ketika ditangani oleh prosesor:

1. **Control:** Digunakan untuk mengaktifkan periferal dan memberi tahu apa yang harus dilakukan. Misalnya, unit pita magnetik dapat diperintahkan untuk memundurkan atau untuk bergerak maju satu catatan. Perintah-perintah ini dirancang untuk jenis perangkat periferal tertentu.
2. **Test:** Digunakan untuk menguji berbagai kondisi status yang terkait dengan modul I/O dan periferalnya. Prosesor akan ingin tahu bahwa perangkat yang menarik dihidupkan dan tersedia untuk digunakan. Ini juga ingin tahu apakah operasi I/O terbaru selesai dan jika ada kesalahan terjadi.

3. **Read**: Menyebabkan modul I/O mendapatkan item data dari periferal dan menempatkannya di buffer internal (digambarkan sebagai register data pada Gambar 4.5). Prosesor kemudian dapat memperoleh item data dengan meminta agar modul I/O menempatkannya di bus data.
4. **Write**: Menyebabkan modul I/O untuk mengambil item data (*byte* atau *word*) dari bus data dan kemudian mengirimkan item data tersebut ke perangkat.

Gambar 4.5a adalah contoh *flowchart* penggunaan I/O terprogram untuk membaca blok data dari perangkat periferal (misalnya: Catatan dari tape) ke dalam memori. Data dibaca dalam satu *word* (misalnya 16 bit) sekaligus. Untuk setiap *word* yang dibaca, prosesor harus tetap dalam siklus pemeriksaan status hingga menentukan bahwa *word* tersebut tersedia dalam register data modul-I/O. Satu kelemahan metode ini adalah proses yang memakan waktu yang membuat prosesor sibuk membaca status I/O.

I/O Instruction

Dengan I/O yang diprogram, terdapat korespondensi yang erat antara instruksi terkait I/O yang diambil prosesor dari memori dan perintah I/O yang dikeluarkan prosesor pada modul I/O untuk menjalankan instruksi. Artinya, instruksi mudah dipetakan ke dalam perintah I/O, dan seringkali ada hubungan satu-ke-satu yang sederhana. Bentuk instruksi tergantung pada cara di mana perangkat eksternal ditangani.

Biasanya, akan ada banyak perangkat I/O yang terhubung melalui modul I/O ke sistem. Setiap perangkat diberi pengenal atau alamat unik. Ketika prosesor mengeluarkan perintah I/O, perintah tersebut berisi alamat perangkat yang diinginkan. Dengan demikian, setiap modul I/O harus menginterpretasikan garis alamat untuk menentukan apakah perintah itu untuk dirinya sendiri.

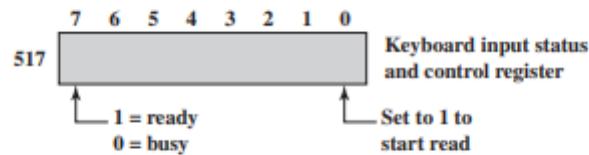
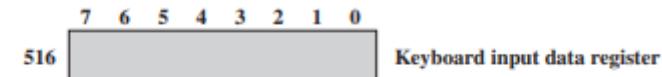
Ketika prosesor, memori utama, dan I/O berbagi bus umum, dua mode pengalamatan dimungkinkan: I/O dipetakan pada memori (*memory-mapped I/O*) dan I/O terisolasi (*isolated I/O*). Dengan I/O yang dipetakan dengan memori, hanya ada satu ruang alamat yang digunakan untuk lokasi memori dan perangkat I/O. Prosesor memperlakukan register status dan data dari modul I/O sebagai lokasi memori dan menggunakan instruksi mesin yang sama untuk mengakses memori dan perangkat I/O. Jadi, misalnya, dengan 10 baris alamat, total gabungan $2^{10} = 1024$ lokasi memori dan alamat I/O dapat didukung, dalam kombinasi apa pun.

Dengan I/O yang dipetakan dengan memori, satu baris baca dan satu baris tulis diperlukan di bus. Atau, bus dapat dilengkapi dengan memori baca dan tulis plus baris perintah *input* dan *output*. Baris perintah menentukan apakah alamat merujuk ke lokasi memori atau perangkat I/O. Berbagai alamat lengkap mungkin tersedia untuk keduanya. Sekali lagi, dengan 10 garis alamat, sistem sekarang dapat mendukung 1024 lokasi memori dan 1024 alamat I/O. Karena ruang alamat untuk I/O terisolasi dari itu untuk memori, ini disebut sebagai I/O terisolasi.

Gambar 4.5 menunjukkan perbedaan kedua teknik I/O terprogram (*memory-mapped* dan *Isolated*). Gambar 4.5a menunjukkan bagaimana antarmuka untuk perangkat *input* sederhana seperti keyboard terminal mungkin muncul ke programmer menggunakan *memory-mapped I/O*. Asumsikan alamat 10-bit, dengan memori 512-bit (lokasi 0–511) dan hingga 512 alamat I/O (lokasi 512–1023). Dua alamat didedikasikan untuk *input keyboard* dari terminal tertentu. Alamat 516 merujuk pada register data dan alamat 517 mengacu pada register status, yang juga berfungsi sebagai register kontrol untuk menerima perintah prosesor. Program yang ditampilkan akan membaca 1 *byte* data dari keyboard ke register akumulator di prosesor. Perhatikan bahwa prosesor *loop* sampai *byte* data tersedia.

Dengan I/O terisolasi (Gambar 4.6b), port I/O hanya dapat diakses oleh perintah I/O khusus, yang mengaktifkan jalur perintah I/O pada bus. Jika I/O terisolasi digunakan, hanya ada beberapa instruksi I/O. Dengan demikian, keuntungan dari *memory-mapped I/O* adalah bahwa instruksi yang digunakan

untuk baca/tulis memori dapat digunakan juga untuk baca/tulis I/O, memungkinkan pemrograman yang lebih efisien. Kerugiannya adalah ruang alamat memori yang berharga habis.



ADDRESS	INSTRUCTION	OPERAND	COMMENT
200	Load AC	"1"	Load accumulator
	Store AC	517	Initiate keyboard read
202	Load AC	517	Get status byte
	Branch if Sign = 0	202	Loop until ready
	Load AC	516	Load data byte

(a) Memory-mapped I/O

ADDRESS	INSTRUCTION	OPERAND	COMMENT
200	Load I/O	5	Initiate keyboard read
201	Test I/O	5	Check for completion
	Branch Not Ready	201	Loop until complete
	In	5	Load data byte

(b) Isolated I/O

Sumber: (Stallings, 2016)
Gambar 4.6. Memory-Mapped dan Isolated I/O

B. Interrupt Driven I/O

Masalah dengan I/O terprogram adalah bahwa prosesor harus menunggu lama untuk modul I/O yang menjadi perhatian untuk penerimaan atau pengiriman data. Prosesor, sambil menunggu, harus berulang kali menginterogasi status modul I/O. Akibatnya, tingkat kinerja seluruh sistem sangat menurun.

Alternatifnya adalah interrupt driven I/O. Agar prosesor mengeluarkan perintah I/O ke sebuah modul dan kemudian melanjutkan untuk melakukan pekerjaan lain yang bermanfaat. Modul I/O kemudian akan menginterupsi prosesor untuk meminta layanan ketika siap untuk bertukar data dengan prosesor. Prosesor kemudian melakukan transfer data, seperti sebelumnya, dan kemudian melanjutkan pemrosesan sebelumnya.

Dari sudut pandang modul I/O, modul I/O menerima perintah READ dari prosesor. Modul I/O kemudian mulai membaca data dari perangkat terkait. Setelah data berada dalam register data modul, modul memberi sinyal interupsi kepada prosesor melalui jalur kontrol. Modul kemudian menunggu hingga datanya diminta oleh prosesor. Ketika permintaan dibuat, modul menempatkan datanya di bus data dan kemudian siap untuk operasi I/O lainnya.

Dari sudut pandang prosesor, tindakan untuk *input* adalah sebagai berikut. Prosesor mengeluarkan perintah READ. Itu kemudian padam dan melakukan sesuatu yang lain (misalnya Prosesor dapat

bekerja pada beberapa program yang berbeda pada saat yang sama). Pada akhir setiap siklus instruksi, prosesor memeriksa interupsi. Ketika interupsi dari modul I/O terjadi, prosesor menyimpan konteks (misalnya: PC dan register prosesor) dari program saat ini, dan memproses interupsi. Dalam hal ini, prosesor membaca *word* data dari modul I/O dan menyimpannya dalam memori. Kemudian mengembalikan konteks program yang sedang dikerjakannya (atau program lain) dan melanjutkan eksekusi.

Gambar 4.5b menunjukkan penggunaan interupsi I/O untuk membaca dalam satu blok data. Bandingkan ini dengan Gambar 4.5a. *Interrupt driven* I/O lebih efisien daripada I/O yang diprogram karena menghilangkan menunggu yang tidak perlu. Namun, interupsi I/O masih menghabiskan banyak waktu prosesor, karena setiap *word* data yang berpindah dari memori ke modul I/O atau dari modul I/O ke memori harus melewati prosesor.

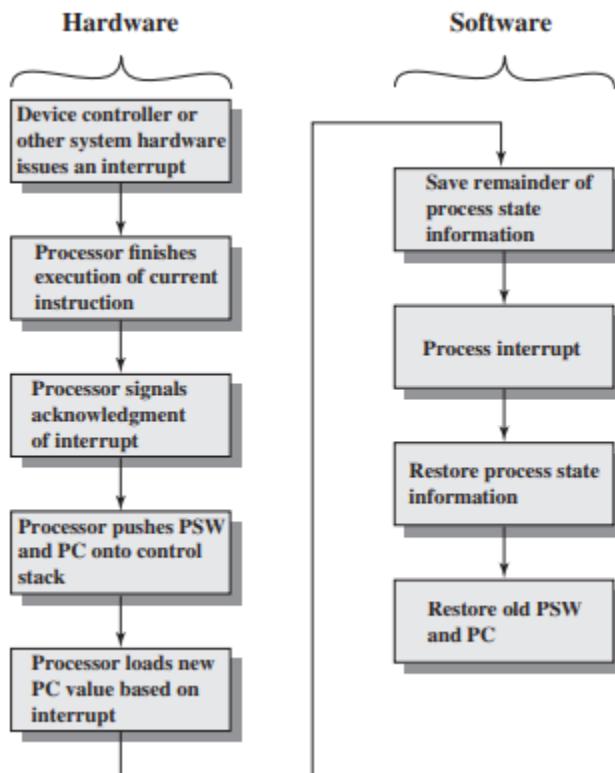
Pemrosesan interupsi

Terjadinya interupsi memicu sejumlah peristiwa, baik dalam perangkat keras prosesor dan perangkat lunak. Gambar 4.7 menunjukkan urutan yang khas ketika perangkat I/O menyelesaikan operasi I/O. Urutan kejadian perangkat keras berikut terjadi:

1. Perangkat mengeluarkan sinyal permintaan interupsi ke prosesor.
2. Prosesor menyelesaikan eksekusi instruksi saat ini sebelum menanggapi interupsi.
3. Prosesor mengirimkan sinyal pengakuan (*acknowledge*) ke perangkat yang mengeluarkan sinyal permintaan interupsi. Pengakuan ini memungkinkan perangkat untuk menghapus sinyal interupsi.
4. Prosesor sekarang perlu bersiap untuk mentransfer kontrol ke rutin interupsi. Untuk memulai, perlu menyimpan informasi yang diperlukan untuk melanjutkan program saat ini pada titik interupsi. Informasi minimum yang diperlukan adalah (a) status prosesor, yang terkandung dalam register yang disebut *Program Status Word (PSW)*; dan (b) lokasi instruksi berikutnya yang akan dieksekusi, yang terkandung dalam **PC**. Ini dapat di-PUSH ke stack.
5. Prosesor sekarang memuat **PC** dengan lokasi program penanganan interupsi yang akan merespons interupsi ini. Tergantung pada arsitektur komputer dan desain sistem operasi, mungkin ada satu program; satu program untuk setiap jenis interupsi; atau satu program untuk setiap perangkat dan setiap jenis interupsi. Jika ada lebih dari satu rutinitas penanganan interupsi, prosesor harus menentukan yang mana yang akan dipanggil. Informasi ini mungkin telah dimasukkan dalam sinyal interupsi asli, atau prosesor mungkin harus mengeluarkan permintaan ke perangkat yang mengeluarkan permintaan interupsi untuk mendapatkan respons yang berisi informasi yang diperlukan. Setelah **PC** dimuat, prosesor melanjutkan ke siklus instruksi berikutnya, yang dimulai dengan pengambilan instruksi. Karena pengambilan instruksi ditentukan oleh isi dari **PC**, hasilnya adalah kontrol ditransfer ke *program interrupt-handler*.
6. Pada titik ini, **PC** dan **PSW** yang berkaitan dengan program yang terputus telah disimpan pada *stack* sistem. Namun, ada informasi lain yang dianggap sebagai bagian dari "keadaan" program pelaksana. Secara khusus, isi register prosesor perlu disimpan, karena register ini dapat digunakan oleh pengendali interupsi. Jadi, semua nilai ini, ditambah informasi status lainnya, perlu disimpan. Biasanya, pengendali interupsi akan mulai dengan menyimpan konten dari semua register pada stack. Gambar 4.4a menunjukkan contoh sederhana. Dalam hal ini, *user program* terganggu setelah instruksi di lokasi N. Isi dari semua register ditambah alamat dari instruksi berikutnya (N +1) didorong ke stack. Penunjuk *stack* diperbarui untuk menunjuk ke bagian atas *stack* yang baru, dan **PC** diperbarui untuk menunjuk ke awal rutin layanan interupsi.
7. *Interrupt handler* selanjutnya memproses interupsi. Ini termasuk pemeriksaan informasi status yang berkaitan dengan operasi I/O atau peristiwa lain yang menyebabkan interupsi. Ini mungkin juga melibatkan pengiriman perintah atau ucapan terima kasih tambahan ke perangkat I/O.

8. Ketika pemrosesan interupsi selesai, nilai register yang disimpan diambil dari *stack* dan dikembalikan ke register (Gambar 4.4b).
9. Tindakan terakhir adalah mengembalikan nilai **PSW** dan program counter dari stack. Akibatnya, instruksi selanjutnya yang akan dieksekusi akan berasal dari program yang sebelumnya terganggu.

Perhatikan bahwa penting untuk menyimpan semua informasi status tentang program yang terganggu untuk dimulainya kembali nanti. Ini karena interupsi bukanlah panggilan rutin dari program. Sebaliknya, interupsi dapat terjadi kapan saja dan karenanya pada setiap saat dalam pelaksanaan *user program*. Kejadiannya tidak dapat diprediksi.



Sumber: (Stallings, 2016)
Gambar 4.7. Pemrosesan Instrupsi Sederhana

Masalah Desain

Dua masalah desain muncul dalam mengimplementasikan interupsi I/O. Pertama, karena hampir selalu ada modul I/O yang meminta interupsi, bagaimana prosesor menentukan perangkat mana yang mengeluarkan interupsi tersebut?. Dan kedua, jika beberapa interupsi telah terjadi secara bersamaan, bagaimana prosesor memutuskan mana yang akan diproses?

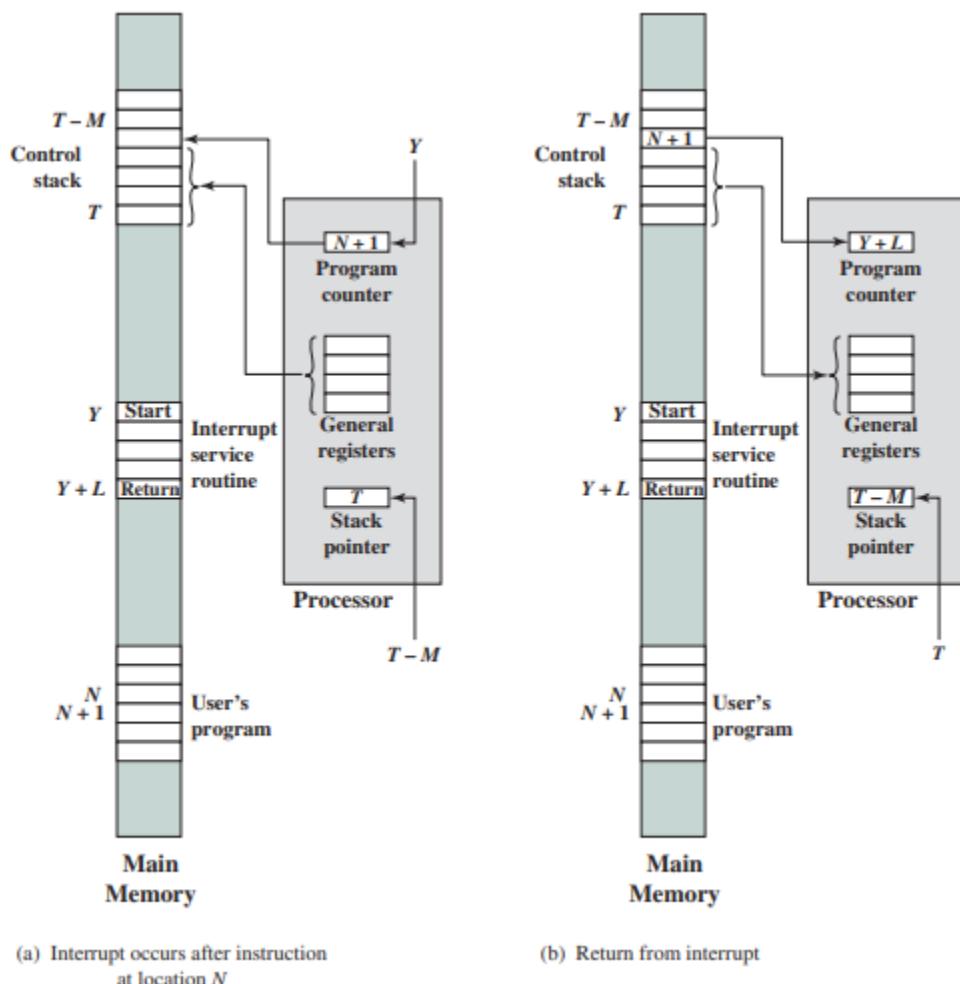
Dengan mempertimbangkan identifikasi perangkat terlebih dahulu, maka terdapat empat kategori umum teknik yang umum digunakan:

- Beberapa saluran interupsi (*multiple interrupt line*)
- Polling perangkat lunak (*software poll*)
- Rantai daisy (polling perangkat keras, vektor)
- Arbitrase bus (vektor)

Pendekatan yang paling mudah untuk masalah ini adalah dengan menyediakan beberapa jalur interupsi antara prosesor dan modul I/O. Namun, tidak praktis untuk mendedikasikan lebih dari beberapa jalur bus atau pin prosesor untuk memotong jalur. Akibatnya, bahkan jika beberapa baris

digunakan, kemungkinan setiap baris akan memiliki beberapa modul I/O yang menyertainya. Dengan demikian, salah satu dari tiga teknik lainnya harus digunakan pada setiap baris.

Salah satu alternatif adalah *pooling* perangkat lunak. Ketika prosesor mendeteksi interupsi, prosesor ini bercabang ke rutinitas layanan interupsi yang melakukan polling pada setiap modul I/O untuk menentukan modul mana yang menyebabkan interupsi. *Polling* bisa dalam bentuk baris perintah terpisah (misalnya: TEST I/O). Dalam hal ini, prosesor menaikkan TEST I/O dan menempatkan alamat modul I/O tertentu pada baris alamat. Modul I/O merespons secara positif jika ia mengatur interupsi. Atau, setiap modul I/O dapat berisi register status yang dapat dialamatkan. Prosesor kemudian membaca register status setiap modul I/O untuk mengidentifikasi modul yang menginterupsi. Setelah modul yang benar diidentifikasi, prosesor akan melakukan cabang ke layanan rutin perangkat khusus untuk perangkat itu.



Sumber: (Stallings, 2016)

Gambar 4.8. Perubahan dalam Memori dan Register pada sebuah Interupsi

Kerugian dari *pooling* perangkat lunak adalah memakan waktu. Teknik yang lebih efisien adalah dengan menggunakan rantai daisy, yang pada dasarnya memberikan *pooling* perangkat keras. Untuk interupsi, semua modul I/O berbagi jalur permintaan interupsi yang umum. Baris pengakuan interupsi daisy dirantai melalui modul. Saat prosesor merasakan interupsi, prosesor mengirimkan interupsi pengakuan. Sinyal ini merambat melalui serangkaian modul I/O hingga sampai ke modul yang meminta. Modul yang meminta biasanya merespons dengan menempatkan *word* pada baris data. *Word* ini disebut sebagai vektor dan merupakan alamat modul I/O atau pengidentifikasi unik lainnya.

Dalam kedua kasus tersebut, prosesor menggunakan vektor sebagai penunjuk ke rutinitas servis perangkat yang sesuai. Ini menghindari kebutuhan untuk menjalankan rutin layanan interupsi umum terlebih dahulu. Teknik ini disebut interrupt vektor (gambar 4.8).

Ada teknik lain yang memanfaatkan interupsi vektor, dan itu adalah arbitrase bus. Dengan arbitrase bus, modul I/O harus terlebih dahulu mendapatkan kendali atas bus sebelum dapat meningkatkan jalur permintaan interupsi. Dengan demikian, hanya satu modul yang dapat menaikkan garis sekaligus. Saat prosesor mendeteksi interupsi, prosesor merespons pada jalur interupsi pengakuan. Modul yang meminta kemudian menempatkan vektornya pada baris data.

Teknik-teknik yang disebutkan di atas berfungsi untuk mengidentifikasi modul I/O yang diminta. Mereka juga menyediakan cara menetapkan prioritas ketika lebih dari satu perangkat meminta layanan interupsi. Dengan beberapa jalur, prosesor hanya memilih jalur interupsi dengan prioritas tertinggi. Dengan polling perangkat lunak, urutan modul yang disurvei menentukan prioritasnya. Demikian pula, urutan modul pada rantai daisy menentukan prioritasnya. Akhirnya, arbitrase bus dapat menggunakan skema prioritas.

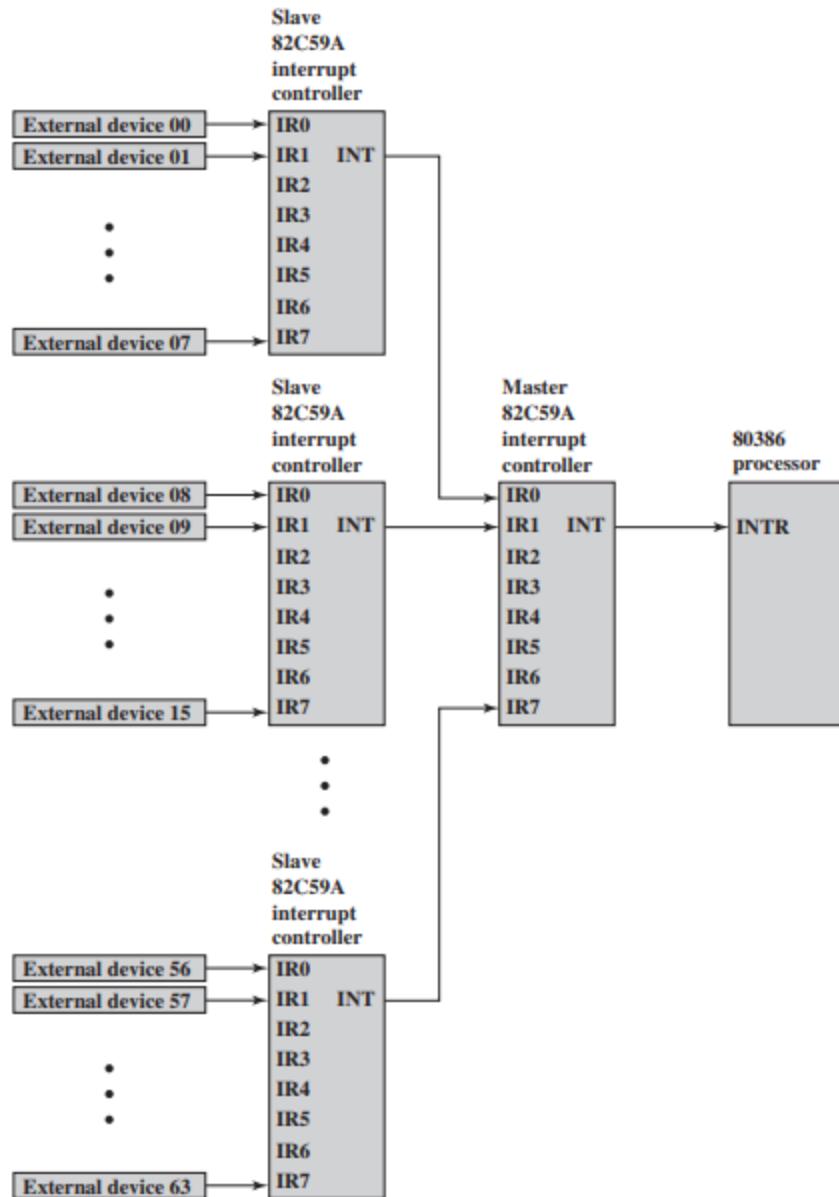
Pengontrol Interupsi Intel 82C59A

Intel 80386 menyediakan permintaan Interrupt tunggal (INTR) dan single Interrupt Acknowledge (INTA). Untuk memungkinkan 80386 menangani berbagai perangkat dan struktur prioritas, biasanya dikonfigurasi dengan arbiter interupsi eksternal, PIC 82C59A. Perangkat eksternal terhubung ke 82C59A, yang pada gilirannya terhubung ke 80386.

Gambar 4.9 menunjukkan penggunaan 82C59A untuk menghubungkan beberapa modul I/O untuk 80386. 82C59A tunggal dapat menangani hingga delapan modul. Jika kontrol untuk lebih dari delapan modul diperlukan, pengaturan kaskade dapat digunakan untuk menangani hingga 64 modul. Tanggung jawab 82C59A adalah pengelolaan interupsi. Ini menerima permintaan interupsi dari modul yang terpasang, menentukan interupsi mana yang memiliki prioritas tertinggi, dan kemudian memberi sinyal prosesor dengan menaikkan jalur INTR. Prosesor mengakui melalui jalur INTA. Ini meminta 82C59A untuk menempatkan informasi vektor yang sesuai pada bus data. Prosesor kemudian dapat melanjutkan untuk memproses interupsi dan berkomunikasi langsung dengan modul I/O untuk dibaca atau menulis data.

82C59A adalah sebuah *programmable interrupt controller* yang dapat diprogram. 80386 menentukan skema prioritas yang akan digunakan dengan menetapkan *control word* di 82C59A. Mode interupsi berikut dimungkinkan:

- Sepenuhnya bersarang (*fully nested*): Permintaan interupsi diminta dalam prioritas dari 0 (IRO) hingga 7 (IR7).
- Memutar (*rotating*): Dalam beberapa aplikasi sejumlah perangkat yang menginterupsi memiliki prioritas yang sama. Dalam mode ini, perangkat, setelah dilayani maka akan menerima prioritas terendah dalam grup.
- Topeng khusus (*special mask*): Ini memungkinkan prosesor menghambat interupsi dari perangkat tertentu.



Sumber: (Brey, 2009)
Gambar 4.9. Penggunaan PIC 825C59A

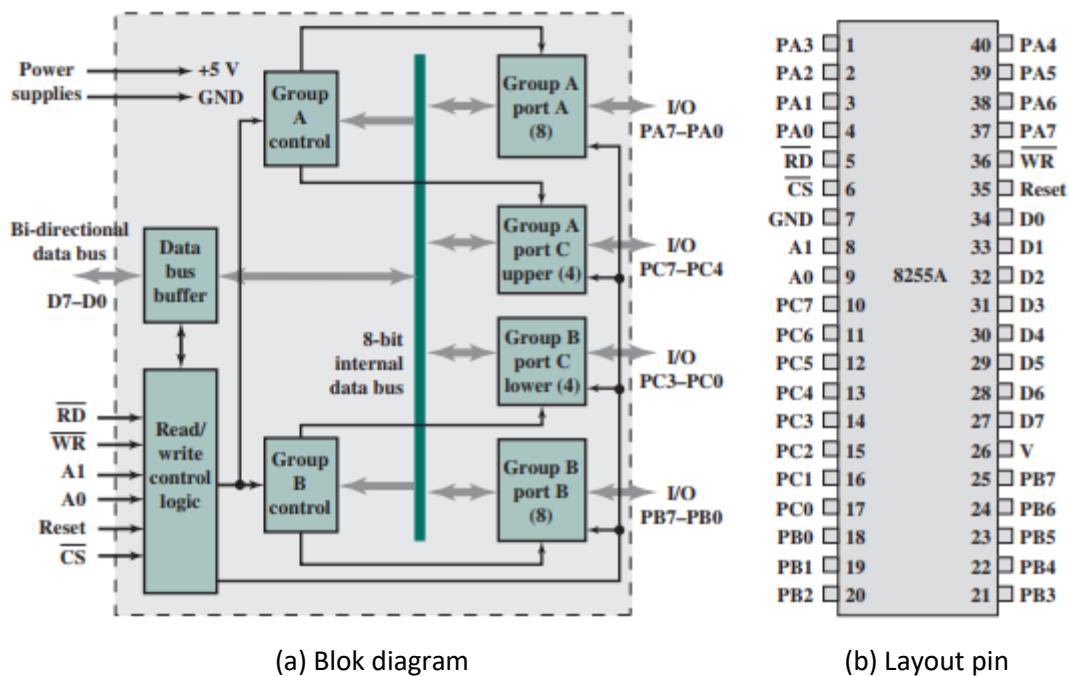
Antarmuka Periferal Yang Dapat Diprogram Intel 8255A

Sebagai contoh dari modul I/O yang digunakan untuk I/O terprogram dan I/O yang digerakkan oleh interupsi, mengacu pada Intel 8255A Programmable Peripheral Interface. PPI 8255A adalah modul I/O tujuan umum *chip* tunggal yang awalnya dirancang untuk digunakan dengan prosesor Intel 80386. Sejak itu telah dikloning oleh pabrikan lain dan merupakan *chip* pengontrol periferal yang banyak digunakan. Penggunaannya termasuk sebagai pengontrol untuk perangkat I/O sederhana untuk mikroprosesor dan dalam sistem tertanam, termasuk sistem mikrokontroler.

Arsitektur dan operasi PPI 8255

Gambar 4.10 menunjukkan diagram blok umum ditambah penugasan pin untuk paket 40-pin tempat paket tersebut ditempatkan. Seperti yang ditunjukkan pada tata letak pin, 8255A mencakup baris berikut:

- D0 – D7: Ini adalah jalur I/O data untuk perangkat. Semua informasi yang dibaca dan ditulis ke 8255A terjadi melalui delapan baris data ini.
- CS (*Input Pilih Chip*): Jika baris ini adalah logika 0, mikroprosesor dapat membaca dan menulis ke 8255A.
- RD (*Baca Input*): Jika baris ini adalah logika 0 dan *input CS* adalah logika 0, *output* data 8255A diaktifkan ke bus data sistem.
- WR (*Tulis Input*): Jika baris *input* ini adalah logika 0 dan *input CS* adalah logika 0, data ditulis ke 8255A dari bus data sistem.
- RESET: 8255A ditempatkan ke kondisi reset-nya jika jalur *input* ini adalah logika 1. Semua port periferal diatur ke mode *input*.
- PA0 – PA7, PB0 – PB7, PC0 – PC7: Saluran sinyal ini digunakan sebagai port I/O 8-bit. Mereka dapat dihubungkan ke perangkat periferal.
- A0, A1: Kombinasi logika dari dua jalur *input* ini menentukan register internal mana dari data 8255A yang ditulis atau dibaca.



Sumber: (Brey, 2009)
Gambar 4.10. PPI Intel 8255A

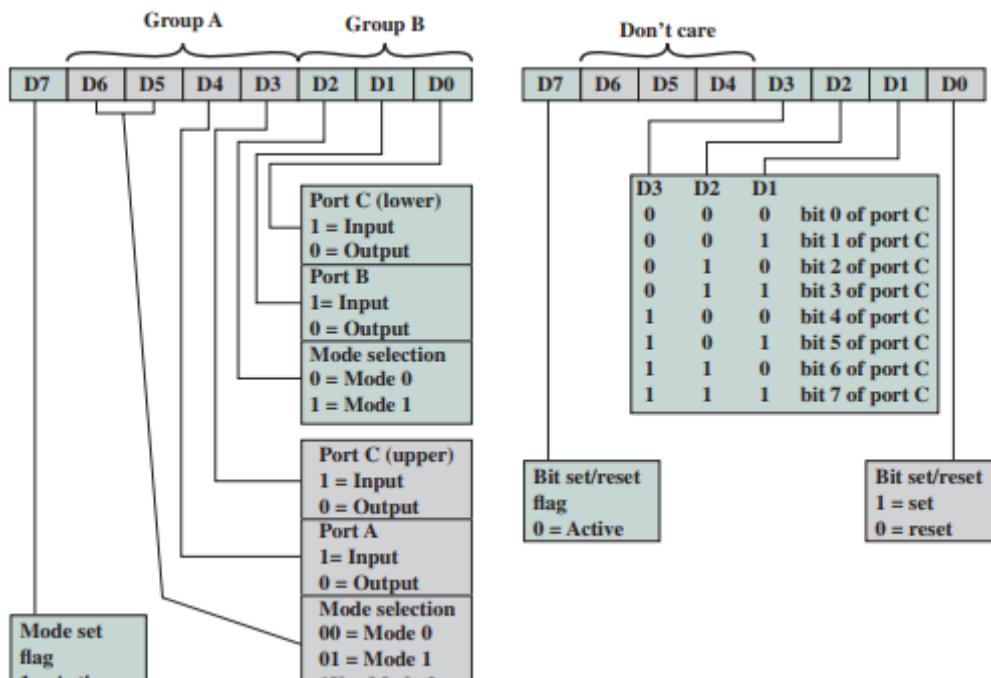
Sisi kanan diagram blok pada Gambar 4.10a adalah antarmuka eksternal 8255A. Garis 24 I/O dibagi menjadi tiga kelompok 8-bit (A, B, C). Setiap grup dapat berfungsi sebagai port I/O 8-bit, sehingga menyediakan koneksi untuk tiga perangkat periferal. Selain itu, grup C dibagi lagi menjadi grup 4-bit (CA dan CB), yang dapat digunakan bersama dengan port A/B I/O. Dikonfigurasi dengan cara ini, jalur grup C membawa sinyal kontrol dan status. Sisi kiri diagram blok adalah antarmuka internal ke bus sistem mikroprosesor. Ini termasuk bus data dua arah 8-bit (D0 hingga D7), yang digunakan untuk mentransfer data antara mikroprosesor dan port I/O dan untuk mentransfer informasi kontrol.

Prosesor mengontrol 8255A melalui register kontrol 8-bit pada prosesor. Prosesor dapat mengatur nilai register kontrol untuk menentukan berbagai mode operasi dan konfigurasi. Dari sudut pandang prosesor, ada port kontrol, dan bit register kontrol diatur dalam prosesor dan kemudian dikirim ke port kontrol melalui jalur D0 – D7. Dua baris alamat menentukan salah satu dari tiga port I/O atau register kontrol, sebagai berikut:

A1	A2	Select
0	0	Port A
0	1	Port B
1	0	Port C
1	1	Control register

Jadi, ketika prosesor menetapkan A1 dan A2 ke 1, 8255A mengartikan nilai 8-bit pada bus data sebagai *control word*. Ketika prosesor mentransfer *control word* 8-bit dengan garis D7 yang diatur ke 1 (Gambar 4.11a), *control word* digunakan untuk mengonfigurasikan mode operasi dari 24 jalur I/O. Tiga mode tersebut adalah:

- 1) Mode 0: Ini adalah mode I/O dasar. Tiga kelompok delapan garis eksternal berfungsi sebagai tiga port I/O 8-bit. Setiap port dapat ditunjuk sebagai *input* atau *output*. Data hanya dapat dikirim ke port jika port didefinisikan sebagai *output*, dan data hanya dapat dibaca dari port jika port diatur ke *input*.
- 2) Mode 1: Dalam mode ini, port A dan B dapat dikonfigurasikan sebagai *input* atau *output*, dan jalur dari port C berfungsi sebagai garis kontrol untuk A dan B. Sinyal kontrol melayani dua tujuan utama: "berjabat tangan" dan permintaan interupsi. Berjabat tangan adalah mekanisme pengaturan waktu yang sederhana. Satu jalur kontrol digunakan oleh pengirim sebagai jalur DATA READY, untuk menunjukkan kapan data hadir pada jalur data I/O. Baris lain digunakan oleh penerima sebagai ACKNOWLEDGE, menunjukkan bahwa data telah dibaca dan jalur data dapat dihapus. Jalur lain dapat ditetapkan sebagai jalur INTERRUPT REQUEST dan diikat kembali ke bus sistem.
- 3) Mode 2: Ini adalah mode dua arah. Dalam mode ini, port A dapat dikonfigurasikan sebagai jalur *input* atau *output* untuk lalu lintas dua arah pada port B, dengan jalur port B memberikan arah yang berlawanan. Sekali lagi, garis port C digunakan untuk pensinyalan kendali.

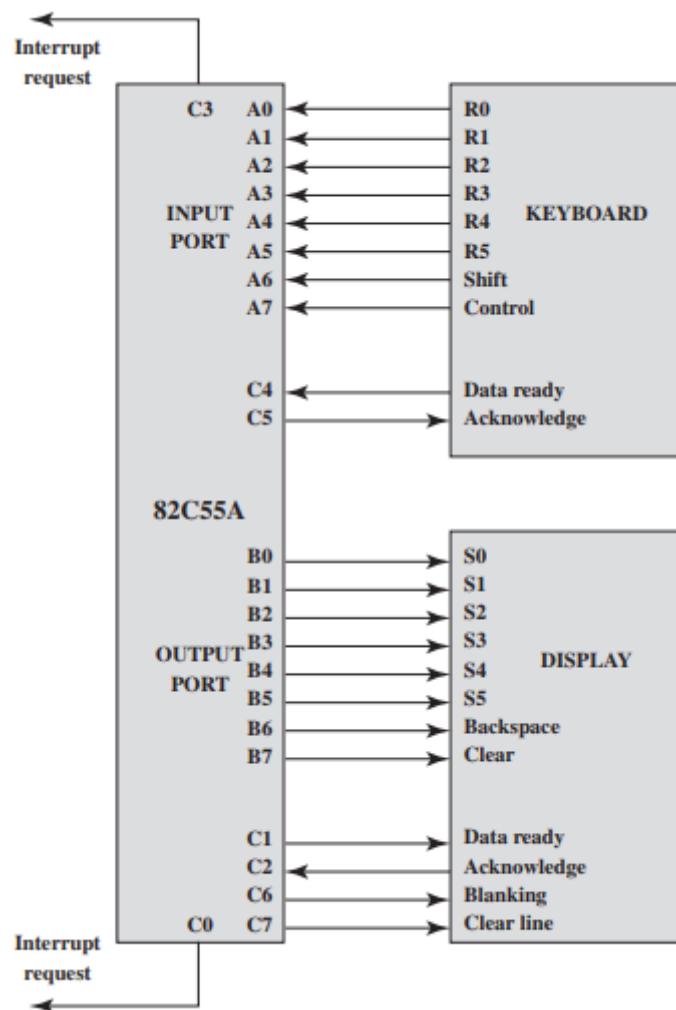


Sumber: (Brey, 2009)
Gambar 4.11. Control Word PPI Intel 8255A

Saat prosesor menetapkan D7 ke 0 (Gambar 4.11b), *control word* digunakan untuk memprogram nilai bit port C secara terpisah. Fitur ini jarang digunakan.

Contoh keyboard/Dsplay

Karena 8255A dapat diprogram melalui register kontrol, ia dapat digunakan untuk mengontrol berbagai perangkat periferal sederhana. Gambar 4.12 menggambarkan penggunaannya untuk mengontrol keyboard-terminal tampilan. Keyboard menyediakan 8 bit *input*. Dua bit ini, SHIFT dan CONTROL, memiliki arti khusus untuk program penanganan keyboard yang dijalankan pada prosesor. Namun, interpretasi ini transparan untuk 8255A, yang hanya menerima 8 bit data dan menyajikannya pada bus data sistem. Dua garis kontrol handshaking disediakan untuk digunakan dengan keyboard. Layar juga dihubungkan oleh port data 8-bit. Sekali lagi, dua bit memiliki arti khusus yang transparan ke 8255A. Selain dua garis jabat tangan, dua garis menyediakan fungsi kontrol tambahan.



Sumber: (Brey, 2009)

Gambar 4.12. Antarmuka Keyboard/Display dengan PPI Intel 8255A

I/O yang digerakkan oleh interupsi, meskipun lebih efisien daripada I/O yang diprogram sederhana, masih memerlukan intervensi aktif dari prosesor untuk mentransfer data antara memori dan modul I/O, dan setiap transfer data harus melintasi lintasan melalui prosesor. Dengan demikian, kedua bentuk I/O ini menderita dari dua kelemahan yang melekat:

1. Laju transfer I/O dibatasi oleh kecepatan prosesor dapat menguji dan memperbaiki perangkat.

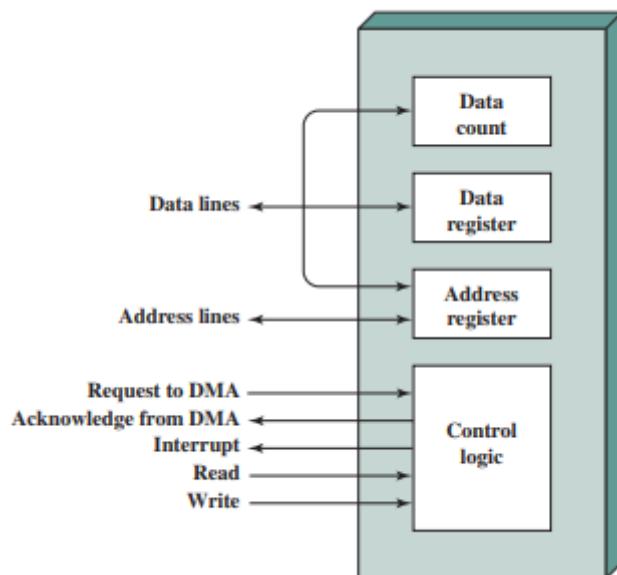
2. Prosesor terikat dalam mengelola transfer I/O; sejumlah instruksi harus dijalankan untuk setiap transfer I/O.

Ada semacam *trade-off* antara dua kelemahan ini. Pertimbangkan transfer blok data. Menggunakan I/O yang diprogram sederhana, prosesor ini didelegasikan untuk tugas I/O dan dapat memindahkan data pada tingkat yang agak tinggi, dengan biaya tidak melakukan hal lain. Interupsi I/O membebaskan prosesor sampai batas tertentu dengan mengorbankan laju transfer I/O. Namun demikian, kedua metode memiliki dampak buruk pada aktivitas prosesor dan kecepatan transfer I/O. Ketika sejumlah besar data akan dipindahkan, teknik yang lebih efisien diperlukan: akses memori langsung (DMA).

C. Direct Memory Access (DMA)

DMA melibatkan modul tambahan pada bus sistem. Modul DMA (Gambar 4.13) mampu meniru prosesor dan, tentu saja, mengambil alih kendali sistem dari prosesor. Perlu melakukan ini untuk mentransfer data ke dan dari memori melalui bus sistem. Untuk tujuan ini, modul DMA harus menggunakan bus hanya ketika prosesor tidak membutuhkannya, atau harus memaksa prosesor untuk menunda pengoperasian sementara. Teknik terakhir lebih umum dan disebut sebagai mencuri siklus, karena modul DMA berlaku mencuri siklus bus. Ketika prosesor ingin membaca atau menulis blok data, ia mengeluarkan perintah ke modul DMA, dengan mengirim ke modul DMA informasi berikut:

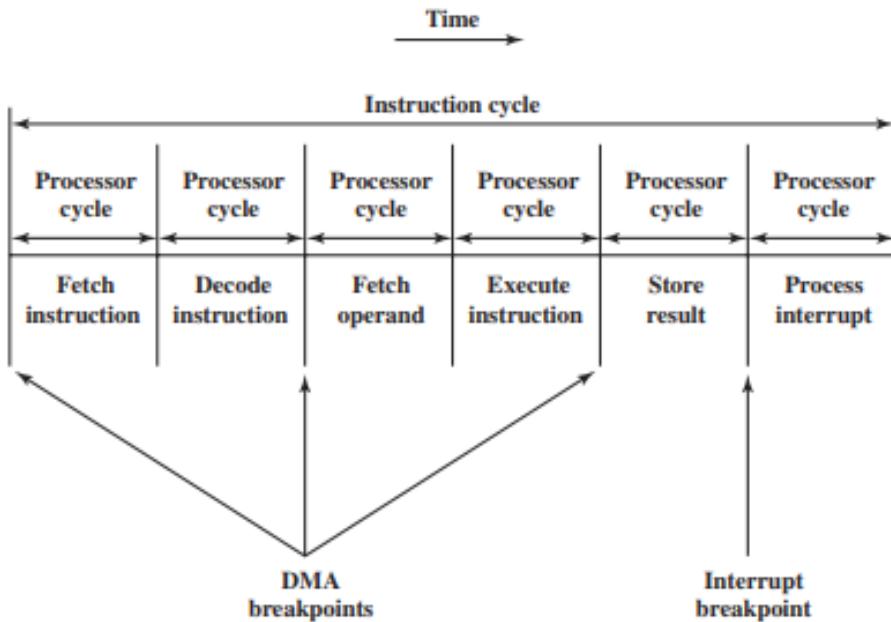
- Apakah diminta membaca atau menulis, menggunakan jalur kontrol baca atau tulis antara prosesor dan modul DMA.
- Alamat perangkat I/O yang terlibat, dikomunikasikan pada jalur data.
- Lokasi awal dalam memori untuk membaca atau menulis, dikomunikasikan pada baris data dan disimpan oleh modul DMA dalam register alamatnya.
- Jumlah *word* yang harus dibaca atau ditulis, sekali lagi dikomunikasikan melalui jalur data dan disimpan dalam register penghitungan data.



Sumber: (Stallings, 2016)

Gambar 4.13. Blok Diagram DMA yang umum

Prosesor kemudian dilanjutkan dengan pekerjaan lain. Ini telah mendelegasikan operasi I/O ini ke modul DMA. Modul DMA mentransfer seluruh blok data, satu *word* setiap kali, langsung ke atau dari memori, tanpa melalui prosesor. Ketika transfer selesai, modul DMA mengirimkan sinyal interupsi ke prosesor. Dengan demikian, prosesor hanya terlibat pada awal dan akhir transfer.



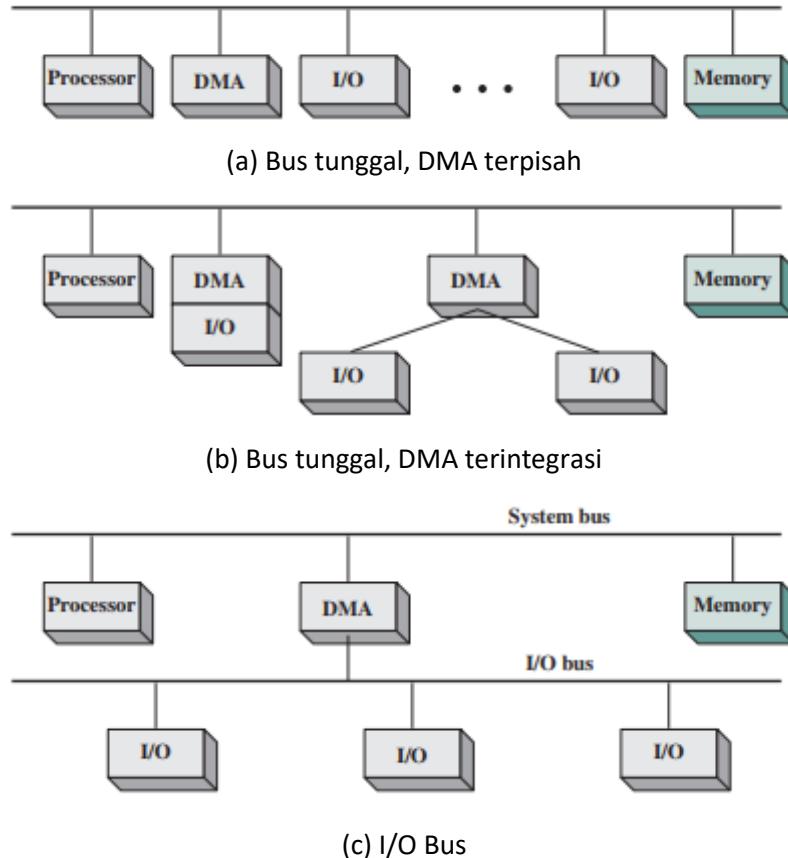
Sumber: (Stallings, 2016)

Gambar 4.14. DMA dan *Interrupt Breakpoint* selama Siklus Instruksi

Gambar 4.14 menunjukkan di mana dalam siklus instruksi prosesor dapat ditangguhkan. Dalam setiap kasus, prosesor ditangguhkan sesaat sebelum perlu menggunakan bus. Modul DMA kemudian mentransfer satu *word* dan mengembalikan kontrol ke prosesor. Perhatikan bahwa ini bukan interupsi; prosesor tidak menyimpan konteks dan melakukan sesuatu yang lain. Sebaliknya, prosesor berhenti selama satu siklus bus. Efek keseluruhannya adalah menyebabkan prosesor melakukan lebih lambat. Namun demikian, untuk transfer I/O beberapa *word*, DMA jauh lebih efisien daripada I/O yang digerakkan oleh terputus atau terprogram.

Mekanisme DMA dapat dikonfigurasi dalam berbagai cara. Beberapa kemungkinan ditunjukkan pada Gambar 4.15. Dalam contoh pertama, semua modul berbagi bus sistem yang sama. Modul DMA, bertindak sebagai prosesor pengganti, menggunakan I/O yang diprogram untuk bertukar data antara memori dan modul I/O melalui modul DMA. Konfigurasi ini, walaupun mungkin murah, jelas tidak efisien. Seperti halnya I/O yang diprogram dengan kontrol prosesor, setiap transfer *word* menghabiskan dua siklus bus.

Jumlah siklus bus yang diperlukan dapat dikurangi secara substansial dengan mengintegrasikan fungsi DMA dan I/O. Seperti ditunjukkan pada Gambar 4.15b, ini berarti ada jalur antara modul DMA dan satu atau lebih modul I/O yang tidak termasuk bus sistem. Logika DMA sebenarnya dapat menjadi bagian dari modul I/O, atau mungkin merupakan modul terpisah yang mengontrol satu atau lebih modul I/O. Konsep ini dapat diambil satu langkah lebih jauh dengan menghubungkan modul I/O ke modul DMA menggunakan bus I/O (Gambar 4.15c). Ini mengurangi jumlah antarmuka I/O dalam modul DMA menjadi satu dan menyediakan konfigurasi yang mudah diperluas. Dalam kedua kasus ini (Gambar 4.15b dan c), bus sistem yang berbagi modul DMA dengan prosesor dan memori digunakan oleh modul DMA hanya untuk bertukar data dengan memori. Pertukaran data antara modul DMA dan I/O berlangsung dari bus sistem.



Sumber: (Stallings, 2016)

Gambar 4.15. Konfigurasi Alternatif DMA

D. DMAC Intel 8237A

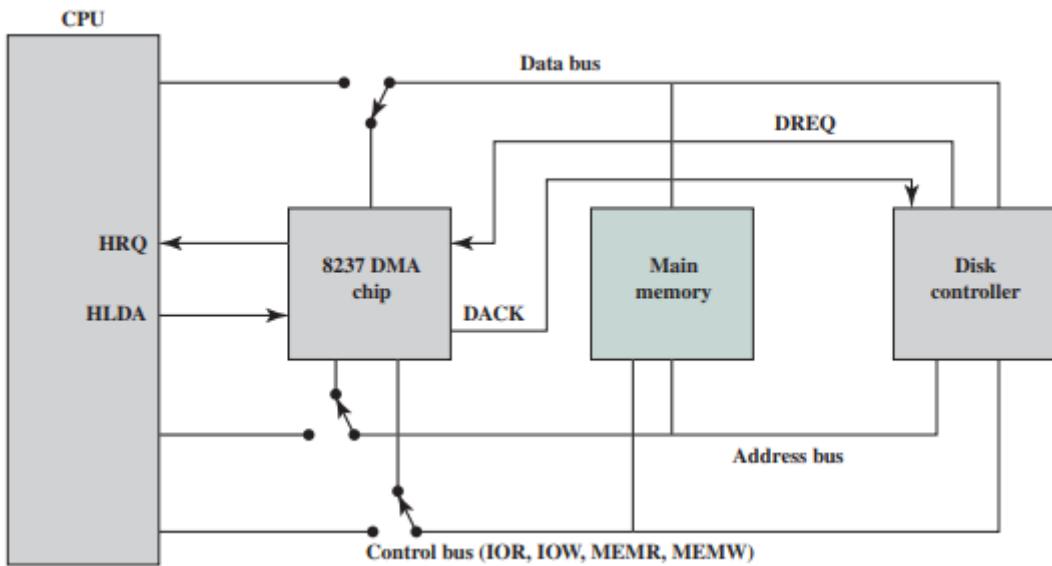
DMAC 8237A berinteraksi dengan kelompok prosesor 80x86 dan memori DRAM untuk memberikan kemampuan DMA. Gambar 4.16 menunjukkan lokasi modul DMA. Ketika modul DMA perlu menggunakan bus sistem (data, alamat, dan kontrol) untuk mentransfer data, ia mengirim sinyal yang disebut HOLD ke prosesor. Prosesor merespon dengan sinyal HLDA (tahan mengakui), menunjukkan bahwa modul DMA dapat menggunakan bus. Misalnya, jika modul DMA akan mentransfer blok data dari memori ke disk, ia akan melakukan hal berikut:

1. Perangkat periferal (seperti pengontrol disk) akan meminta layanan DMA dengan menarik DREQ (DMA request) tinggi.
2. DMA akan memberikan HRQ (Hold Request) yang tinggi, menandakan CPU melalui pin HOLDnya yang harus digunakan bus.
3. CPU akan menyelesaikan siklus bus saat ini (belum tentu instruksi ini) dan menanggapi permintaan DMA dengan menempatkan tinggi pada HDLA-nya (Hold Acknowledge), dengan demikian memberitahu DMA 8237 bahwa ia dapat melanjutkan dan menggunakan bus untuk melakukan tugasnya. HOLD harus tetap aktif tinggi selama DMA melakukan tugasnya.
4. DMA akan mengaktifkan DACK (DMA Acknowledge), yang memberi tahu perangkat periferal bahwa ia akan mulai mentransfer data.
5. DMA mulai mentransfer data dari memori ke perangkat dengan memasukkan alamat byte pertama dari blok pada bus alamat dan mengaktifkan MEMR, sehingga membaca byte dari memori ke dalam bus data; itu kemudian mengaktifkan TKI untuk menulisnya ke perangkat. Kemudian DMA mengurangi penghitungan dan menambah penunjuk alamat dan mengulangi proses ini sampai hitungan mencapai nol dan tugas selesai.

- Setelah DMA menyelesaikan tugasnya, ia akan menonaktifkan HRQ, menandakan kepada CPU bahwa ia telah mendapatkan kembali kendali atas bus-nya.

Sementara DMA menggunakan bus untuk mentransfer data, prosesor tidak bekerja. Demikian pula, ketika prosesor menggunakan bus, DMA idle. DMA 8237 dikenal sebagai pengontrol DMA *fly-by*. Ini berarti bahwa data yang dipindahkan dari satu lokasi ke lokasi lain tidak melewati *chip* DMA dan tidak disimpan dalam *chip* DMA. Oleh karena itu, DMA hanya dapat mentransfer data antara port I/O dan alamat memori, dan bukan antara dua port I/O atau dua lokasi memori. Namun, seperti yang dijelaskan kemudian, *chip* DMA dapat melakukan transfer memori ke memori melalui register.

DMAC 8237 berisi empat saluran permintaan DMA yang dapat diprogram secara independen, dan salah satu saluran mungkin aktif setiap saat. Saluran-saluran ini diberi nomor 0, 1, 2, dan 3.



Keterangan:

- DACK = DMA Acknowledge
- DREQ = DMA Request
- HLDA = HOLD Acknowledge
- HRQ = HOLD Request

Sumber: (Stallings, 2016)
Gambar 4.16. Sistem Bus pada DMAC 8237

8237 memiliki lima register kontrol/perintah untuk memprogram dan mengontrol operasi DMA melalui salah satu salurannya (Tabel 4.2):

- Perintah: Prosesor memuat register ini untuk mengontrol operasi DMA. D0 memungkinkan transfer memori ke memori, di mana saluran 0 digunakan untuk mentransfer byte ke register sementara 8237 dan saluran 1 digunakan untuk mentransfer byte dari register ke memori. Ketika memori ke memori diaktifkan, D1 dapat digunakan untuk menonaktifkan kenaikan/penurunan pada saluran 0 sehingga nilai tetap dapat ditulis ke dalam blok memori. D2 mengaktifkan atau menonaktifkan DMA.
- Status: Prosesor membaca register ini untuk menentukan status DMA. Bit D0 – D3 digunakan untuk mengindikasikan apakah saluran 0–3 telah mencapai TC (hitungan terminal). Bit D4 – D7 digunakan oleh prosesor untuk menentukan apakah saluran memiliki permintaan DMA yang tertunda.
- Mode: Prosesor mengatur register ini untuk menentukan mode operasi DMA. Bit D0 dan D1 digunakan untuk memilih saluran. Bit lain memilih berbagai mode operasi untuk saluran yang dipilih. Bit D2 dan D3 menentukan apakah transfer dari perangkat I/O ke memori (tulis) atau dari memori ke I/O (baca), atau operasi verifikasi. Jika D4 diatur, maka register alamat memori

dan register hitung dimuat ulang dengan nilai aslinya pada akhir transfer data DMA. Bit D6 dan D7 menentukan cara penggunaan 8237. Dalam mode tunggal, satu byte data ditransfer. Mode blok dan permintaan digunakan untuk transfer blok, dengan mode permintaan memungkinkan berakhirnya transfer secara prematur. Mode kaskade memungkinkan beberapa 8237 yang akan mengalir untuk memperluas jumlah saluran menjadi lebih dari 4.

- 4) Mask Tunggal: Prosesor mengatur register ini. Bit D0 dan D1 memilih saluran. Bit D2 menghapus atau mengatur bit mask untuk saluran itu. Melalui register ini bahwa *input* DREQ dari saluran tertentu dapat ditutup-tutupi (dinonaktifkan) atau dibuka kedoknya (diaktifkan). Sementara register perintah dapat digunakan untuk menonaktifkan seluruh *chip* DMA, register topeng tunggal memungkinkan programmer untuk menonaktifkan atau mengaktifkan saluran tertentu.
- 5) Semua Mask: Register ini mirip dengan register topeng tunggal kecuali bahwa keempat saluran dapat ditutup atau dibuka dengan satu operasi penulisan.

Selain itu, 8237A memiliki delapan register data: satu register alamat memori dan satu register jumlah untuk setiap saluran. Prosesor mengatur register ini untuk menunjukkan lokasi ukuran memori utama yang akan dipengaruhi oleh transfer.

Tabel 4.2. Register pada Intel DMAC 8237A

Bit	Command	Status	Mode	SingleMask	All Mask
D0	Memory-to-memory E/D	Channel 0 has reached TC	Channel select	Select channel mask bit	Clear/set channel 0 mask bit
D1	Channel 0 address hold E/D	Channel 1 has reached TC		Select channel mask bit	Clear/set channel 1 mask bit
D2	Controller E/D	Channel 2 has reached TC	Verify/write/read transfer	Clear/set mask bit	Clear/set channel 2 mask bit
D3	Normal/compressed timing	Channel 3 has reached TC		Not used	Clear/set channel 3 mask bit
D4	Fixed/rotating priority	Channel 0 request	Auto-initialization E/D		Not used
D5	Late/extended write selection	Channel 0 request	Address increment/decrement select		
D6	DREQ sense active high/low	Channel 0 request			
D7	DACK sense active high/low	Channel 0 request	Demand/single/block/cascade mode select		

Sumber: (Stallings, 2016)

4.5. Saluran I/O dan Prosesor

A. Evolusi Fungsi I/O

Dari perkembangan sistem komputer, terdapat pola peningkatan kompleksitas dan kecanggihan komponen individu. Langkah-langkah evolusi dapat diringkas sebagai berikut:

- 1) CPU secara langsung mengontrol perangkat periferal.
- 2) Modul pengontrol atau I/O ditambahkan. CPU menggunakan I/O yang diprogram tanpa henti. Dengan langkah ini, CPU menjadi agak bercerai dari detail spesifik antarmuka perangkat eksternal.

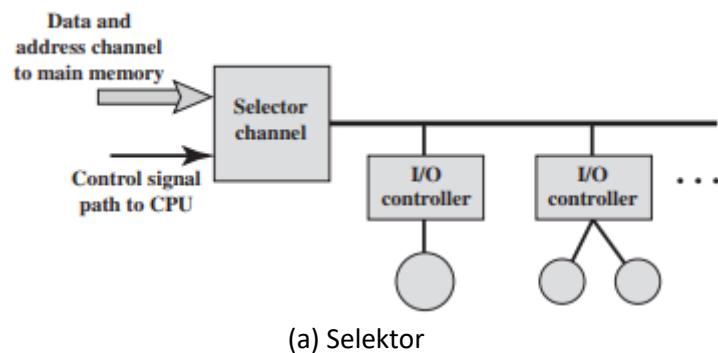
- 3) Konfigurasi yang sama seperti pada langkah 2 digunakan, tetapi sekarang interupsi digunakan. CPU tidak perlu menghabiskan waktu menunggu operasi I/O dilakukan, sehingga meningkatkan efisiensi.
- 4) Modul I/O diberikan akses langsung ke memori melalui DMA. Sekarang dapat memindahkan blok data ke atau dari memori tanpa melibatkan CPU, kecuali pada awal dan akhir transfer.
- 5) Modul I/O ditingkatkan menjadi prosesor dengan sendirinya, dengan set instruksi khusus yang dirancang untuk I/O. CPU mengarahkan prosesor I/O untuk menjalankan program I/O di memori. Prosesor I/O mengambil dan menjalankan instruksi ini tanpa intervensi CPU. Hal ini memungkinkan CPU untuk menentukan urutan kegiatan I/O dan hanya akan terganggu ketika seluruh urutan telah dilakukan.
- 6) Modul I/O memiliki memori lokal sendiri dan, pada kenyataannya, adalah komputer dengan sendirinya. Dengan arsitektur ini, satu set besar perangkat I/O dapat dikontrol, dengan keterlibatan CPU minimal. Penggunaan umum untuk arsitektur seperti itu adalah untuk mengontrol komunikasi dengan terminal interaktif. Prosesor I/O menangani sebagian besar tugas yang terlibat dalam mengendalikan terminal.

Di sepanjang jalur evolusi ini, semakin banyak fungsi I/O dilakukan tanpa keterlibatan CPU. CPU semakin terbebas dari tugas-tugas terkait I/O, meningkatkan kinerja. Dengan dua langkah terakhir (langkah 5, dan 6), perubahan besar terjadi dengan pengenalan konsep modul I/O yang mampu menjalankan suatu program. Untuk langkah 5, modul I/O sering disebut sebagai saluran I/O. Untuk langkah 6, istilah prosesor I/O sering digunakan. Namun, kedua istilah tersebut kadang-kadang diterapkan pada kedua situasi.

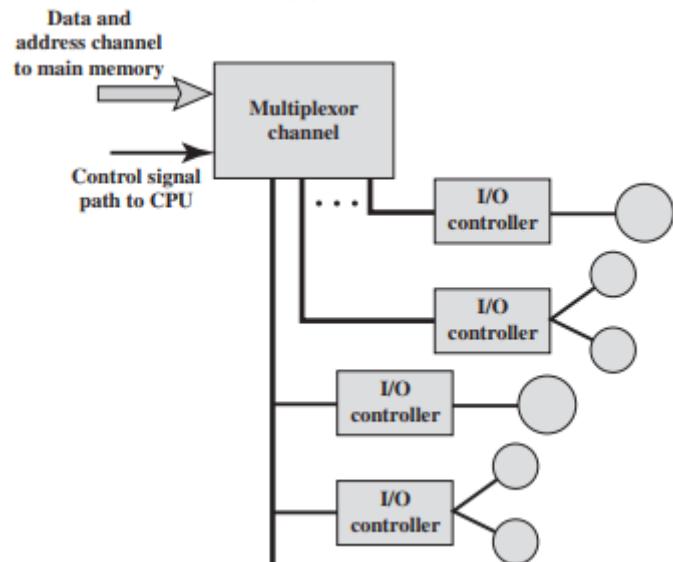
B. Karakteristik Saluran I/O

Saluran I/O merupakan perpanjangan dari konsep DMA. Saluran I/O memiliki kemampuan untuk menjalankan instruksi I/O, yang memberinya kendali penuh atas operasi I/O. Dalam sistem komputer dengan perangkat tersebut, CPU tidak menjalankan instruksi I/O. Instruksi tersebut disimpan dalam memori utama untuk dijalankan oleh prosesor tujuan khusus di saluran I/O itu sendiri. Dengan demikian, CPU memulai transfer I/O dengan menginstruksikan saluran I/O untuk menjalankan program dalam memori. Program akan menentukan perangkat, area memori untuk penyimpanan, prioritas, dan tindakan yang harus diambil untuk kondisi kesalahan tertentu. Saluran I/O mengikuti instruksi ini dan mengontrol transfer data.

Dua jenis saluran I/O (*I/O channel*) yang umum adalah selector dan multiplexor, seperti yang diilustrasikan dalam Gambar 4.17. Saluran pemilih (*selector channel*) mengontrol beberapa perangkat berkecepatan tinggi dan, pada satu waktu, didedikasikan untuk transfer data dengan salah satu perangkat tersebut. Dengan demikian, saluran I/O memilih satu perangkat dan memengaruhi transfer data. Setiap perangkat, atau satu set kecil perangkat, ditangani oleh pengontrol, atau modul I/O. Dengan demikian, saluran I/O berfungsi mengantikan CPU dalam mengendalikan pengontrol I/O ini. Saluran multiplexor (*multiplexor channel*) dapat menangani I/O dengan beberapa perangkat secara bersamaan. Untuk perangkat kecepatan rendah, multiplexer menerima atau mentransmisikan karakter secepat mungkin ke beberapa perangkat. Sebagai contoh, stream karakter yang dihasilkan dari tiga perangkat dengan laju yang berbeda dan aliran individual A1A2A3A4, B1B2B3B4, dan C1C2C3C4 mungkin A1B1C1A2C2A3B2C3A4, dan seterusnya. Untuk perangkat berkecepatan tinggi, multipleks blok atau interleaf blok data dari beberapa perangkat.



(a) Selektor



(a) Multipleksor

Sumber: (Stallings, 2016)

Gambar 4.17. Arsitekur I/O Channel

5. Sistem Interkoneksi

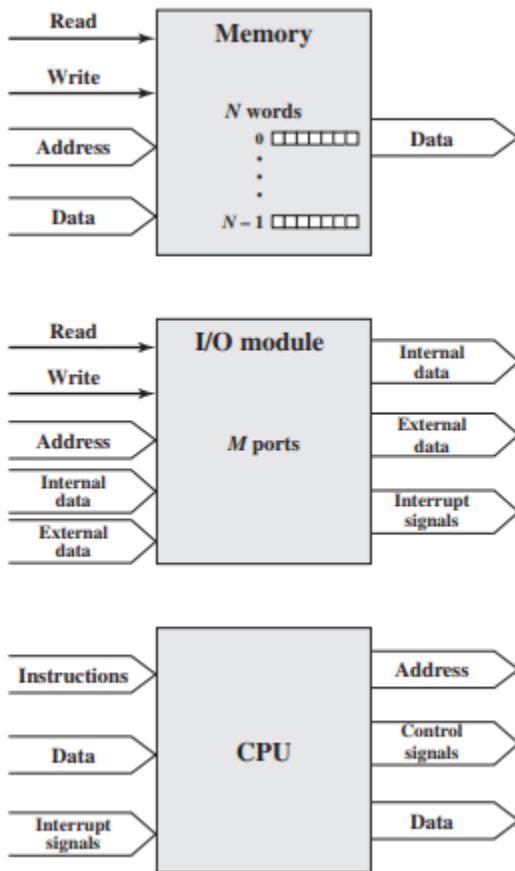
5.1 Struktur Interkoneksi

Komputer terdiri dari sekumpulan komponen atau modul dari tiga tipe dasar (prosesor, memori, I/O) yang saling berkomunikasi. Akibatnya, komputer adalah jaringan modul dasar. Jadi, harus ada jalur untuk menghubungkan modul. Kumpulan jalur yang menghubungkan berbagai modul disebut struktur interkoneksi. Desain struktur ini akan tergantung pada pertukaran yang harus dilakukan antar modul. Gambar 5.1 menunjukkan jenis pertukaran yang dibutuhkan dengan menunjukkan bentuk utama *input* dan *output* untuk setiap jenis modul:

- 1) Memori: Biasanya, modul memori akan terdiri dari N *word* dengan panjang yang sama. Setiap kata diberi alamat numerik yang unik ($0, 1, \dots, N - 1$). Sebuah *word* data dapat dibaca dari atau ditulis ke dalam memori. Sifat operasi ditunjukkan dengan membaca dan menulis sinyal kontrol. Lokasi operasi ditentukan oleh alamat.
- 2) Modul I/O: Dari sudut pandang internal (ke sistem komputer), I/O secara fungsional mirip dengan memori. Ada dua operasi, baca dan tulis. Selanjutnya, modul I/O dapat mengontrol lebih dari satu perangkat eksternal. Modul I/O dapat merujuk ke masing-masing antarmuka ke perangkat eksternal sebagai port dan memberikan masing-masing alamat unik (misalnya: $0, 1, \dots, M - 1$). Selain itu, ada jalur data eksternal untuk *input* dan *output* data dengan perangkat eksternal. Akhirnya, modul I/O mungkin dapat mengirim sinyal interupsi ke prosesor.
- 3) Prosesor: Prosesor membaca dalam instruksi dan data, menulis data setelah pemrosesan, dan menggunakan sinyal kontrol untuk mengontrol operasi keseluruhan sistem. Ini juga menerima sinyal interupsi.

Struktur interkoneksi harus mendukung jenis transfer berikut:

- Memori ke prosesor: Prosesor membaca instruksi atau unit data dari memori.
- Prosesor ke memori: Prosesor menulis unit data ke memori.
- I/O ke prosesor: Prosesor membaca data dari perangkat I/O melalui modul I/O.
- Prosesor ke I/O: Prosesor mengirimkan data ke perangkat I/O.
- I/O ke atau dari memori: Untuk dua kasus ini, modul I/O diperbolehkan untuk bertukar data secara langsung dengan memori, tanpa melalui prosesor, menggunakan akses memori langsung (DMA).



Sumber: (Stallings, 2016)

Gambar 5.1. Modul Komputer

Struktur interkoneksi harus mendukung jenis transfer berikut:

- Memori ke prosesor: Prosesor membaca instruksi atau unit data dari memori.
- Prosesor ke memori: Prosesor menulis unit data ke memori.
- I/O ke prosesor: Prosesor membaca data dari perangkat I/O melalui modul I/O.
- Prosesor ke I/O: Prosesor mengirimkan data ke perangkat I/O.
- I/O ke atau dari memori: Untuk dua kasus ini, modul I/O diizinkan untuk bertukar data secara langsung dengan memori, tanpa melalui prosesor, menggunakan akses memori langsung.

Selama bertahun-tahun, sejumlah struktur interkoneksi telah dicoba. Sejauh ini yang paling umum adalah (1) bus dan berbagai struktur multi-bus, dan (2) struktur interkoneksi *point-to-point* dengan transfer data paket.

5.2. Bus Interkoneksi

Bus adalah jalur komunikasi yang menghubungkan dua perangkat atau lebih. Karakteristik utama dari sebuah bus adalah bahwa itu adalah media transmisi bersama. Beberapa perangkat terhubung ke bus, dan sinyal yang dikirim oleh satu perangkat apa pun tersedia untuk penerimaan oleh semua perangkat lain yang terhubung ke bus. Jika dua perangkat mentransmisikan selama periode waktu yang sama, sinyal mereka akan tumpang tindih dan menjadi kacau. Dengan demikian, hanya satu perangkat pada satu waktu yang berhasil mentransmisikan.

Biasanya, bus terdiri dari beberapa jalur komunikasi, atau jalur. Setiap garis mampu mentransmisikan sinyal yang mewakili biner 1 dan biner 0. Seiring waktu, urutan angka biner dapat ditransmisikan melintasi satu baris. Secara bersama-sama, beberapa jalur bus dapat digunakan untuk mengirimkan

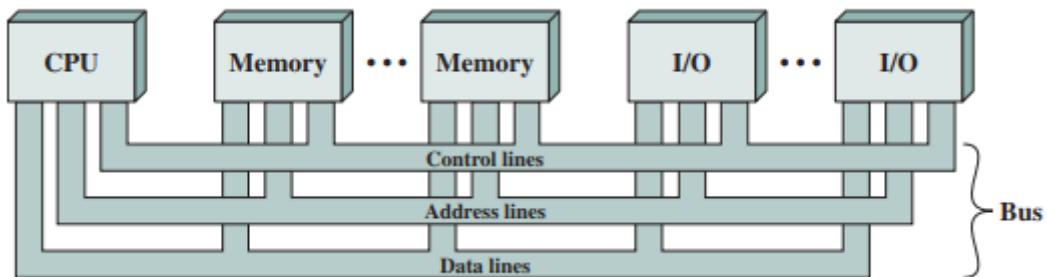
digit biner secara bersamaan (secara paralel). Misalnya, unit data 8-bit dapat ditransmisikan melalui delapan jalur bus.

Sistem komputer berisi sejumlah bus berbeda yang menyediakan jalur antar komponen pada berbagai tingkat hierarki sistem komputer. Bus yang menghubungkan komponen komputer utama (prosesor, memori, I/O) disebut bus sistem. Struktur interkoneksi komputer yang paling umum didasarkan pada penggunaan satu atau lebih bus sistem.

A. Struktur Bus

Bus sistem biasanya terdiri dari sekitar 50 hingga ratusan jalur atau garis terpisah. Setiap jalur diberi arti atau fungsi tertentu. Meskipun ada banyak desain bus yang berbeda, pada bus apa pun jalur dapat diklasifikasikan ke dalam tiga kelompok fungsional (Gambar 5.2): bus data, bus alamat, dan bus kontrol. Selain itu, mungkin ada jalur distribusi daya yang memasok daya ke modul yang terpasang. Bus data menyediakan jalur untuk memindahkan data di antara modul sistem. Garis-garis ini, secara kolektif, disebut bus data. Bus data dapat terdiri dari 32, 64, 128, atau bahkan lebih banyak jalur yang terpisah, jumlah jalur yang disebut sebagai lebar bus data. Karena setiap baris hanya dapat membawa 1 bit pada satu waktu, jumlah baris menentukan berapa banyak bit yang dapat ditransfer pada suatu waktu. Lebar bus data adalah faktor kunci dalam menentukan kinerja sistem secara keseluruhan. Misalnya, jika bus data memiliki lebar 32 bit dan setiap instruksi panjangnya 64 bit, maka prosesor harus mengakses modul memori dua kali selama setiap siklus instruksi.

Baris alamat digunakan untuk menunjuk sumber atau tujuan data pada bus data. Misalnya, jika prosesor ingin membaca *word* (8, 16, atau 32 bit) data dari memori, prosesor akan meletakkan alamat *word* yang diinginkan pada baris alamat. Jelas, lebar bus alamat menentukan kapasitas memori maksimum yang mungkin dari sistem. Selain itu, garis alamat umumnya juga digunakan untuk mengatasi port I/O. Biasanya, bit orde tinggi digunakan untuk memilih modul tertentu pada bus, dan bit orde bawah memilih lokasi memori atau port I/O dalam modul. Misalnya, pada bus alamat 8-bit, alamat 01111111 dan di bawahnya mungkin merujuk lokasi dalam modul memori (modul 0) dengan 128 *word* memori, dan alamat 10000000 dan di atasnya merujuk ke perangkat yang terpasang pada modul I/O.



Sumber: (Stallings, 2016)

Gambar 5.2. Skema interkoneksi bus

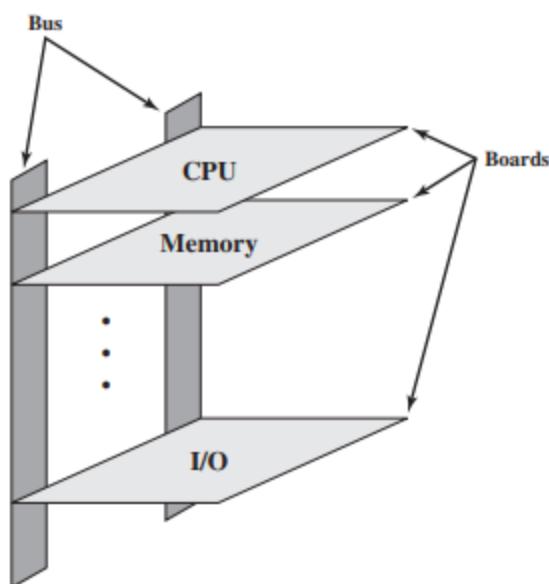
Bus kontrol digunakan untuk mengontrol akses ke dan penggunaan data dan garis alamat. Karena bus data dan alamat dibagikan oleh semua komponen, harus ada cara untuk mengendalikan penggunaannya. Sinyal kontrol mengirimkan informasi perintah dan waktu di antara modul sistem. Sinyal waktu menunjukkan validitas data dan informasi alamat. Sinyal perintah menentukan operasi yang akan dilakukan. Bus kontrol yang umum termasuk:

- **Memori write:** Menyebabkan data pada bus yang akan ditulis ke lokasi yang dituju
- **Memory read:** Menyebabkan data dari lokasi yang dituju untuk ditempatkan di bus
- **I/O write:** Menyebabkan data pada bus menjadi *output* ke port I/O yang dialamatkan
- **I/O read:** Menyebabkan data dari port I/O yang dituju untuk ditempatkan di bus
- **Transfer ACK:** Menunjukkan bahwa data telah diterima dari atau ditempatkan di bus
- **Bus request:** Menunjukkan bahwa suatu modul perlu menguasai bus

- **Bus grant:** Menunjukkan bahwa modul yang meminta telah diberikan kendali atas bus
- **Interupsi request:** Menunjukkan bahwa interupsi sedang menunggu
- **Interrupt ACK:** Mengakui bahwa interupsi yang tertunda telah dikenali
- **Clock:** Digunakan untuk menyinkronkan operasi
- **Reset:** Menginisialisasi semua modul

Pengoperasian bus adalah sebagai berikut. Jika satu modul ingin mengirim data ke yang lain, itu harus melakukan dua hal: (1) mendapatkan penggunaan bus, dan (2) mentransfer data melalui bus. Jika satu modul ingin meminta data dari modul lain, ia harus (1) mendapatkan penggunaan bus, dan (2) mentransfer permintaan ke modul lain melalui jalur kontrol dan alamat yang sesuai. Kemudian harus menunggu modul kedua untuk mengirim data.

Secara fisik, bus sistem sebenarnya adalah sejumlah konduktor listrik paralel. Dalam pengaturan bus klasik, konduktor ini adalah garis logam yang terukir dalam kartu atau papan (papan sirkuit tercetak). Bus meluas melintasi semua komponen sistem, yang masing-masing menyentuh beberapa atau semua jalur bus. Susunan fisik klasik digambarkan pada Gambar 5.3. Dalam contoh ini, bus terdiri dari dua kolom konduktor vertikal. Secara berkala di sepanjang kolom, ada titik lampiran dalam bentuk slot yang memanjang secara horizontal untuk mendukung papan sirkuit tercetak. Masing-masing komponen sistem utama menempati satu atau lebih papan dan colokan ke dalam bus di slot ini. Seluruh pengaturan ditempatkan dalam sasis. Skema ini masih dapat digunakan untuk beberapa bus yang terkait dengan sistem komputer. Namun, sistem modern cenderung memiliki semua komponen utama pada papan yang sama dengan lebih banyak elemen pada *chip* yang sama dengan prosesor. Dengan demikian, bus *on-chip* dapat menghubungkan prosesor dan memori *cache*, sedangkan bus *on-board* dapat menghubungkan prosesor ke memori utama dan komponen lainnya. Pengaturan ini paling nyaman. Sistem komputer kecil dapat diperoleh dan kemudian diperluas kemudian (lebih banyak memori, lebih banyak I/O) dengan menambahkan lebih banyak papan. Jika komponen pada papan gagal, papan itu dapat dengan mudah dilepas dan diganti.



Sumber: (Stallings, 2016)
Realisasi Fisik Arsitektur Bus yang umum

B. Hirarki Multi-Bus

Jika sejumlah besar perangkat terhubung ke bus, kinerja sistem secara keseluruhan akan berkurang. Ada dua penyebab utama, yaitu:

- 1) Secara umum, semakin banyak perangkat yang terpasang pada bus, semakin besar panjang bus dan karenanya semakin besar penundaan propagasi. Penundaan ini menentukan waktu yang diperlukan perangkat untuk mengoordinasikan penggunaan bus. Ketika kontrol bus lewat dari satu perangkat ke perangkat lainnya secara teratur, penundaan propagasi ini dapat secara nyata mempengaruhi kinerja.
- 2) Bus dapat menjadi hambatan karena permintaan transfer data agregat mendekati kapasitas bus. Masalah ini dapat diatasi sampai batas tertentu dengan meningkatkan laju data yang dapat dibawa oleh bus dan dengan menggunakan bus yang lebih luas (misalnya: meningkatkan data bus dari 32 menjadi 64 bit). Namun, karena laju data yang dihasilkan oleh perangkat yang terpasang (misalnya pengontrol grafis dan video, antarmuka jaringan) tumbuh dengan cepat, ini adalah perlombaan yang akhirnya ditakdirkan oleh satu bus tunggal.

Dengan demikian, sebagian besar sistem komputer menggunakan banyak bus, umumnya diletakkan dalam hierarki. Struktur tradisional yang khas ditunjukkan pada Gambar 5.3a. Ada bus lokal yang menghubungkan prosesor ke memori *cache* dan yang dapat mendukung satu atau lebih perangkat lokal. Pengontrol memori *cache* menghubungkan *cache* tidak hanya ke bus lokal ini, tetapi ke bus sistem yang terpasang semua modul memori utama. Penggunaan struktur *cache* mengisolasi prosesor dari persyaratan untuk mengakses memori utama secara rutin. Oleh karena itu, memori utama dapat dipindahkan dari bus lokal ke bus sistem. Dengan cara ini, transfer I/O ke dan dari memori utama melintasi bus sistem tidak mengganggu aktivitas prosesor.

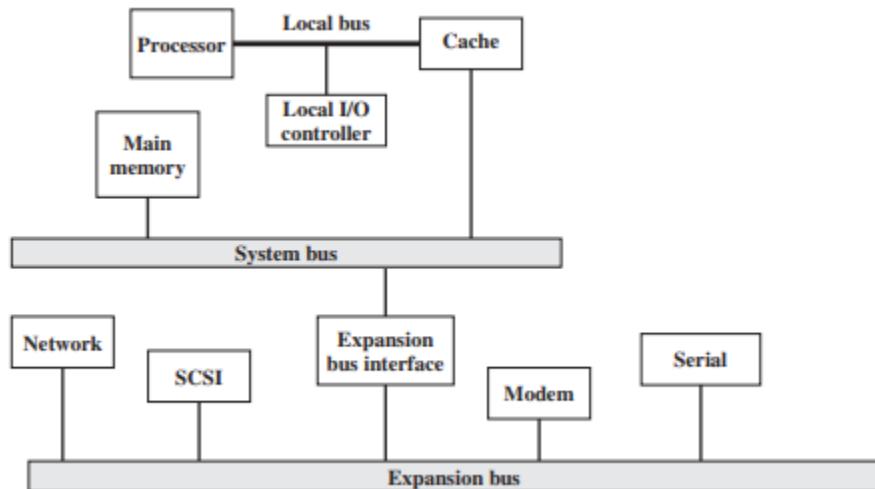
Dimungkinkan untuk menghubungkan pengontrol I/O secara langsung ke bus sistem. Solusi yang lebih efisien adalah dengan menggunakan satu atau lebih bus ekspansi untuk tujuan ini. Antarmuka bus ekspansi menyangga transfer data antara bus sistem dan pengontrol I/O pada bus ekspansi. Pengaturan ini memungkinkan sistem untuk mendukung berbagai perangkat I/O dan pada saat yang sama melindungi lalu lintas memori ke prosesor dari lalu lintas I/O.

Gambar 5.4a menunjukkan beberapa contoh khas perangkat I/O yang mungkin melekat pada bus ekspansi. Koneksi jaringan mencakup *Local Area Network* (LAN) seperti Ethernet 10-Mbps dan koneksi ke *Wide Area Network* (WAN) seperti jaringan packet-switching. SCSI (*Small Computer Scale Interface*) sendiri merupakan jenis bus yang digunakan untuk mendukung disk drive lokal dan periferal lainnya. Port serial dapat digunakan untuk mendukung printer atau pemindai.

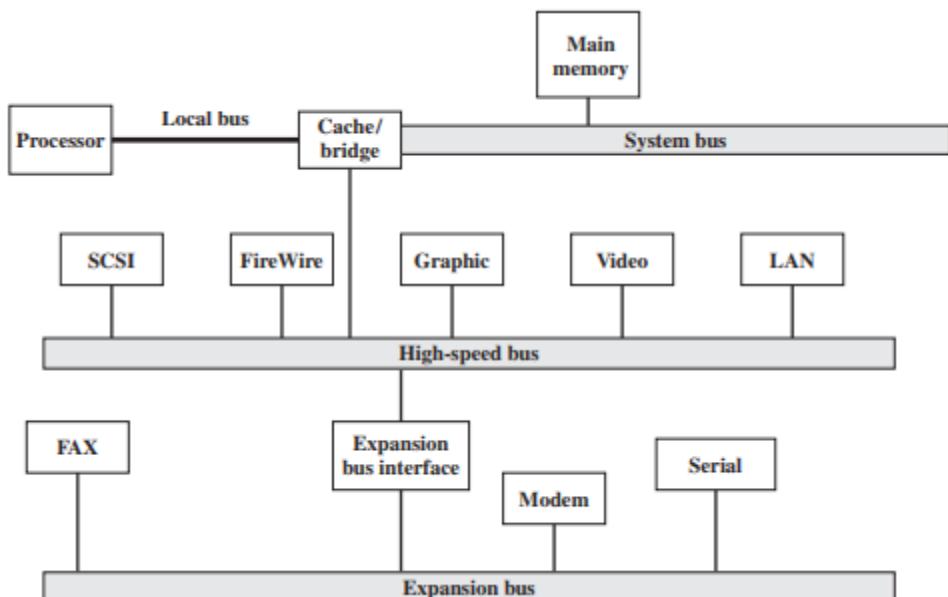
Arsitektur bus tradisional ini cukup efisien tetapi mulai rusak karena kinerja yang lebih tinggi dan lebih tinggi terlihat pada perangkat I/O. Menanggapi permintaan yang berkembang ini, pendekatan umum yang diambil oleh industri adalah membangun bus berkecepatan tinggi yang terintegrasi erat dengan seluruh sistem, hanya membutuhkan jembatan antara bus prosesor dan bus berkecepatan tinggi. Susunan ini kadang-kadang dikenal sebagai arsitektur mezzanine.

Gambar 5.4b menunjukkan realisasi khas dari pendekatan ini. Lagi pula, ada bus lokal yang menghubungkan prosesor ke pengontrol *cache*, yang pada gilirannya terhubung ke bus sistem yang mendukung memori utama. Pengontrol *cache* terintegrasi ke jembatan, atau perangkat penyanga, yang terhubung ke bus berkecepatan tinggi. Bus ini mendukung koneksi ke LAN kecepatan tinggi, seperti Fast Ethernet pada 100 Mbps, pengontrol workstation video dan grafik, serta pengontrol antarmuka ke bus periferal lokal, termasuk SCSI dan FireWire. Yang terakhir adalah pengaturan bus berkecepatan tinggi yang dirancang khusus untuk mendukung perangkat I/O berkapasitas tinggi. Perangkat berkecepatan rendah masih didukung oleh bus ekspansi, dengan lalu lintas penghubung antarmuka antara bus ekspansi dan bus berkecepatan tinggi.

Keuntungan dari pengaturan ini adalah bus berkecepatan tinggi membawa perangkat dengan permintaan tinggi ke dalam integrasi yang lebih dekat dengan prosesor dan pada saat yang sama tidak tergantung pada prosesor. Dengan demikian, perbedaan dalam prosesor dan kecepatan bus berkecepatan tinggi dan definisi garis sinyal ditoleransi. Perubahan arsitektur prosesor tidak memengaruhi bus berkecepatan tinggi, dan sebaliknya.



(a) Arsitektur bus tradisional



(b) Arsitektur bus dengan performa tinggi

Sumber: (Stallings, 2016)

Contoh Konfigurasi Bus

C. Interkoneksi Point-to-Point

Arsitektur bus bersama (shared bus) adalah pendekatan standar untuk interkoneksi antara prosesor dan komponen lain (memori dan I/O) selama beberapa dekade. Tetapi sistem kontemporer semakin bergantung pada interkoneksi point-to-point daripada sharde bus. Alasan utama yang mendorong perubahan dari bus ke interkoneksi point-to-point adalah kendala listrik yang dihadapi dengan peningkatan frekuensi dari lebar bus sinkron. Pada kecepatan data yang semakin tinggi, semakin sulit untuk melakukan fungsi sinkronisasi dan arbitrase secara tepat waktu.

Dengan munculnya chip multicore, dengan banyak prosesor dan memori yang signifikan pada satu chip, ditemukan bahwa penggunaan sharde bus konvensional pada chip yang sama memperbesar kesulitan dalam meningkatkan kecepatan data bus dan mengurangi latensi bus untuk mengimbangi dengan prosesor. Dibandingkan dengan sharde bus, interkoneksi point-to-point memiliki latensi yang lebih rendah, kecepatan data yang lebih tinggi, dan skalabilitas yang lebih baik.

Intel QuickPath Interconnect (QPI) yang diperkenalkan pada tahun 2008, merupakan contoh penting dan representatif dari pendekatan interkoneksi point-to-point.

Berikut ini adalah karakteristik signifikan QPI dan skema interkoneksi point-to-point lainnya:

- Beberapa koneksi langsung: Beberapa komponen dalam sistem menikmati koneksi berpasangan langsung ke komponen lain. Ini menghilangkan kebutuhan untuk arbitrase yang ditemukan dalam sistem transmisi bersama.
- Arsitektur protokol berlapis: Seperti yang ditemukan di lingkungan jaringan, seperti jaringan data berbasis TCP/IP, interkoneksi tingkat prosesor ini menggunakan arsitektur protokol berlapis, daripada penggunaan sederhana sinyal kontrol yang ditemukan dalam pengaturan *sharde bus*.
- Transfer data dalam paket: Data tidak dikirim sebagai aliran bit mentah. Sebaliknya, data dikirim sebagai urutan paket, yang masing-masing menyertakan *control header* dan kode kontrol kesalahan.
-

5.3. Elemen Desain Bus.

Meskipun ada beragam implementasi bus yang berbeda, ada beberapa parameter dasar atau elemen desain yang berfungsi untuk mengklasifikasikan dan membedakan bus (Tabel 5.1)

Type Bus

Jalur bus dapat dipisahkan menjadi dua jenis umum: dedicated dan multiplexed. Dedicated bus ditugaskan secara permanen untuk satu fungsi atau ke subset fisik komponen komputer. Contoh dedikasi fungsional adalah penggunaan alamat khusus dan jalur data terpisah, yang umum di banyak bus. Namun, itu tidak penting. Misalnya, informasi alamat dan data dapat dikirimkan melalui rangkaian jalur yang sama menggunakan jalur kontrol Alamat Valid. Di awal transfer data, alamat ditempatkan di bus dan jalur Alamat Valid diaktifkan. Pada titik ini, setiap modul memiliki periode waktu tertentu untuk menyalin alamat dan menentukan apakah itu adalah modul yang dituju. Alamat tersebut kemudian dihapus dari bus, dan koneksi bus yang sama digunakan untuk transfer data baca atau tulis berikutnya. Metode menggunakan garis yang sama untuk berbagai tujuan ini dikenal sebagai multiplexing waktu.

Tabel 5.1. Elemen-elemen Desain Bus

Type	Bus Width
Dedicated	Address
Multiplexed	Data
Methode of Arbitration	Data Transfer Type
Centralized	Read
Distributed	Write
	Read-modify-write
Timing	Read-after-write
Synchronous	Block
Asynchronous	

Sumber: (Stallings, 2016)

Keuntungan dari multiplexing waktu adalah penggunaan garis yang lebih sedikit, yang menghemat ruang dan, biasanya, biaya. Kerugiannya adalah dibutuhkan sirkuit yang lebih kompleks dalam setiap modul. Juga, ada potensi penurunan kinerja karena peristiwa-peristiwa tertentu yang memiliki garis yang sama tidak dapat terjadi secara paralel.

Dedikasi fisik mengacu pada penggunaan beberapa bus, yang masing-masing hanya menghubungkan sebagian modul. Contoh tipikal adalah penggunaan bus I/O untuk menghubungkan semua modul I/O; bus ini kemudian dihubungkan ke bus utama melalui beberapa jenis modul adaptor I/O. Keuntungan potensial dari pengabdian fisik adalah throughput yang tinggi, karena pertikaian bus semakin sedikit. Kerugiannya adalah peningkatan ukuran dan biaya sistem.

Metode Arbitrase

Dalam semua sistem kecuali yang paling sederhana, lebih dari satu modul mungkin perlu mengendalikan bus. Misalnya, modul I/O mungkin perlu membaca atau menulis langsung ke memori, tanpa mengirim data ke prosesor. Karena hanya satu unit pada satu waktu yang berhasil mentransmisikan melalui bus, beberapa metode arbitrase diperlukan.

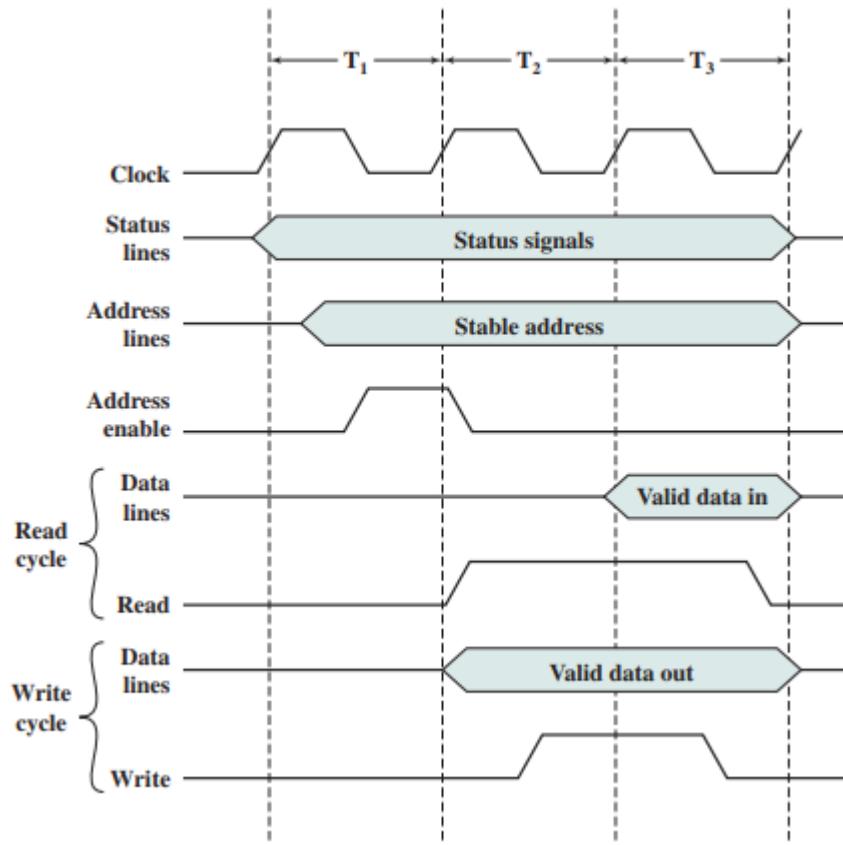
Berbagai metode dapat secara kasar diklasifikasikan sebagai arbitrasi terpusat atau arbitrasi terdistribusi. Dalam skema terpusat, perangkat perangkat keras tunggal, yang disebut sebagai pengendali bus atau arbiter, bertanggung jawab untuk mengalokasikan waktu di bus. Perangkat dapat berupa modul atau bagian terpisah dari prosesor. Dalam skema terdistribusi, tidak ada pengendali pusat. Sebaliknya, setiap modul berisi logika kontrol akses dan modul-modul tersebut bekerja bersama untuk berbagi bus. Dengan kedua metode arbitrase, tujuannya adalah untuk menunjuk satu perangkat, baik prosesor atau modul I/O, sebagai master. Master kemudian dapat melakukan transfer data (misalnya Membaca atau menulis) dengan beberapa perangkat lain, yang bertindak sebagai budak untuk pertukaran khusus ini.

Timing

Timing mengacu pada cara di mana kejadian (*event*) dikoordinasikan di bus. Bus dapat menggunakan *synchronous* maupun *asynchronous timing*. Dengan *synchronous timing*, terjadinya *event* di bus ditentukan oleh *clock*. *Clock* mentransmisikan urutan sinyal 1 dan 0 dengan durasi yang sama secara bergantian. Transmisi sebuah 1-0 disebut sebagai *clock cycle* atau *bus cycle* dan menentukan slot waktu. Semua perangkat lain di bus dapat membaca *Clock line*, dan semua *event* dimulai pada awal *clock cycle*.

Operasi Bus Sinkron.

Gambar 5.5 menunjukkan *timing diagram* yang khas, tetapi disederhanakan, untuk operasi baca dan tulis yang sinkron. Sinyal bus lain dapat berubah pada *leading edge* sinyal *clock* (dengan sedikit delay). Sebagian besar *event* menempati satu siklus *clock* tunggal. Dalam contoh sederhana ini, prosesor menempatkan alamat memori pada baris alamat selama siklus *clock* pertama dan dapat menegaskan berbagai baris status. Setelah garis alamat stabil, prosesor mengeluarkan alamat yang memungkinkan sinyal. Untuk operasi baca, prosesor mengeluarkan perintah baca pada awal siklus kedua. Modul memori mengenali alamat dan, setelah penundaan satu siklus, menempatkan data pada baris data. Prosesor membaca data dari saluran data dan menjatuhkan sinyal baca. Untuk operasi tulis, prosesor meletakkan data pada garis data pada awal siklus kedua dan mengeluarkan perintah tulis setelah garis data stabil. Modul memori menyalin informasi dari jalur data selama siklus *clock-3*.

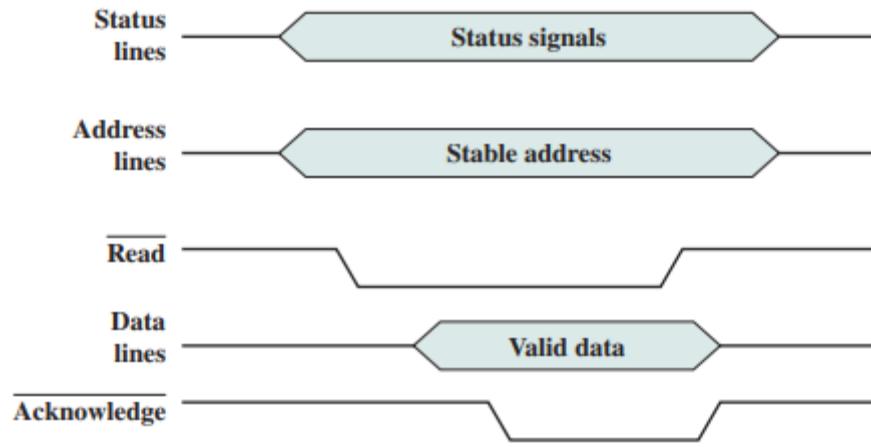


Sumber: (Stallings, 2016)

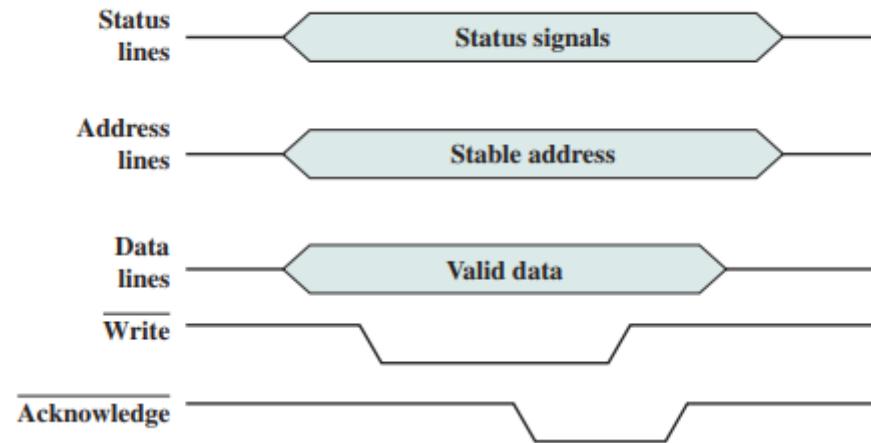
Gambar 5.3. Diagram Pewaktu Operasi Bus Sinkron

Operasi Bus Asinkron

Dengan pengaturan waktu asinkron, terjadinya satu peristiwa di bus akan mengikuti dan bergantung pada terjadinya peristiwa sebelumnya. Dalam contoh bacaan sederhana pada Gambar 5.6a, prosesor menempatkan sinyal alamat dan status pada bus. Setelah berhenti sejenak agar sinyal-sinyal ini stabil, ia mengeluarkan perintah baca, yang menunjukkan adanya alamat yang valid dan sinyal kontrol. Memori yang sesuai menerjemahkan alamat dan merespons dengan menempatkan data pada jalur data. Setelah jalur data stabil, modul memori menegaskan jalur yang diakui untuk memberi sinyal kepada prosesor bahwa data tersedia. Setelah master membaca data dari jalur data, ia membatalkan sinyal baca. Ini menyebabkan modul memori melepaskan data dan mengakui baris. Akhirnya, setelah jalur *acknowledge* dilepaskan, master menghapus informasi alamat.



(a) Timing Untuk Siklus Baca



(b) Timing Untuk Siklus Tulis

Sumber: (Stallings, 2016)

Gambar 5.4. Diagram Pewaktu Operasi Bus Asinkron

Gambar 5.6b menunjukkan operasi tulis asinkron sederhana. Dalam hal ini, master menempatkan data pada jalur data pada saat yang sama menempatkan sinyal pada jalur status dan alamat. Modul memori menanggapi perintah tulis dengan menyalin data dari jalur data dan kemudian menegaskan jalur yang diakui. Master kemudian melepas sinyal tulis dan modul memori melepas sinyal *acknowledge*.

Pengaturan waktu sinkron lebih sederhana untuk diterapkan dan diuji. Namun, ini kurang fleksibel dibandingkan pengaturan waktu asinkron. Karena semua perangkat pada bus sinkron terikat ke laju jam tetap, sistem tidak dapat memanfaatkan kemajuan kinerja perangkat. Dengan pengaturan waktu asinkron, campuran perangkat lambat dan cepat, menggunakan teknologi lama dan lebih baru, dapat berbagi bus.

D. Dedicated dan Multiplexed

Jalur bus dapat dipisahkan menjadi dua jenis umum: dedicated dan multiplexed. Jalur bus khusus ditugaskan secara permanen untuk satu fungsi atau ke subset fisik komponen komputer. Contoh dedikasi fungsional adalah penggunaan alamat khusus dan jalur data terpisah, yang umum di banyak bus. Misalnya, informasi alamat dan data dapat dikirim melalui rangkaian jalur yang sama menggunakan jalur kontrol Valid Address. Di awal transfer data, alamat ditempatkan di bus dan jalur Valid Address diaktifkan. Pada titik ini, setiap modul memiliki periode waktu tertentu untuk menyalin

alamat dan menentukan apakah itu adalah modul yang dituju. Alamat tersebut kemudian dihapus dari bus, dan koneksi bus yang sama digunakan untuk transfer data baca atau tulis berikutnya. Metode menggunakan jalur yang sama untuk berbagai tujuan ini dikenal sebagai time multiplexing.

Keuntungan dari time multiplexing adalah penggunaan jalur yang lebih sedikit, yang menghemat ruang dan biaya. Kekurangannya adalah dibutuhkan sirkuit yang lebih kompleks dalam setiap modul. Juga, ada potensi penurunan kinerja karena peristiwa-peristiwa tertentu yang memiliki jalur yang sama tidak dapat terjadi secara paralel.

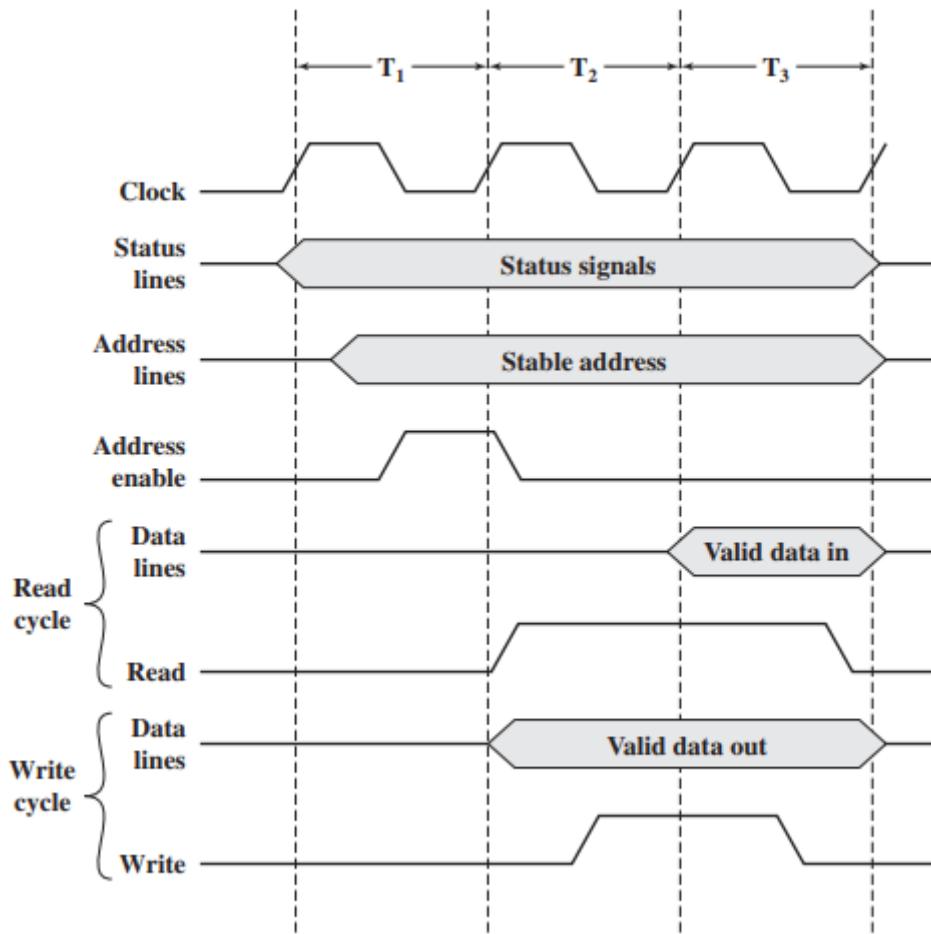
Dedikasi fisik mengacu pada penggunaan beberapa bus, yang masing-masing hanya menghubungkan sebagian modul. Contoh tipikal adalah penggunaan bus I/O untuk menghubungkan semua modul I/O, bus ini kemudian dihubungkan ke bus utama melalui beberapa jenis modul adaptor I/O. Keuntungan potensial dari dedikasi fisik adalah throughput yang tinggi. Kekurangannya adalah peningkatan ukuran dan biaya sistem.

E. Metode Arbitrase

Dalam sistem, lebih dari satu modul mungkin perlu mengendalikan bus. Misalnya, modul I/O mungkin perlu membaca atau menulis langsung ke memori, tanpa mengirim data ke prosesor. Karena hanya satu unit pada satu waktu yang berhasil mentransmisikan melalui bus, diperlukan beberapa metode arbitrase. Berbagai metode dapat secara kasar diklasifikasikan sebagai tersentralisasi atau terdistribusi. Dalam skema terpusat, perangkat perangkat keras tunggal, yang disebut sebagai pengendali bus atau arbiter, bertanggung jawab untuk mengalokasikan waktu di bus. Perangkat dapat berupa modul atau bagian terpisah dari prosesor. Dalam skema terdistribusi, tidak ada pengendali pusat. Sebaliknya, setiap modul berisi logika kontrol akses dan modul-modul tersebut bekerja bersama untuk berbagi bus. Dengan kedua metode arbitrase, tujuannya adalah untuk menunjuk satu perangkat, baik prosesor atau modul I/O, sebagai master. Master kemudian dapat melakukan transfer data (misalnya read atau write) dengan beberapa perangkat lain, yang bertindak sebagai slave untuk pertukaran khusus ini.

F. Timing

Pengaturan *timing* mengacu pada cara di mana acara dikoordinasikan di dalam bus. Bus menggunakan *timing* sinkron. Dengan *timing* yang sinkron, terjadinya peristiwa di bus ditentukan oleh *clock*. Bus termasuk time line di mana *clock* mentransmisikan urutan reguler bergantian 1s dan 0s dengan durasi yang sama. Transmisi 1-0 tunggal disebut sebagai siklus *clock* atau siklus bus dan menentukan slot waktu. Semua perangkat lain di bus dapat membaca time line, dan semua acara dimulai pada awal siklus *clock*. Gambar 5.7 menunjukkan diagram timing yang disederhanakan, untuk operasi baca dan tulis yang sinkron. Sinyal bus lain dapat berubah di ujung depan sinyal *clock* (dengan sedikit *delay*). Sebagian besar peristiwa menempati satu siklus *clock* tunggal. Dalam contoh sederhana ini, prosesor menempatkan alamat memori pada jalur alamat selama siklus *clock* pertama dan dapat menegaskan berbagai baris status. Setelah garis alamat stabil, prosesor mengeluarkan sinyal alamat yang valid. Untuk operasi baca, prosesor mengeluarkan perintah baca pada awal siklus kedua. Modul memori mengenali alamat dan, setelah penundaan satu siklus, menempatkan data pada baris data. Prosesor membaca data dari saluran data dan mengeluarkan sinyal baca. Untuk operasi tulis, prosesor menempatkan data pada jalur data pada awal siklus kedua, dan mengeluarkan perintah tulis setelah jalur data stabil. Modul memori menyalin informasi dari jalur data selama siklus *clock*-3



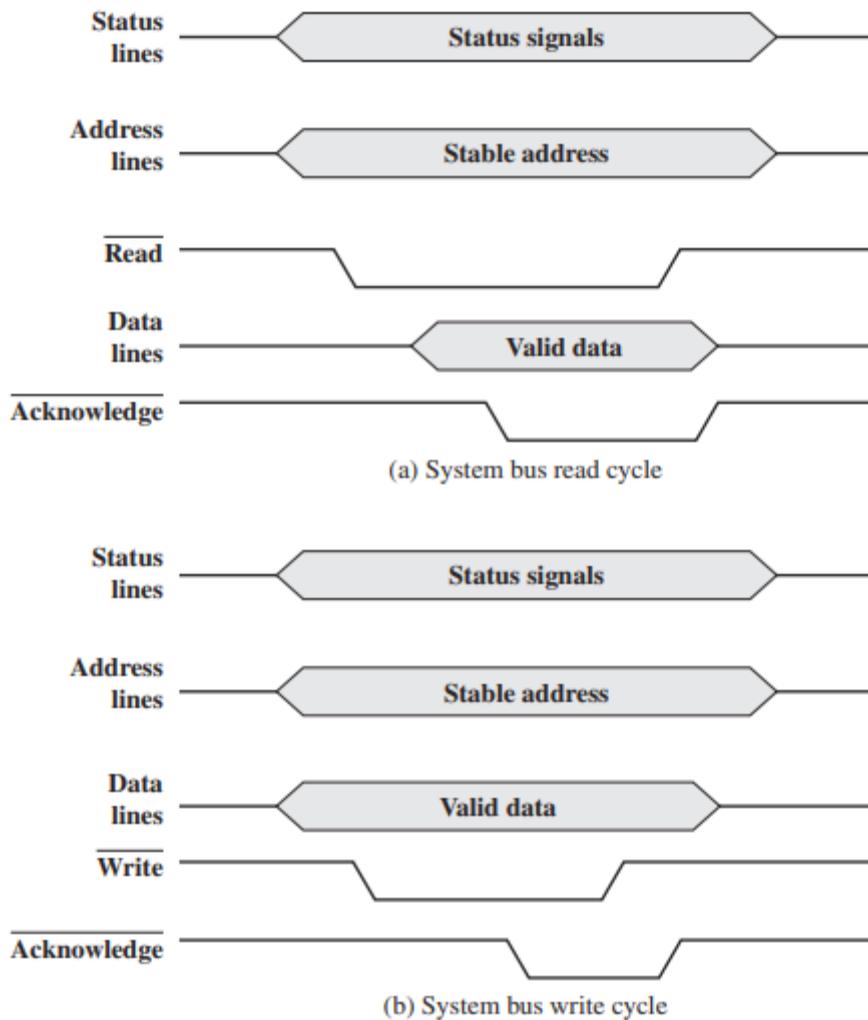
Sumber: (Stallings, 2016)

Gambar 5.5. Pewaktu operasi Synchronous Bus

Dengan pengaturan waktu yang tidak sinkron, kemunculan satu kejadian pada bus mengikuti dan tergantung pada kemunculan kejadian sebelumnya. Dalam contoh sederhana pada Gambar 5.8a, prosesor menempatkan sinyal alamat dan status di bus. Setelah berhenti untuk menstabilkan sinyal-sinyal ini, itu mengeluarkan perintah baca, menunjukkan keberadaan alamat yang valid dan sinyal kontrol. Memori yang sesuai menerjemahkan alamat dan merespons dengan menempatkan data pada jalur data. Setelah jalur data stabil, modul memori menegaskan jalur yang diakui untuk memberi sinyal prosesor bahwa data tersedia. Setelah master membaca data dari jalur data, master akan menghilangkan sinyal baca. Ini menyebabkan modul memori menjatuhkan data dan mengenali jalur. Akhirnya, setelah sinyal acknowledge valid, master menghapus informasi alamat.

Gambar 5.8b menunjukkan operasi tulis asinkron yang sederhana. Dalam hal ini, master menempatkan data pada jalur data pada saat yang sama dengan meletakkan sinyal pada status dan jalur alamat. Modul memori merespons perintah tulis dengan menyalin data dari jalur data dan kemudian menegaskan jalur yang diakui. Master kemudian menjatuhkan sinyal tulis dan modul memori menjatuhkan sinyal terima.

Waktu sinkron lebih mudah diterapkan dan diuji. Namun, itu kurang fleksibel daripada waktu asinkron. Karena semua perangkat pada bus sinkron terikat pada laju clock tetap, sistem tidak dapat memanfaatkan kemajuan dalam kinerja perangkat. Dengan pengaturan waktu yang tidak sinkron, campuran perangkat yang lambat dan cepat, menggunakan teknologi yang lebih lama dan lebih baru, dapat berbagi bus.



Sumber: (Stallings, 2016)

Gambar 5.6. Pewaktu operasi Asynchronous Bus

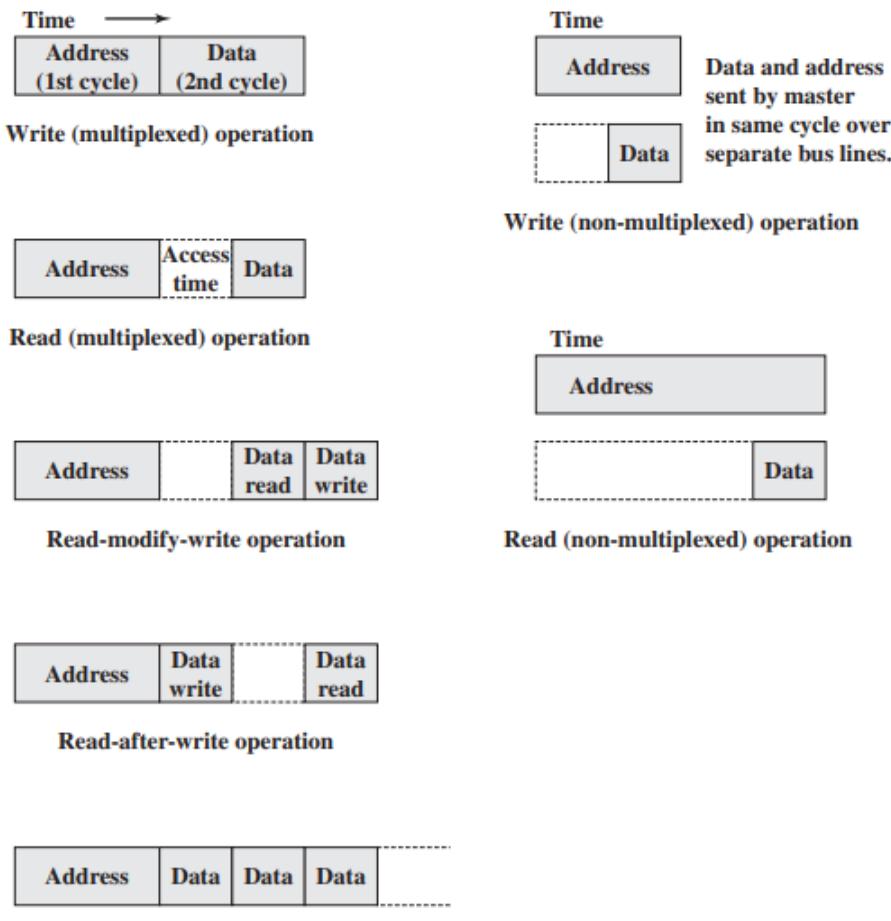
G. Bus Width

Lebar bus data berdampak pada kinerja sistem: Semakin lebar bus data, semakin besar jumlah bit yang ditransfer sekaligus. Pada umumnya lebar bus data adalah 8 bit atau kelipatannya. Lebar bus alamat berdampak pada kapasitas sistem: semakin besar bus alamat, semakin besar jangkauan lokasi yang bisa direferensikan. Bus alamat 10 bit dapat menjangkau lokasi sebanyak $2^{10} = 1K$ lokasi. Setiap lokasi memiliki alamat yang unik, misalnya pada sistem dengan lebar bus alamat 10 bit, maka alamat memori yang dapat dijangkau adalah mulai dari 000H sampai 3FFH.

H. Jenis Transfer Data

Sebuah bus pada umumnya dapat mendukung berbagai jenis transfer data, seperti yang diilustrasikan dalam Gambar 5.99. Semua bus mendukung transfer tulis (master ke slave) dan baca (slave to master). Dalam kasus bus alamat/data multipleks, bus pertama kali digunakan untuk menentukan alamat dan kemudian untuk mentransfer data. Baik untuk membaca atau menulis, mungkin juga ada penundaan jika perlu melalui arbitrase untuk mendapatkan kendali atas bus selama sisa operasi (yaitu, merebut bus untuk meminta baca atau tulis, lalu ambil bus lagi untuk melakukan baca atau tulis). Dalam hal bus dedicated, alamat diletakkan di bus alamat dan tetap di sana sementara data diletakkan di bus data. Untuk operasi penulisan, master meletakkan data ke bus data segera setelah alamat telah stabil

dan slave memiliki kesempatan untuk mengenali alamatnya. Untuk operasi baca, slave meletakkan data ke bus data segera setelah ia mengenali alamatnya dan telah mengambil data.



Sumber: (Stallings, 2016)

Gambar 5.7. Type Transfer Bus Data

Ada juga beberapa operasi kombinasi yang diizinkan oleh beberapa bus. Operasi baca-modifikasi-tulis hanyalah pembacaan yang diikuti segera oleh penulisan ke alamat yang sama. Alamat ini hanya disiarkan sekali pada awal operasi. Seluruh operasi biasanya tidak dapat dipisahkan untuk mencegah akses ke elemen data oleh master bus potensial lainnya, tujuannya adalah untuk melindungi sumber daya memori bersama dalam sistem multiprogramming.

Baca-setelah-tulis adalah operasi yang tidak dapat dibagi yang terdiri dari penulisan yang segera diikuti oleh pembacaan dari alamat yang sama. Operasi baca dapat dilakukan untuk tujuan pemeriksaan. Beberapa sistem bus juga mendukung transfer data blok. Dalam hal ini, satu siklus alamat diikuti oleh n siklus data. Item data pertama ditransfer ke atau dari alamat yang ditentukan; item data yang tersisa ditransfer ke atau dari alamat selanjutnya.

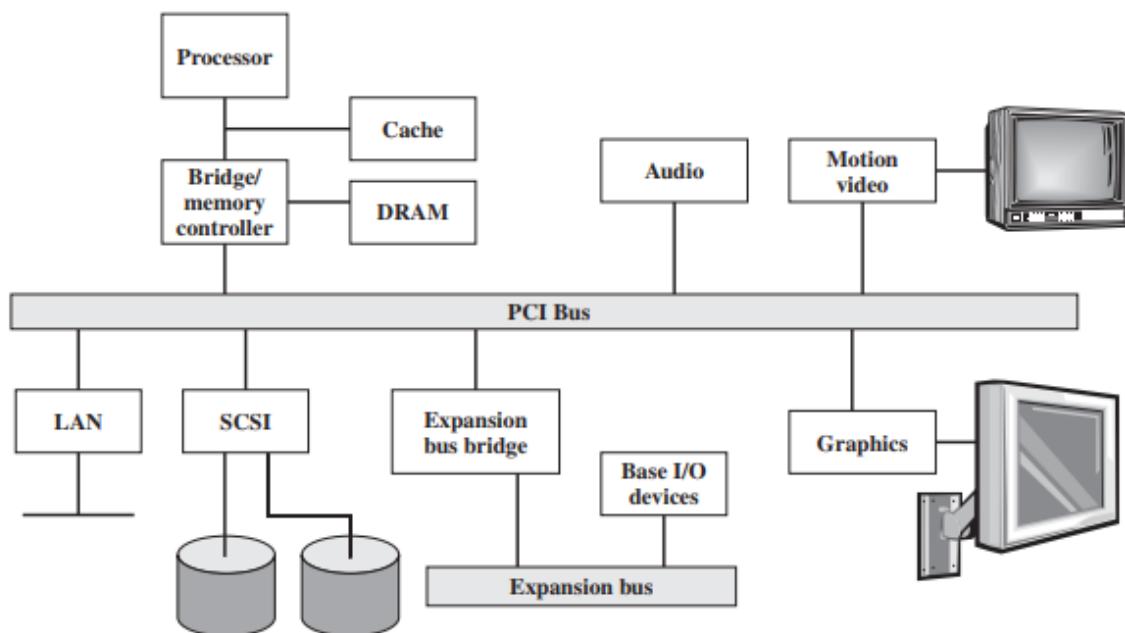
5.4. PCI

Interconnect Component Peripheral (PCI) adalah bus populer dengan bandwidth tinggi, prosesor-independen yang dapat berfungsi sebagai mezzanine atau bus periferal. Dibandingkan dengan spesifikasi bus umum lainnya, PCI memberikan kinerja sistem yang lebih baik untuk subsistem I/O berkecepatan tinggi (misalnya: *graphic display adapter*, *network interface controller*, pengontrol disk,

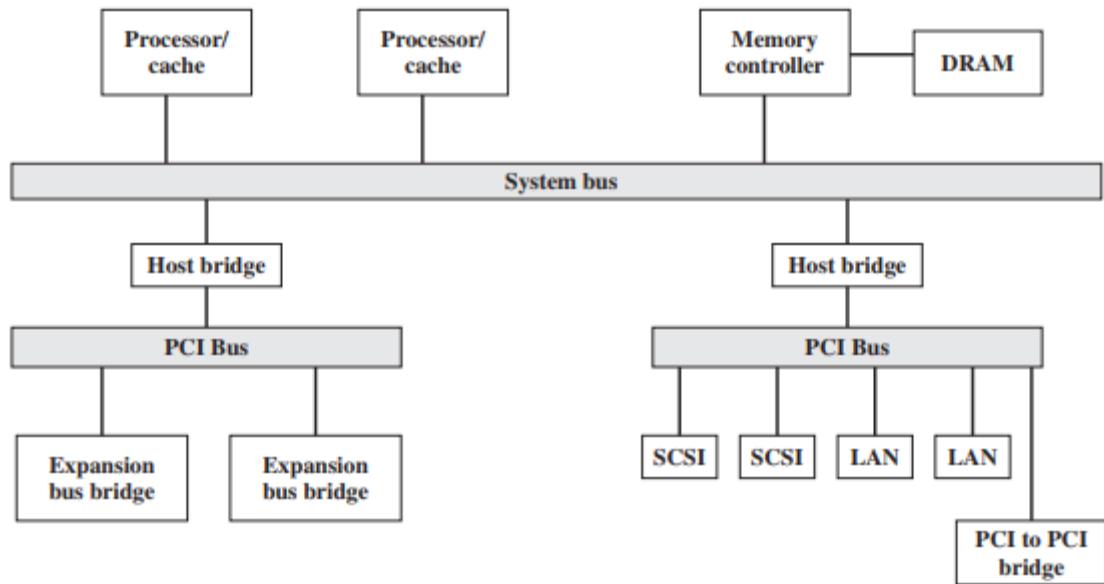
dan sebagainya). PCI memungkinkan penggunaan hingga 64 jalur data pada 66 MHz, untuk kecepatan raw transfer 528 MByte/dtk, atau 4,224 Gbps. Intel mulai bekerja pada PCI pada tahun 1990 untuk sistem berbasis Pentium. Intel segera merilis semua paten ke domain publik dan mempromosikan pembentukan asosiasi industri, PCI Special Interest Group (SIG), untuk mengembangkan lebih lanjut dan menjaga kompatibilitas spesifikasi PCI. Hasilnya adalah PCI telah diadopsi secara luas dan semakin banyak digunakan di komputer pribadi, workstation, dan server. Karena spesifikasi berada dalam domain publik dan didukung oleh berbagai industri mikroprosesor dan periferal, sehingga produk PCI yang dibuat oleh vendor yang berbeda menjadi kompatibel untuk semua sistem.

PCI dirancang untuk mendukung berbagai konfigurasi berbasis mikroprosesor, termasuk sistem prosesor tunggal maupun *multiple processor*. Karena itu, PCI menyediakan serangkaian fungsi general purpose, menggunakan timing sinkron dan skema arbitrase terpusat.

Gambar 5.10a menunjukkan penggunaan PCI pada sistem prosesor tunggal. Pengontrol DRAM dan jembatan gabungan ke bus PCI menyediakan koneksi yang erat dengan prosesor dan kemampuan untuk mengirimkan data pada kecepatan tinggi. Bridge bertindak sebagai buffer data sehingga kecepatan bus PCI dapat berbeda dari kemampuan I/O prosesor. Dalam sistem multiprosesor (Gambar 5.10b), satu atau lebih konfigurasi PCI dapat dihubungkan oleh jembatan ke bus sistem prosesor. Bus sistem hanya mendukung unit prosesor/cache, memori utama, dan jembatan PCI. Sekali lagi, penggunaan bridge menjaga PCI independen dari kecepatan prosesor namun memberikan kemampuan untuk menerima dan mengirimkan data dengan cepat



(a) Konfigurasi PCI Pada Personal Computer



(b) Typical server system

(b) Konfigurasi PCI Pada Sistem Server

Sumber: (Stallings, 2016)

Gambar 5.8. Contoh Konfigurasi PCI

A. Struktur Bus PCI

PCI dapat dikonfigurasi sebagai bus 32 atau 64-bit. Tabel 5.2 mendefinisikan 49 jalur sinyal wajib untuk PCI. Ini dibagi menjadi beberapa kelompok fungsional berikut:

- *System pins*: Termasuk *clock* dan *reset pin*.
- Pin alamat dan data: Sertakan 32 baris yang multipleks waktu untuk alamat dan data. Baris lain dalam grup ini digunakan untuk menafsirkan dan memvalidasi garis sinyal yang membawa alamat dan data.
- *Address and Data Pins*: Mengontrol waktu transaksi dan menyediakan koordinasi di antara penggagas dan target.
- *Interface Control Pins*: Mengontrol peripheral untuk melakukan baca dan
- Pin arbitrase: Tidak seperti jalur sinyal PCI lainnya, ini bukan saluran yang dipakai bersama. Sebaliknya, setiap master PCI memiliki sepasang jalur arbitrase sendiri yang menghubungkannya langsung ke PCI bus arbiter.
- Pin pelaporan kesalahan: Digunakan untuk melaporkan paritas dan kesalahan lainnya

Tabel 5.2. Sinyal Wajib (Mandatory) PCI

Pin	Type	Deskripsi
System Pins		
CLK	in	Memberikan timing untuk semua transaksi dan diambil sampelnya oleh semua input di tepi menaik (<i>raising edge</i>). Kecepatan <i>clock</i> hingga 33 MHz didukung.
RST#	in	Memaksa semua register, sequencer, dan sinyal khusus PCI ke keadaan yang diinisialisasi.
Address and Data Pins		
AD[31::0]	t/s	Jalur multiplexed digunakan untuk alamat dan data
C/BE[3::0]#	t/s	Perintah bus multipleks dan sinyal pengaktifan <i>byte</i> . Selama fase data, jalur tersebut menunjukkan yang mana dari empat jalur <i>byte</i> yang membawa data yang berarti.

PAR	t/s	Parity. Memberikan pariti ganjil di seluruh jalur AD dan C/BE satu siklus <i>clock</i> kemudian. Master menggerakkan PAR untuk mengalamatkan dan menulis fase data; PAR target drive untuk membaca fase data
<i>Interface Control Pins</i>		
FRAME#	s/t/s	Dikendalikan oleh master saat ini untuk menunjukkan awal dan durasi transaksi. Ini ditegaskan di awal dan dibatalkan saat inisiator siap untuk memulai fase data akhir.
IRDY#	s/t/s	<i>Initiator Ready</i> . Dikendalikan oleh master bus saat ini (inisiator transaksi). Saat membaca, menunjukkan bahwa master siap menerima data; selama penulisan, menunjukkan bahwa data yang valid ada di AD.
TRDY#	s/t/s	<i>Target Ready</i> . Dikendalikan oleh target (perangkat yang dipilih). Selama pembacaan, menunjukkan bahwa data yang valid ada di AD; selama menulis, menunjukkan bahwa target siap menerima data.
STOP#	s/t/s	Menunjukkan bahwa target saat ini menginginkan inisiator untuk menghentikan transaksi saat ini
IDSEL	in	<i>Initialization Device Select</i> . Digunakan sebagai chip yang dipilih selama konfigurasi baca dan tulis transaksi.
DEVSEL#	in	<i>Device Select</i> . Ditegaskan oleh target ketika telah mengenali alamatnya. Menunjukkan kepada inisiator saat ini apakah ada perangkat yang telah dipilih.
<i>Arbitration Pins</i>		
REQ#	t/s	Menunjukkan kepada arbiter bahwa perangkat ini memerlukan penggunaan bus. Ini adalah jalur <i>point-to-point</i> khusus perangkat.
GN#	t/s	Menunjukkan ke perangkat bahwa wasit telah memberikan akses bus. Ini adalah jalur <i>point-to-point</i> khusus perangkat.
<i>Error Reporting Pins</i>		
PERR#	s/t/s	<i>Parity Error</i> . Menunjukkan kesalahan paritas data terdeteksi oleh target selama fase data tulis atau oleh pemrakarsa selama fase data baca.
SERR#	o/d	<i>System Error</i> . Dapat digerakkan oleh perangkat apa pun untuk melaporkan kesalahan paritas alamat dan kesalahan kritis selain paritas.

in: sinyal Input-only

out: sinyal Output-only

t/s: Bidirectional, tri-state, sinyal I/O

s/t/s (Sinyal tri-state): yang dipertahankan hanya

digerakkan oleh satu pemilik pada satu waktu

o/d (open drain): memungkinkan beberapa perangkat

berbagi sebagai wire-OR# Status aktif sinyal terjadi pada

tegangan rendah

Sumber: (Stallings, 2016)

Selain itu, spesifikasi PCI mendefinisikan 51 jalur sinyal opsional (Tabel 5.3), yang dibagi menjadi beberapa kelompok fungsional berikut:

- 1) *Interrupt pins*: Ini disediakan untuk perangkat PCI yang harus menghasilkan permintaan layanan. Seperti halnya pin arbitrase, ini bukan baris yang dibagikan. Sebaliknya, setiap perangkat PCI memiliki jalur interupsi sendiri atau jalur ke pengendali interupsi.
- 2) *Cache upport pins*: Pin ini diperlukan untuk mendukung memori pada PCI yang dapat di-*cache* di prosesor atau perangkat lain.
- 3) Pin ekstensi bus 64-bit: Sertakan 32 baris yang multipleks waktu untuk alamat dan data dan yang dikombinasikan dengan alamat wajib/jalur data untuk membentuk bus alamat/data 64-bit. Jalur lain dalam grup ini digunakan untuk menafsirkan dan memvalidasi garis sinyal yang membawa alamat dan data. Terakhir, ada dua jalur yang memungkinkan dua perangkat PCI menyetujui penggunaan kemampuan 64-bit.
- 4) JTAG/boundary scan pins: Jalur sinyal ini mendukung prosedur pengujian yang didefinisikan dalam IEEE Standard 1149.1.

Tabel 5.3. Sinyal Optional pada PCI

Pin	Type	Deskripsi
Interrupt Pins		
INTA#	o/d	Digunakan untuk meminta interupsi.
INTB#	o/d	Digunakan untuk meminta interupsi; hanya memiliki arti pada perangkat multifungsi.
INTC#	o/d	Digunakan untuk meminta interupsi; hanya memiliki arti pada perangkat multifungsi.
INTD#	o/d	Digunakan untuk meminta interupsi; hanya memiliki arti pada perangkat multifungsi.
Cache support pins		
SBO#	in/out	Snoop Backoff. Menunjukkan hit ke baris yang dimodifikasi.
SDONE	in/out	Snoop DOne. Menunjukkan status pengintai untuk akses saat ini. Ditegaskan saat pengintaian telah selesai.
64 Bits Bus Extension Pins		
AD[63::32]	t/s	Jalur multipleks digunakan untuk alamat dan data untuk memperluas bus ke 64 bit
C/BE[7::4]#	t/s	Perintah bus multipleks dan sinyal pengaktifan byte. Selama fase alamat, baris menyediakan perintah bus tambahan. Selama fase data, garis-garis tersebut menunjukkan yang mana dari empat jalur byte tambahan yang membawa data yang berarti.
REQ64#	s/t/s	Digunakan untuk meminta transfer 64-bit.
ACK64#	s/t/s	Mengindikasikan target bersedia melakukan transfer 64-bit.
PAR64	t/s	Memberikan paritas yang merata di seluruh garis AD dan C/BE yang diperpanjang satu siklus jam kemudian.
JTAG/Boundary Scan Pins		
TCK	in	Clock Test. Digunakan untuk mencatat informasi status dan menguji data ke dalam dan ke luar perangkat selama pemindaian batas.
TDI	in	Test input. Digunakan untuk menggeser data uji dan instruksi secara serial ke dalam perangkat.
TDO	out	Test output. Digunakan untuk menggeser data uji dan instruksi secara serial keluar dari perangkat.
TMS	in	Test mode Select. Digunakan untuk mengontrol status pengontrol port akses uji
TRST#	in	Test reset. Digunakan untuk menginisialisasi pengontrol port akses uji.

Sumber: (Stallings, 2016)

B. Perintah PCI

Aktivitas bus terjadi dalam bentuk transaksi antara inisiator, atau master, dan target. Ketika master bus memperoleh kendali atas bus, bus menentukan jenis transaksi yang akan terjadi selanjutnya. Selama fase alamat transaksi, garis C/BE digunakan untuk memberi sinyal jenis transaksi. Perintahnya adalah sebagai berikut:

- *Interrupt Acknowledge*
- *Special Cycle*
- *I/O Read*
- *I/O Write*
- *Memory Read*
- *Memory Read Line*
- *Memory Read Multiple*
- *Memory Write*
- *Memory Write and Invalidate*
- *Configuration Read*
- *Configuration Write*
- *Dual address Cycle*

Interrupt Acknowledge adalah perintah baca yang ditujukan untuk interrupt controller pada bus PCI. Baris alamat tidak digunakan selama fase alamat, dan jalur *byte* memungkinkan menunjukkan ukuran pengidentifikasi interupsi yang akan dikembalikan. Perintah *Special Cycle* digunakan oleh inisiator untuk menyiarakan pesan ke satu atau lebih target. Perintah Baca/Tulis I/O digunakan untuk mentransfer data antara inisiator dan pengontrol I/O. Setiap perangkat I/O memiliki ruang alamatnya sendiri, dan garis alamat digunakan untuk menunjukkan perangkat tertentu dan untuk menentukan data yang akan ditransfer ke atau dari perangkat itu.

Perintah Read dan Write memori digunakan untuk menetapkan transfer data, menempati satu atau beberapa siklus *clock*. Interpretasi dari perintah-perintah ini tergantung pada apakah *memory controller* pada bus PCI mendukung protokol PCI untuk transfer antara memori dan *cache*. Jika demikian, transfer data ke dan dari memori biasanya dalam hal garis *cache*, atau blok. Tiga perintah membaca memori memiliki penggunaan yang diuraikan dalam Tabel 5.4. Perintah Memory Write digunakan untuk mentransfer data dalam satu atau beberapa siklus data ke memori. Perintah *Memory Write and Invalidate* mentransfer data dalam satu atau beberapa siklus ke memori.

Tabel 5.4. Interpretasi Read Commands pada PCI

<i>Read Command Type</i>	<i>For Cachable Memory</i>	<i>For Noncachable Memory</i>
<i>Memory Read</i>	Melonjak satu setengah atau kurang dari baris <i>cache</i>	Melonjak 2 siklus transfer data atau kurang
<i>Memory Read Line</i>	Melonjak lebih dari satu setengah baris <i>cache</i> ke tiga baris <i>cache</i>	Melonjak 3 hingga 12 transfer data
<i>Memory Read Multiple</i>	Meledak lebih dari tiga baris <i>cache</i>	Melonjak lebih dari 12 transfer data

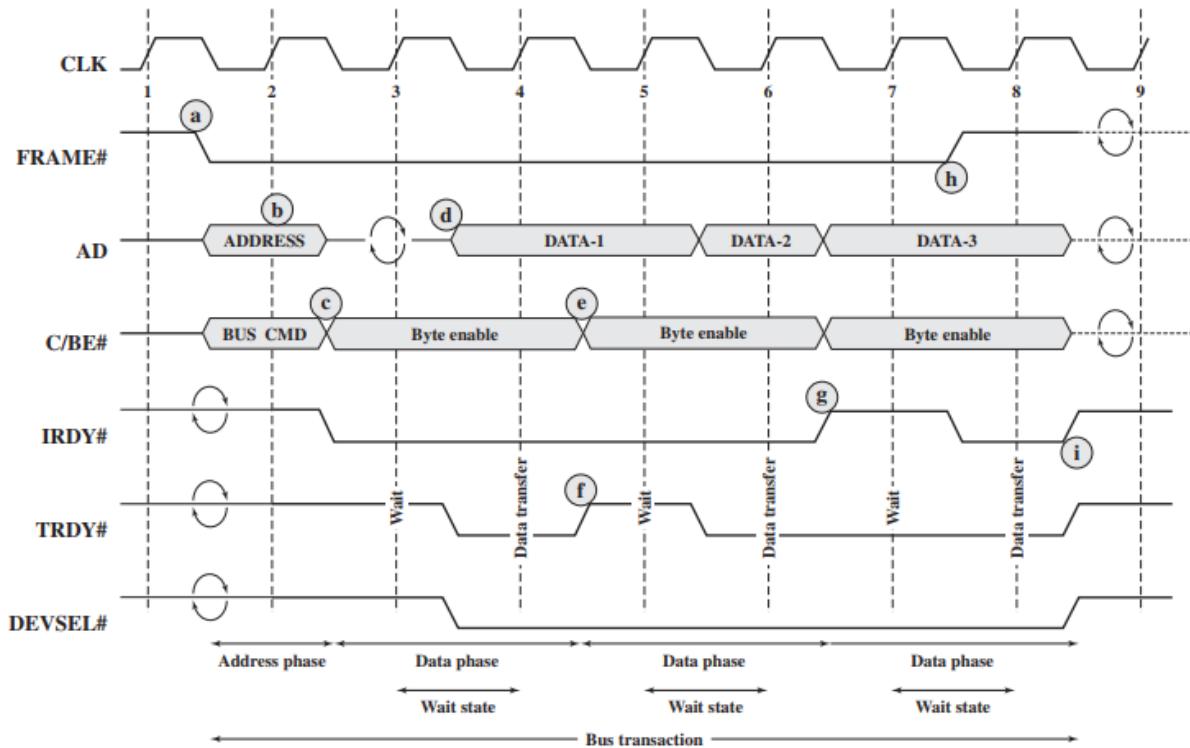
Sumber: (Stallings, 2016)

Selain itu, jaminan bahwa setidaknya satu baris *cache* ditulis. Perintah ini mendukung fungsi *cache* untuk menulis kembali satu baris ke memori. Dua perintah konfigurasi memungkinkan master untuk membaca dan memperbarui parameter konfigurasi di perangkat yang terhubung ke PCI. Setiap perangkat PCI dapat menyertakan hingga 256 register internal yang digunakan selama inisialisasi sistem untuk mengkonfigurasi perangkat itu. Perintah *Dual Address Cycle* digunakan oleh inisiator untuk menunjukkan bahwa ia menggunakan pengalamatan 64-bit.

C. Transfer Data

Setiap transfer data pada bus PCI adalah transaksi tunggal yang terdiri dari satu fase alamat dan satu atau lebih fase data. Gambar 5.8 menunjukkan waktu transaksi baca. Semua acara disinkronkan ke transisi menurun sinyal *clock*, yang terjadi di tengah setiap siklus *clock*. Perangkat bus mengenali jalur bus di tepi menaik sinyal *clock* pada awal siklus bus. Berikut ini adalah peristiwa penting, yang ditandai pada diagram:

- a. Setelah *master* bus mendapatkan kendali atas bus, ia dapat memulai transaksi dengan menyatakan FRAME. Baris ini tetap valid sampai inisiator siap untuk menyelesaikan fase data terakhir. Inisiator juga meletakkan alamat mulai pada bus alamat, dan perintah baca pada baris C/BE.
- b. Pada awal *clock-2*, perangkat target akan mengenali alamatnya pada jalur AD.
- c. Inisiator berhenti mengenali bus AD. Siklus putaran (ditunjukkan oleh dua panah melingkar) diperlukan pada semua saluran sinyal yang mungkin digerakkan oleh lebih dari satu perangkat, sehingga menjatuhkan sinyal alamat akan menyiapkan bus untuk digunakan oleh perangkat target. Pemrakarsa mengubah informasi tentang jalur C/BE untuk menentukan jalur AD mana yang akan digunakan untuk transfer untuk data yang saat ini ditangani (dari 1 hingga 4 byte). Inisiator juga menyatakan IRDY untuk mengindikasikan bahwa item tersebut siap untuk item data pertama.
- d. Target yang dipilih menegaskan DEVSEL untuk menunjukkan bahwa ia telah mengenali alamatnya dan akan merespons. Ini menempatkan data yang diminta pada jalur AD dan menegaskan TRDY untuk menunjukkan bahwa data yang valid ada di bus.
- e. Inisiator membaca data pada awal sinyal *clock-4* dan mengubah *byte* yang memungkinkan jalur yang diperlukan dalam persiapan untuk pembacaan berikutnya.
- f. Dalam contoh ini, target perlu waktu untuk menyiapkan blok data kedua untuk transmisi. Oleh karena itu, ia menegaskan kepada TRDY untuk memberi sinyal kepada inisiator bahwa tidak akan ada data baru selama siklus yang akan datang. Oleh karena itu, inisiator tidak membaca garis data pada awal siklus *clock* kelima dan tidak mengubah *byte* aktif selama siklus itu. Blok data dibaca pada awal *clock-6*.
- g. Selama *clock-6*, target menempatkan item data ketiga di dalam bus. Namun, dalam contoh ini, pemrakarsa belum siap untuk membaca item data (misalnya: memiliki kondisi buffer penuh sementara). Oleh karena itu deasserts IRDY. Ini akan menyebabkan target mempertahankan item data ketiga di bus untuk siklus *clock* tambahan.
- h. Inisiator tahu bahwa transfer data ketiga adalah yang terakhir, dan dengan demikian deasserts FRAME memberi sinyal kepada target bahwa ini adalah transfer data terakhir. Itu juga menegaskan IRDY untuk memberi sinyal bahwa ia siap untuk menyelesaikan transfer itu.
- i. Inisiator deasserts IRDY, mengembalikan bus ke status siaga, dan target deasserts TRDY dan DEVSEL.

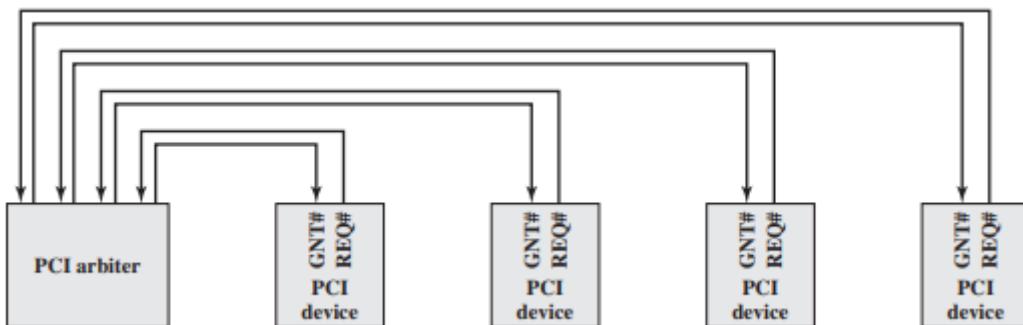


Sumber: (Stallings, 2016)

Gambar 5.9. Operasi Read PCI

D. Arbitrasi

PCI menggunakan skema arbitrase terpusat dan sinkron di mana setiap master memiliki sinyal permintaan unik (REQ) dan hibah (GNT). Garis sinyal ini melekat pada arbiter pusat (Gambar 5.9) dan jabat tangan permintaan-hibah sederhana digunakan untuk memberikan akses ke bus.



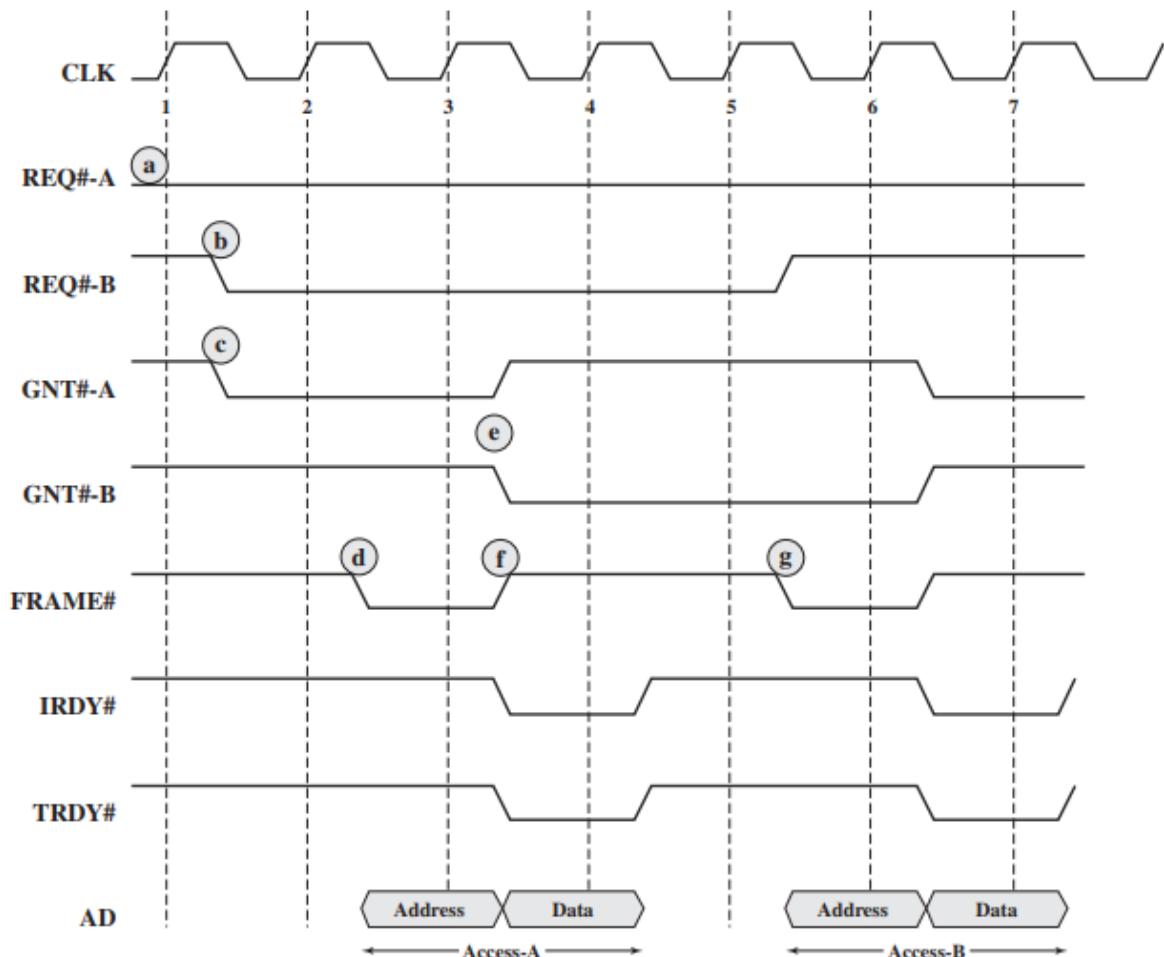
Sumber: (Stallings, 2016)

Gambar 5.10. PCI Bus Arbiter

Spesifikasi PCI tidak menentukan algoritma arbitrase tertentu. Arbiter dapat menggunakan pendekatan *first-come-first-served*, pendekatan round-robin, atau semacam skema prioritas. Master PCI harus melakukan arbitrase untuk setiap transaksi yang ingin dilakukan, di mana satu transaksi terdiri dari fase alamat diikuti oleh satu atau lebih fase data yang berdekatan.

Gambar 5.10 adalah contoh di mana perangkat A dan B melakukan arbitrase untuk bus. Urutan berikut terjadi:

- a. Pada beberapa titik sebelum dimulainya *clock-1*, A telah menegaskan sinyal REQ-nya. Arbiter mengambil sampel sinyal ini pada awal siklus *clock-1*.
- b. Selama siklus *clock-1*, B meminta penggunaan bus dengan menegaskan sinyal REQ-nya.
- c. Pada saat yang sama, arbiter menegaskan GNT-A untuk memberikan akses bus ke A.
- d. Master bus A mencicipi GNT-A pada awal *clock-2* dan mengetahui bahwa bus tersebut telah diberikan akses bus. Itu juga menemukan IRDY dan TRDY deasserted, menunjukkan bahwa bus itu menganggur. Dengan demikian, ini menegaskan FRAME dan menempatkan informasi alamat pada bus alamat dan perintah pada bus C/B/E (tidak ditampilkan). Ita juga terus menegaskan REQ-A, karena ia memiliki transaksi kedua untuk dilakukan setelah ini.
- e. Arbiter bus mengambil sampel semua jalur REQ pada awal *clock-3* dan membuat keputusan arbitrase untuk memberikan bus ke B untuk transaksi selanjutnya. Itu kemudian menegaskan GNT-B dan deasserts GNT-A. B tidak akan dapat menggunakan bus sampai kembali ke status siaga.
- f. FRAME deasserts untuk menunjukkan bahwa transfer data terakhir (dan satu-satunya) sedang berlangsung. Ini menempatkan data pada bus data dan memberi sinyal target dengan IRDY. Target membaca data di awal siklus *clock* berikutnya.
- g. Pada awal *clock-5*, B menemukan IRDY dan FRAME didekati sehingga dapat mengendalikan bus dengan menyatakan FRAME. Itu juga deasserts jalur REQ-nya, karena hanya ingin melakukan satu transaksi.



Sumber: (Stallings, 2016)

Gambar 5.11. PCI Bus Arbitration antara Dua Master

5.5. PCI Express

Seperti halnya bus sistem yang dibahas di bagian sebelumnya, skema PCI berbasis bus belum mampu mengimbangi permintaan laju data perangkat yang terpasang. Dengan demikian, versi baru, yang dikenal sebagai *PCI Express* (PCIe) telah dikembangkan. PCIe, seperti halnya QPI, adalah skema interkoneksi titik-ke-titik yang dimaksudkan untuk menggantikan skema berbasis bus seperti PCI.

Persyaratan utama untuk PCIe adalah kapasitas tinggi untuk mendukung kebutuhan perangkat I/O kecepatan data yang lebih tinggi, seperti Gigabit Ethernet. Persyaratan lain berkaitan dengan kebutuhan untuk mendukung aliran data yang tergantung waktu. Aplikasi seperti video-on-demand dan redistribusi audio juga memberikan batasan waktu nyata pada server. Banyak aplikasi komunikasi dan sistem kontrol PC tertanam juga memproses data secara *real-time*. Platform hari ini juga harus berurusan dengan beberapa transfer bersamaan dengan kecepatan data yang terus meningkat. Tidak lagi dapat diterima untuk memperlakukan semua data sama, lebih penting, misalnya, untuk memproses data streaming terlebih dahulu karena data real-time terlambat sama tidak bergunanya dengan tidak ada data. Data perlu ditandai agar sistem I/O dapat memprioritaskan alirannya di seluruh platform.

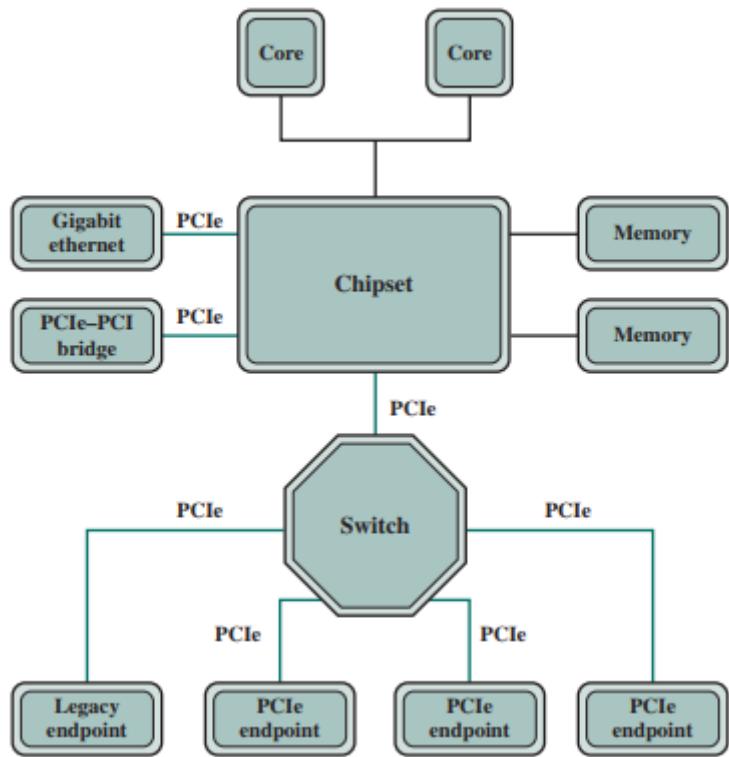
Arsitektur Fisik dan Logistik PCI

Gambar 5.11 menunjukkan konfigurasi tipikal yang mendukung penggunaan PCIe. Perangkat *root complex*, juga disebut sebagai *chipset* atau *host bridge*, menghubungkan prosesor dan subsistem memori ke *fabric switch* PCIe yang terdiri dari satu atau lebih perangkat *switch* PCIe dan PCIe.. Kompleks root bertindak sebagai perangkat penyangga, untuk menangani perbedaan kecepatan data antara pengontrol I/O dan komponen memori dan prosesor. Kompleks root juga menerjemahkan antara format transaksi PCIe dan prosesor serta sinyal memori dan persyaratan kontrol. *Chipset* biasanya akan mendukung beberapa port PCIe, beberapa di antaranya terpasang langsung ke perangkat PCIe, dan satu atau lebih yang terpasang pada saklar yang mengelola beberapa aliran PCIe. Tautan PCIe dari *chipset* dapat dilampirkan ke jenis perangkat berikut yang menerapkan PCIe:

- Switch: Switch mengelola beberapa aliran PCIe.
- Titik akhir PCIe: Perangkat I/O atau pengontrol yang mengimplementasikan PCIe, seperti saklar ethernet Gigabit, pengontrol grafis atau video, antarmuka disk, atau pengontrol komunikasi.
- Titik akhir *legacy*: Kategori titik akhir legacy dimaksudkan untuk desain yang ada yang telah dimigrasi ke PCIe, dan memungkinkan perilaku lama seperti penggunaan ruang I/O dan transaksi terkunci. Titik akhir PCIe tidak diizinkan untuk menggunakan ruang I/O saat runtime dan tidak boleh menggunakan transaksi yang dikunci. Dengan membedakan kategori-kategori ini, dimungkinkan bagi perancang sistem untuk membatasi atau menghilangkan perilaku lama yang berdampak negatif pada kinerja dan ketahanan sistem.
- PCIe/PCI bridge: Memungkinkan perangkat PCI yang lebih lama untuk terhubung ke sistem berbasis PCIe.
- Seperti QPI, interaksi PCIe didefinisikan menggunakan arsitektur protokol (gambar 5.12)

Arsitektur protokol PCIe mencakup lapisan berikut:

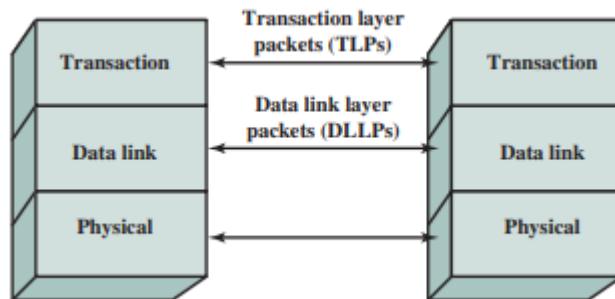
- o Fisik: Terdiri dari kabel aktual yang membawa sinyal, serta sirkuit dan logika untuk mendukung fitur tambahan yang diperlukan dalam transmisi dan penerimaan 1s dan 0s.
- o Tautan data: Bertanggung jawab atas transmisi dan kontrol aliran yang andal. Paket data yang dihasilkan dan dikonsumsi oleh DLL disebut Paket *Data Link Layer* (DLLPs).
- o Transaksi: Menghasilkan dan menggunakan paket data yang digunakan untuk menerapkan mekanisme transfer data pemuat/penyimpanan dan juga mengelola kontrol aliran paket-paket tersebut di antara dua komponen pada suatu tautan. Paket data yang dihasilkan dan dikonsumsi oleh TL disebut *Transaction Layer Packets* (TLPs).



Sumber: (Stallings, 2016)

Gambar 5.12. Konfigurasi Khas Menggunakan PCIe dan QPI

Di atas TL adalah lapisan perangkat lunak yang menghasilkan permintaan baca dan tulis yang diangkut oleh lapisan transaksi ke perangkat I/O menggunakan protokol transaksi berbasis paket.



Sumber: (Stallings, 2016)

Gambar 5.13. Layar pada Protokol PCIe

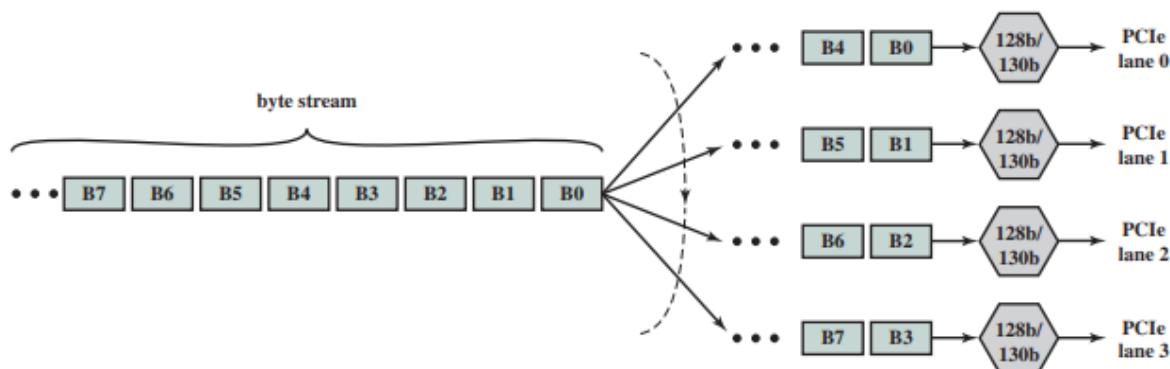
Lapisan Fisik PCIe

Mirip dengan QPI, PCIe adalah arsitektur *point-to-point*. Setiap port PCIe terdiri dari sejumlah lajur dua arah (perhatikan bahwa dalam QPI, lajur mengacu pada transfer dalam satu arah saja). Transfer di setiap arah di jalur adalah dengan cara memberi sinyal diferensial pada sepasang kabel. Port PCI dapat menyediakan 1, 4, 6, 16, atau 32 jalur. Uraian berikut merujuk pada spesifikasi PCIe 3.0, yang diperkenalkan pada akhir 2010.

Seperti halnya QPI, PCIe menggunakan teknik distribusi *multilane*. Gambar 5.13 menunjukkan contoh untuk port PCIe yang terdiri dari empat jalur. Data didistribusikan ke empat jalur 1 byte sekaligus menggunakan skema round-robin sederhana. Di setiap jalur fisik, data di-buffer dan diproses 16 byte (128 bit) sekaligus. Setiap blok 128 bit dikodekan ke dalam kode sandi 130-bit yang unik untuk

transmisi; ini disebut sebagai 128b/130b encoding. Dengan demikian, kecepatan data efektif dari jalur individu dikurangi dengan faktor 128/130.

Untuk memahami alasan penyandian 128b/130b, perhatikan bahwa tidak seperti QPI, PCIe tidak menggunakan jalur *clock* nya untuk menyinkronkan bit stream. Artinya, jalur *clock* tidak digunakan untuk menentukan titik awal dan akhir dari setiap bit yang masuk; digunakan untuk tujuan pensinyalan lainnya saja. Namun, penerima perlu disinkronkan dengan pemancar, sehingga penerima tahu kapan setiap bit dimulai dan berakhir. Jika ada penyimpangan antara *clock* yang digunakan untuk transmisi bit dan penerimaan pemancar dan penerima, kesalahan dapat terjadi. Untuk mengimbangi kemungkinan drift, PCIe bergantung pada sinkronisasi penerima dengan pemancar berdasarkan sinyal yang ditransmisikan. Seperti halnya QPI, PCIe menggunakan pensinyalan diferensial pada sepasang kabel. Sinkronisasi dapat dicapai oleh penerima yang mencari transisi dalam data dan menyinkronkan *clock*-nya ke transisi. Namun, pertimbangkan bahwa dengan string panjang 1s atau 0s yang menggunakan pensinyalan diferensial, *output*nya adalah tegangan konstan selama periode waktu yang lama. Dalam keadaan ini, setiap penyimpangan antara *clock* pemancar dan penerima akan mengakibatkan hilangnya sinkronisasi antara keduanya.



Sumber: (Stallings, 2016)
Gambar 5.14. Distribusi PCIe Multiline

Pendekatan umum, dan yang digunakan dalam PCIe 3.0, untuk mengatasi masalah string panjang bit dari satu nilai sedang berebut. *Scrambling*, yang tidak menambah jumlah bit yang akan ditransmisikan, adalah teknik pemetaan yang cenderung membuat data tampak lebih acak. Perebutan cenderung menyebar jumlah transisi sehingga mereka muncul di penerima lebih seragam, yang bagus untuk sinkronisasi. Juga, sifat transmisi lainnya, seperti sifat spektral, ditingkatkan jika data lebih bersifat acak daripada konstan atau berulang. Untuk diskusi lebih lanjut tentang pengacakan.

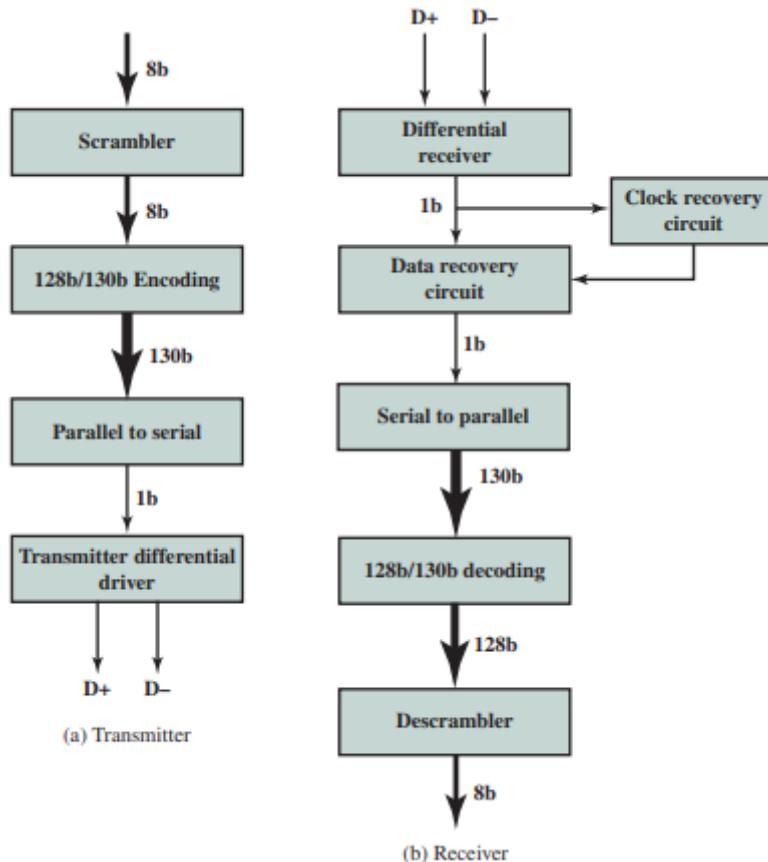
Teknik lain yang dapat membantu sinkronisasi adalah penyandian, di mana bit tambahan dimasukkan ke dalam aliran bit untuk memaksa transisi. Untuk PCIe 3.0, masing-masing kelompok 128 bit *input* dipetakan ke dalam blok 130-bit dengan menambahkan *header* sinkronisasi blok 2-bit. Nilai *header* adalah 10 untuk blok data dan 01 untuk apa yang disebut blok set berurutan, yang mengacu pada blok informasi tingkat tautan.

Gambar 5.14 menggambarkan penggunaan pengacakan dan pengodean. Data yang akan dikirim dimasukkan ke pengacakan. *Output* teracak kemudian dimasukkan ke dalam encoder 128b/130b, yang buffer 128 bit dan kemudian memetakan blok 128-bit ke dalam blok 130-bit. Blok ini kemudian melewati konverter paralel ke serial dan ditransmisikan sedikit demi sedikit menggunakan pensinyalan diferensial.

Di penerima, *clock* disinkronkan ke data yang masuk untuk memulihkan aliran bit. Ini kemudian melewati konverter serial-ke-paralel untuk menghasilkan aliran blok 130-bit. Setiap blok dilewatkan

melalui decoder 128b/130b untuk memulihkan pola bit orak asli, yang kemudian diuraikan untuk menghasilkan bit stream asli.

Dengan menggunakan teknik ini, kecepatan data 16 GB/s dapat dicapai. Satu detail terakhir lagi; setiap pengiriman blok data melalui tautan PCI dimulai dan diakhiri dengan urutan pembingkaian 8-bit yang dimaksudkan untuk memberi waktu kepada penerima untuk menyinkronkan dengan aliran bit lapisan fisik yang masuk.



Sumber: (Stallings, 2016)

Gambar 5.15. Blok Diagram PCIe Transmit dan Receive Block

Lapisan Transaksi PCIe

Transaction Layer (TL) menerima permintaan baca dan tulis dari perangkat lunak di atas TL dan membuat paket permintaan untuk pengiriman ke tujuan melalui lapisan tautan. Sebagian besar transaksi menggunakan teknik transaksi split, yang bekerja dengan cara berikut. Paket permintaan dikirim oleh perangkat PCIe sumber, yang kemudian menunggu respons, disebut paket penyelesaian. Penyelesaian setelah permintaan dimulai oleh penyelesaian hanya ketika memiliki data dan/atau status siap untuk pengiriman. Setiap paket memiliki pengidentifikasi unik yang memungkinkan paket penyelesaian diarahkan ke originator yang benar. Dengan teknik transaksi split, penyelesaian dipisahkan dalam waktu dari permintaan, berbeda dengan operasi bus yang khas di mana kedua sisi transaksi harus tersedia untuk merebut dan menggunakan bus. Antara permintaan dan penyelesaian, lalu lintas PCIe lain dapat menggunakan tautan.

Pesan TL dan beberapa transaksi tulis adalah transaksi yang diposting, artinya tidak ada respons yang diharapkan. Format paket TL mendukung pengalaman memori 32-bit dan pengalaman memori 64-bit yang diperluas. Paket juga memiliki atribut seperti *no-snoop*, *relaxingordering*, dan *priority*, yang dapat digunakan untuk merutekan paket-paket ini secara optimal melalui subsistem I/O.

ruang alamat dan jenis transaksi TL mendukung empat ruang alamat:

- 1) Memori: Ruang memori termasuk memori utama sistem. Ini juga termasuk perangkat PCIe I/O. Rentang alamat memori tertentu dipetakan ke dalam perangkat I/O.
- 2) I/O: Ruang alamat ini digunakan untuk perangkat PCI lawas, dengan rentang alamat memori yang dicadangkan digunakan untuk mengatasi perangkat I/O lawas.
- 3) Konfigurasi: Ruang alamat ini memungkinkan TL untuk membaca/menulis register konfigurasi yang terkait dengan perangkat I/O.
- 4) Pesan: Ruang alamat ini untuk sinyal kontrol yang terkait dengan interupsi, penanganan kesalahan, dan manajemen daya.

Tabel 5.5 menunjukkan jenis transaksi yang disediakan oleh TL. Untuk memori, I/O, dan ruang alamat konfigurasi, ada transaksi baca dan tulis. Dalam kasus transaksi memori, ada juga fungsi permintaan kunci baca. Penguncian operasi terjadi sebagai akibat driver perangkat yang meminta akses atom ke register pada perangkat PCIe. Driver perangkat, misalnya, dapat secara atom membaca, memodifikasi, dan kemudian menulis ke register perangkat. Untuk mencapai ini, driver perangkat menyebabkan prosesor untuk menjalankan instruksi atau rangkaian instruksi. Kompleks root mengubah instruksi prosesor ini menjadi urutan transaksi PCIe, yang melakukan permintaan baca dan tulis individual untuk driver perangkat. Jika transaksi ini harus dijalankan secara atom, kompleks akar mengunci tautan PCIe saat menjalankan transaksi. Penguncian ini mencegah transaksi yang bukan merupakan bagian dari urutan yang terjadi. Urutan transaksi ini disebut operasi locked. Rangkaian instruksi prosesor tertentu yang dapat menyebabkan operasi terkunci terjadi tergantung pada

kumpulan chip sistem dan arsitektur prosesor.

Untuk menjaga kompatibilitas dengan PCI, PCIe mendukung siklus konfigurasi Tipe 0 dan Tipe 1. Siklus Tipe 1 merambat ke hilir hingga mencapai antarmuka jembatan yang menampung bus (tautan) tempat perangkat target berada. Transaksi konfigurasi dikonversi pada tautan tujuan dari Tipe 1 ke Tipe 0 oleh jembatan. Akhirnya, pesan penyelesaian digunakan dengan transaksi split untuk memori, I/O, dan transaksi konfigurasi.

Tabel 5.5. Type Transaksi PCIe TLP

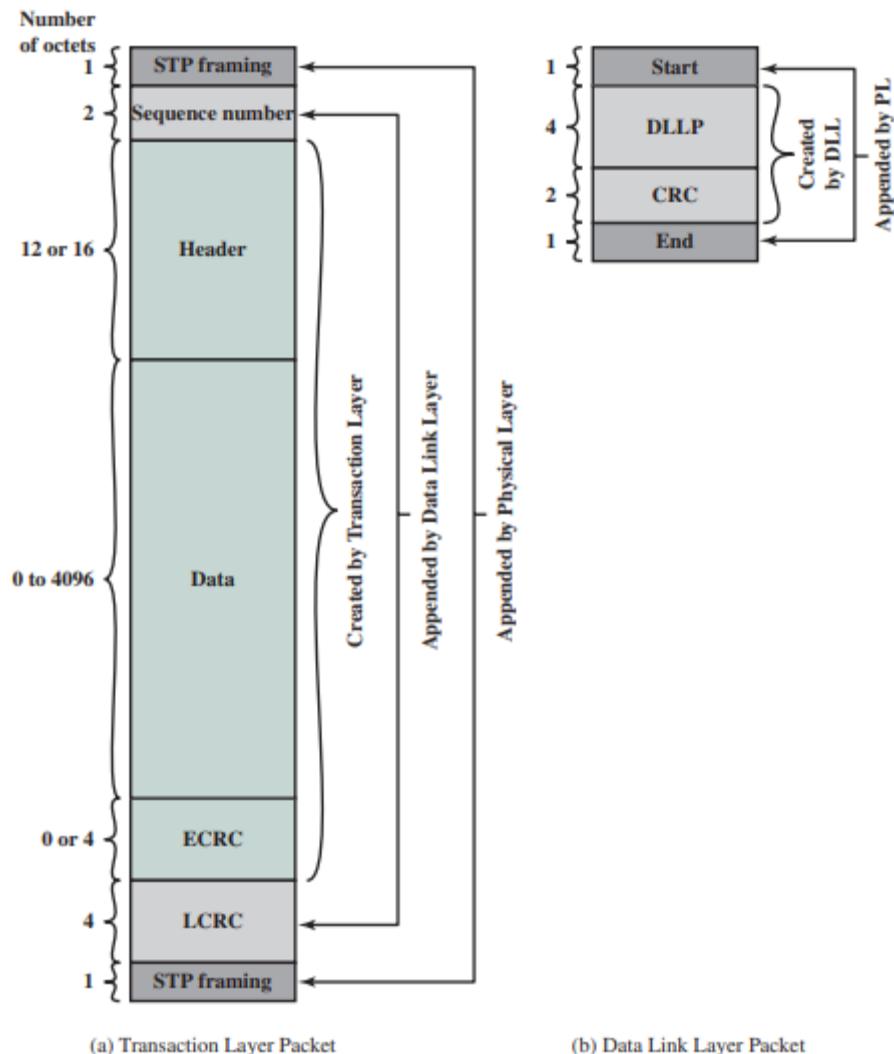
Address Space	TLP Type	Purpose
Memory	Memory Read Request	Transfer data ke atau dari lokasi dalam sistem memory map
	Memory Read Lock Request	
	Memory Write Request	
I/O	I/O Read Request	Transfer data ke atau dari lokasi dalam sistem memory map untuk perangkat lama
	I/O Write Request	
Configuration	Config Type 0 Read Request	Transfer data ke atau dari lokasi dalam ruang konfigurasi dari sebuah perangkat PCIe
	Config Type 0 Write Request	
	Config Type 1 Read Request	
	Config Type 1 Write Request	
Message	Message Request	menyediakan perpesanan dalam merek dan pelaporan acara
	Message Request With Data	
Memory, I/O, Configuration	Completion	Kembali ke persyaratan tertentu
	Completion with Data	
	Completion Locked	
	Completion Locked with Data	

Sumber: (Stallings, 2016)

Rakitan paket TLP pada transaksi PCIe disampaikan menggunakan paket lapisan transaksi, yang diilustrasikan dalam Gambar 5.15a. TLP berasal dari lapisan transaksi perangkat pengirim dan berakhir pada lapisan transaksi perangkat penerima.

Perangkat lunak lapisan atas mengirim ke TL informasi yang diperlukan untuk TL untuk membuat inti dari TLP, yang terdiri dari bidang-bidang berikut:

- *Header*: *Header* menggambarkan tipe paket dan termasuk informasi yang dibutuhkan oleh penerima untuk memproses paket, termasuk informasi routing yang diperlukan. Format tajuk internal dibahas selanjutnya.
- *Data*: Bidang data hingga 4096 byte dapat dimasukkan dalam TLP. Beberapa TLP tidak mengandung bidang data.
- *ECRC*: Bidang CRC end-to-end opsional memungkinkan lapisan TL tujuan untuk memeriksa kesalahan dalam *header* dan bagian data dari TLP.



Sumber: (Stallings, 2016)

Gambar 5.16. Format Unit Data Protokol PCIe

Lapisan Tautan Data PCIe

Tujuan dari lapisan tautan data PCIe adalah untuk memastikan pengiriman paket yang dapat diandalkan di seluruh tautan PCIe. DLL berpartisipasi dalam pembentukan TLP dan juga mentransmisikan DLLP.

Paket Lapisan Data Link.

Paket data lapisan tautan berasal dari lapisan tautan data dari perangkat pengirim dan berakhir pada DLL perangkat di ujung lain dari tautan tersebut. Gambar 5.15b menunjukkan format DLLP. Ada tiga

kelompok penting DLL yang digunakan dalam mengelola tautan: paket kontrol aliran, paket manajemen daya, dan paket TLP ACK dan NAK. Paket manajemen daya digunakan dalam mengelola penganggaran platform daya. Paket kontrol aliran mengatur laju di mana TLP dan DLLP dapat ditransmisikan melalui tautan. Paket ACK dan NAK digunakan dalam pemrosesan TLP, dibahas dalam paragraf berikut.

Pemrosesan Paket Layer Transaksi.

DLL menambahkan dua bidang ke inti TLP yang dibuat oleh TL (Gambar 5.15a): nomor urut 16-bit dan CRC layer-link 32-bit (LCRC). Sedangkan bidang inti yang dibuat di TL hanya digunakan di TL tujuan, dua bidang yang ditambahkan oleh DLL diproses di setiap simpul perantara dalam perjalanan dari sumber ke tujuan. Ketika TLP tiba di suatu perangkat, DLL menghapus nomor urut dan bidang LCRC dan memeriksa LCRC. Ada dua kemungkinan:

1. Jika tidak ada kesalahan terdeteksi, bagian inti dari TLP diserahkan ke lapisan transaksi lokal. Jika perangkat penerima ini adalah tujuan yang dituju, maka TL memproses TLP. Jika tidak, TL menentukan rute untuk TLP dan meneruskannya kembali ke DLL untuk transmisi melalui tautan berikutnya dalam perjalanan ke tujuan.
2. Jika kesalahan terdeteksi, DLL menjadwalkan paket NAK DLL untuk kembali ke pemancar jarak jauh. TLP dihilangkan.

Ketika DLL mentransmisikan ke TLP, itu mempertahankan salinan TLP. Jika menerima NAK untuk TLP dengan nomor urut ini, itu mentransmisikan kembali TLP. Ketika menerima ACK, itu membuang TLP buffered.

5.6. Universal Serial Bus (USB)

Universal Serial Bus (USB) adalah standar industri yang menetapkan spesifikasi untuk kabel dan konektor serta protokol untuk koneksi, komunikasi, dan catu daya (interfacing) antara komputer, periferal, dan komputer lainnya. Berbagai macam perangkat keras USB ada, termasuk beberapa konektor yang berbeda, di mana USB-C adalah yang terbaru. Dirilis pada tahun 1996, standar USB saat ini dikelola oleh USB Implementers Forum (USB-IF). Sampai saat ini, ada empat generasi spesifikasi USB: USB 1.x, USB 2.0, USB 3.x dan USB4.

Sekelompok perusahaan (Compaq, DEC, IBM, Intel, Microsoft, NEC, dan Nortel) mulai pengembangan USB pada tahun 1994, dengan tujuan memudahkan untuk menghubungkan perangkat eksternal ke PC dengan mengganti banyak konektor di belakang PC, dan menyederhanakan konfigurasi perangkat lunak dari semua perangkat yang terhubung ke USB, serta memungkinkan lebih besar kecepatan data untuk perangkat eksternal. Tabel 5.6 menyajikan kecepatan data dari beberapa versi USB mulai dari USB 1.0 sampai USB 4.

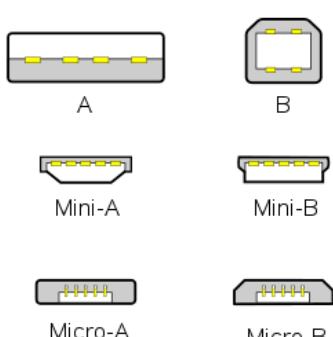
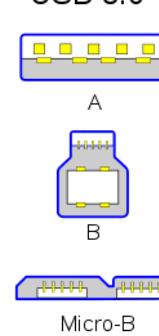
Dari perspektif pengguna komputer, antarmuka USB meningkatkan kemudahan penggunaan dalam beberapa cara:

- 1) Antarmuka USB mengkonfigurasi sendiri, menghilangkan kebutuhan pengguna untuk menyesuaikan pengaturan perangkat untuk kecepatan atau format data, atau mengkonfigurasi interupsi, alamat *input/output*, atau saluran akses memori langsung.
- 2) Konektor USB standar di host, sehingga perangkat apa pun dapat menggunakan sebagian besar wadah yang tersedia.
- 3) USB memanfaatkan sepenuhnya kekuatan pemrosesan tambahan yang secara ekonomis dapat dimasukkan ke dalam perangkat periferal sehingga mereka dapat mengatur sendiri.

Dengan demikian, perangkat USB sering tidak memiliki pengaturan antarmuka yang dapat disesuaikan pengguna.

- 4) Antarmuka USB bersifat hot-swappable (perangkat dapat dipertukarkan tanpa me-reboot komputer host).
- 5) Perangkat kecil dapat diberdayakan langsung dari antarmuka USB, menghilangkan kebutuhan untuk kabel catu daya tambahan.

Tabel 5.6. Kecepatan Data beberapa versi USB

Type	Tahun dibuat	Data rate	Bentuk konektor
USB 1.0	1996	1.5 Mbit/s	
USB 1.1	1998	1.5 Mbit/s (Low Speed) 12 Mbit/s (Full Speed)	
USB 2.0	2001	1.5 Mbit/s (Low Speed) 12 Mbit/s (Full Speed) 480 Mbit/s (High Speed)	<p style="text-align: center;">USB 1.1 – 2.0</p> 
USB 3.0	2011	5 Gbit/s (SuperSpeed)	
USB 3.1	2014	10 Gbit/s (SuperSpeed+)	
USB 3.2.	2017	20 Gbit/s (SuperSpeed+)	<p style="text-align: center;">USB 3.0</p> 
USB 4	2019	40 Gbit/s (SuperSpeed+ and Thunderbolt 3)	<p style="text-align: center;">Type C</p> 

Sumber: (Penulis, 2020)

Standar USB juga memberikan banyak manfaat bagi produsen perangkat keras dan pengembang perangkat lunak, khususnya dalam kemudahan relatif implementasi:

- 1) Standar USB menghilangkan persyaratan untuk mengembangkan antarmuka berpemilik ke perangkat baru.
- 2) Berbagai kecepatan transfer yang tersedia dari antarmuka USB sesuai dengan perangkat mulai dari keyboard dan mouse hingga antarmuka video streaming.
- 3) Antarmuka USB dapat dirancang untuk memberikan latensi terbaik yang tersedia untuk fungsi-fungsi penting waktu, atau dapat diatur untuk melakukan transfer latar belakang data massal dengan sedikit dampak pada sumber daya sistem.

- 4) Antarmuka USB digeneralisasi tanpa jalur sinyal yang didedikasikan hanya untuk satu fungsi dari satu perangkat.

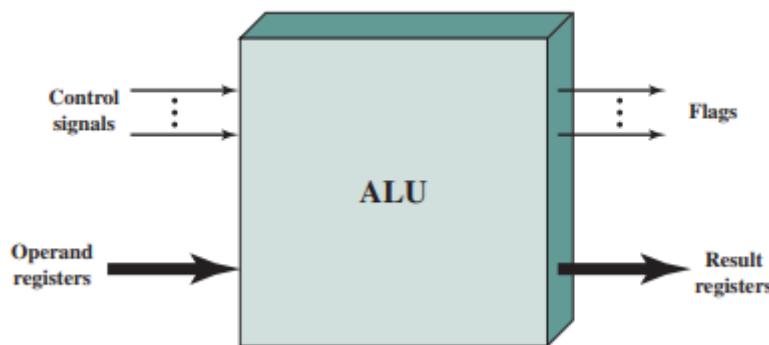
Seperti semua standar, USB memiliki beberapa batasan untuk desainnya:

- 1) Kabel USB panjangnya terbatas, karena standar itu dimaksudkan untuk periferal di atas meja yang sama, bukan di antara ruangan atau bangunan. Namun, port USB dapat dihubungkan ke *gateway* yang mengakses perangkat yang jauh.
- 2) USB memiliki topologi *tree network* yang ketat dan protokol *master/slave* untuk menangani perangkat periferal; perangkat-perangkat itu tidak dapat berinteraksi satu sama lain kecuali melalui host, dan dua host tidak dapat berkomunikasi melalui port USB mereka secara langsung. Beberapa ekstensi untuk batasan ini dimungkinkan melalui USB *On-The-Go*.
- 3) *Host* tidak dapat menyiarakan sinyal ke semua perangkat sekaligus, masing-masing harus ditangani secara individual.

6. Komputer Aritmatika

6.1. ALU

ALU adalah bagian komputer yang melakukan operasi aritmatika dan logika pada data. Semua elemen lain dari sistem komputer (CU, Register, Memori, I/O) bertugas untuk membawa data ke dalam ALU untuk diproses, dan kemudian mengambil hasilnya kembali.



Sumber: (Stallings, 2016)

Gambar 6.1. Skema *Input/Output* pada ALU

ALU pada dasarnya terbentuk dari perangkat logika digital sederhana yang dapat menyimpan angka biner dan melakukan operasi logika Boolean. Gambar 6.1 menunjukkan, secara umum, bagaimana ALU saling terhubung dengan prosesor. *Operand* untuk operasi aritmatika dan logika diberikan kepada ALU melalui register, dan hasil operasi disimpan dalam register. Register ini adalah lokasi penyimpanan sementara di dalam prosesor yang terhubung ke ALU melalui bus prosesor. ALU juga dapat menetapkan nilai beberapa Flag sebagai hasil operasi. Misalkan, flag *overflow* diatur ke 1 jika hasil perhitungan melebihi panjang register tempat penyimpanannya. Nilai Flag yang dihasilkan oleh ALU juga disimpan dalam register di dalam prosesor. Prosesor menyediakan sinyal yang mengontrol operasi ALU dan pergerakan data masuk dan keluar dari ALU.

Aritmatika komputer biasanya dilakukan pada dua jenis angka yang sangat berbeda, yaitu: integer dan floating point.

6.2. Representasi Sembarang Bilangan Biner

Dalam sistem bilangan biner, sembarang bilangan (*arbitrary numbers*) dapat direpresentasikan hanya dengan simbol 0 (nol), 1 (satu), tanda minus (untuk bilangan negatif), dan periode (titik radix) untuk bilangan dengan komponen fraksional (pecahan). Misal: penyajian bilangan biner -1101.0101 setara dengan desimal -13.28125, didapat dengan menjumlahkan hasil perkalian bobot biner dengan digit biner sesuai posisinya ($8 + 4 + 1 + 0.25 + 0.03125$).

Bobot biner -->	2^3	2^2	2^1	2^0	2^{-1}	2^{-2}	2^{-3}	2^{-4}
	↓	↓	↓	↓	↓	↓	↓	↓
	1	1	0	1	.	0	1	0
(Bobot x bit) -->	8	4	0	1	0	0.25	0	0.03125

Untuk keperluan penyimpanan dan pemrosesan komputer, prosesor dan memori tidak mengenal simbol khusus untuk tanda minus dan titik radix. Hanya digit biner (0 dan 1) yang dapat digunakan untuk mewakili bilangan apapun.

Untuk bilangan bulat positif, representasi bilangan disajikan oleh seluruh bit pada bilangan tersebut. Misalnya bilangan bulat 8 bit dapat menyajikan bilangan dari 0 sampai 255. Secara umum, jika urutan n-bit bilangan biner $a_{n-1}, a_{n-2}, \dots, a_1, a_0$ adalah bilangan *unsigned integer* A, maka nilai a digit-ke i adalah:

$$A = \sum_{i=0}^{n-1} 2^i a_i \quad (6.1)$$

A. Representasi *Sign-Magnitude*

Pada *signed-magnitude* untuk menyatakan bilangan bulat positif dan negatif digunakan *sign bit*, yang diletakkan pada bit paling kanan (MSB), sisanya untuk menyatakan nilai dari bilangan itu (*magnitude*). Bentuk representasi paling sederhana yang menggunakan *sign bit* adalah representasi integer *sign-magnitude*. Dalam bilangan bulat n-bit, semua bit n-1 mewakili digit bilangan bulat (*magnitude*), sedangkan bit ke n merupakan sign bit. Jika *sign bit* = 0, angkanya positif; jika *sign bit* = 1, angkanya negatif. Berikut adalah contoh penyajian bilangan *sign-magnitude*:

$$+18 = 00010010$$

$$-18 = 10010010 \text{ (sign-magnitude)}$$

Kasus umum *sign-magnitude* dapat dinyatakan sebagai berikut: Untuk setiap bilangan A yang disajikan dalam bentuk *sign-magnitude*, berlaku:

$$A = \begin{cases} \sum_{i=0}^{n-2} 2^i a_i ; \text{ jika } a_{n-1} = 0 \\ \sum_{i=0}^{n-2} 2^i a_i ; \text{ jika } a_{n-1} = 1 \end{cases} \quad (6.2)$$

Jangkauan bilangan yang dapat direpresentasikan pada bilangan bulat (integer) *sign-magnitude* n bit adalah $-2^{n-1}-1$ sampai $+2^{n-1}-1$. Misalnya pada bilangan sign magnitude 8 bit, dapat merepresentasikan bilangan mulai dari $-2^{8-1}-1$ sampai $+2^{8-1}-1$ atau -127 sampai +127. Salah satu kelemahan representasi sign magnitude adalah ada dua representasi bilangan 0, yaitu:

$$+0_{10} = 00000000$$

$$-0_{10} = 10000000 \text{ (bit MSB merupakan sign-magnitude)}$$

Karena kelemahan ini, representasi sign-magnitude jarang digunakan dalam mengimplementasikan bagian integer dari ALU.

B. Representasi Komplemen-2 (*Two's Complement*)

Seperti sign-magnitude, representasi komplemen-2 menggunakan bit yang paling signifikan sebagai bit tanda, membuatnya mudah untuk menguji apakah bilangan bulat bernilai positif atau negatif. Tabel 6.1 menyajikan karakteristik utama dari representasi komplemen-2. Representasi komplemen-2 pada umumnya digunakan untuk menghasilkan angka negatif.

Tabel 6.1. Karakteristik Representasi Komplemen-2 dan Aritmatika

No	Karakteristik	Nilai
1	Range	-2^n sampai $+2^{n-1}-1$
2	Representasi dari 0	Hanya ada satu
3	Perluasan panjang bit	Tambahkan posisi bit tambahan ke kiri dan isi dengan nilai dari bit tanda asli.
4	Negasi	Dihasilkan dari komplemen-2, yaitu lakukan komplemen Boolean dari setiap bit yang sesuai angka positif, lalu tambahkan 1 ke pola bit yang dihasilkan dilihat sebagai bentuk unsigned integer.
5	Aturan <i>overflow</i>	Jika dua angka dengan tanda yang sama (baik positif atau negatif) ditambahkan, maka <i>overflow</i> terjadi jika dan hanya jika hasilnya memiliki tanda sebaliknya.
6	Aturan pengurangan	Untuk mengurangi B dari A, ambil komplemen-2 dari B dan tambahkan ke A.

Sumber: (Stallings, 2016)

Asumsikan A sebagai bilangan integer n-bit. Dalam representasi komplemen-2, jika A positif maka bit yang paling kiri (*sign bit*) a_{n-1} adalah nol. Bit yang tersisa adalah *magnitude* (mewakili besarnya bilangan). Bilangan positif didapat dengan:

$$A = \sum_{i=0}^{n-2} 2^i a_i ; \text{ untuk } A > 0 \quad (6.3)$$

Angka nol pada *sign bit* diidentifikasi sebagai positif. Jangkauan bilangan bulat positif yang dapat diwakili adalah dari 0 (semua bit besarnya adalah 0) hingga $2^{n-1} - 1$. Misalnya pada bilangan integer 8 bit, maka range bilangan positif yang dapat disajikan adalah mulai dari biner 0000 0000 sampai 0111 1111, atau dalam desimal: mulai 0 sampai 127 ($2^{8-1}-1=128-1$).

Sekarang, untuk angka negatif A ($A < 0$), *sign bit* a_{n-1} bernilai satu. Bit berikutnya yang tersisa dapat mengambil salah satu dari nilai 2^{n-1} . Oleh karena itu, kisaran bilangan bulat negatif yang dapat direpresentasikan adalah dari -1 hingga -2^{n-1} . Misalnya pada bilangan integer 8 bit, maka range bilangan negatif yang dapat disajikan adalah mulai dari biner 1111 1111 sampai 1000 0000, atau dalam desimal: mulai -1 sampai -128 ($-2^{8-1}=-128$).

Dalam representasi *unsigned integer*, untuk menghitung nilai integer dari representasi bit, bobot bit yang paling signifikan adalah $+2^{n-1}$. Untuk representasi dengan bit tanda, ternyata properti aritmatika yang diinginkan tercapai, jika bobot bit paling signifikan adalah -2^{n-1} . Ini adalah konvensi yang digunakan dalam dua perwakilan komplemen, menghasilkan ekspresi berikut untuk bilangan negatif:

$$A = -2^{n-1} a_{n-1} + \sum_{i=0}^{n-2} 2^i a_i \quad (6.4)$$

Tabel 6.2. Representasi Integer 4-Bit

Representasi Desimal	Representasi Sign-Magnitude	Representasi Komplemen-2	Representasi Bias
+8	-	-	1111
+7	0111	0111	1110
+6	0110	0110	1101
+5	0101	0101	1100
+4	0100	0100	1011
+3	0011	0011	1010
+2	0010	0010	1001
+1	0001	0001	1000
+0	0000	0000	0111
-0	1000	-	-
-1	1001	1111	0110
-2	1010	1110	0101
-3	1011	1101	0100
-4	1100	1100	0011
-5	1101	1011	0010
-6	1110	1010	0001
-7	1111	1001	0000
-8	-	1000	-

Sumber: (Stallings, 2016)

Tabel 6.2 membandingkan representasi *sign-magnitude* dan komplemen-2 untuk bilangan bulat 4-bit. Meskipun representasi komplemen-2 sedikit membingungkan dari sudut pandang manusia, namun ini digunakan pada operasi aritmatika dasar pada komputer, yaitu: penambahan dan pengurangan. Untuk alasan ini, komplemen-2 hampir secara universal digunakan sebagai representasi prosesor untuk bilangan bulat.

-128	64	32	16	8	4	2	1

(a) Nilai posisi pada 8 bit Komplemen-2

-128	64	32	16	8	4	2	1
1	0	0	0	0	0	1	1

$$-128 + 2 + 1 = -125$$

(b) Konversi Biner 10000011 ke Desimal

-128	64	32	16	8	4	2	1
1	0	0	0	1	0	0	0

$$-128 + 8 = -120$$

(c) Konversi 120 Desimal ke Biner Komplemen-2

Sumber: (penulis, 2020)

Gambar 6.2. Nilai Posisi antara Komplemen ke-2 dan Desimal

Sebuah ilustrasi mengenai sifat komplemen-2 dengan menggunakan bobot nilai posisi, diperlihatkan pada gambar 6.2, di mana bobot posisi paling kanan dalam kotak adalah $1 (2^0)$ dan setiap posisi berikutnya di sebelah kiri bernilai kelipatannya, sampai posisi paling kiri, yaitu dinegaskan. Bilangan komplemen-2 yang paling negatif yang dapat direpresentasikan adalah -2^{n-1} ; jika ada bit selain *sign bit* yang bernilai satu, bit itu menambah jumlah positif ke bilangan tersebut. Juga, jelas bahwa angka negatif harus bernilai 1 di *sign bit* paling kiri dan angka positif harus memiliki 0 di posisi itu. Dengan

demikian, bilangan positif terbesar adalah 0 diikuti oleh semua 1s, yang sama dengan $2^{n-1}-1$. Gambar 6.2 berikutnya, mengilustrasikan penggunaan kotak nilai untuk mengkonversi dari komplemen-2 (-125 dan -120) menjadi desimal dan dari desimal ke komplemen-2. Terkadang diinginkan untuk mengambil bilangan bulat n-bit dan menyimpannya dalam m bit, di mana $m > n$. Perluasan panjang bit ini disebut sebagai rentang ekstensi (*range extension*), karena rentang notasi *sign-magnitude*, ini mudah dilakukan: cukup pindahkan bit tanda ke posisi paling kiri yang baru dan isi dengan nol.

$+18 = 00010010$	(<i>Sign-magnitude</i> , 8 bit)
$+18 = 000000000001001010$	(<i>Sign-magnitude</i> , 16 bit)
$-18 = 10010010$	(<i>Sign-magnitude</i> , 8 bit)
$-18 = 100000000010010010$	(<i>Sign-magnitude</i> , 16 bit)

Prosedur ini tidak akan bekerja untuk dua bilangan bulat pelengkap negatif. Dengan menggunakan contoh yang sama, angka yang dapat diekspresikan diperluas dengan menambah panjang bit.

$+18 = 00010010$	(komplemen-2, 8 bit)
$+18 = 000000000001001010$	(komplemen-2, 16 bit)
$-18 = 11101110$	(komplemen-2, 8 bit)
$-32.658 = 100000001101110$	(komplemen-2, 16 bit)

Sebagai gantinya, aturan untuk bilangan bulat komplemen dua adalah untuk memindahkan bit tanda ke posisi paling kiri baru dan mengisi dengan salinan bit tanda. Untuk angka positif, isi dengan nol, dan untuk angka negatif, isi dengan angka. Ini disebut ekstensi tanda.

$-18 = 11101110$	(komplemen-2, 8 bits)
$-18 = 111111111101110$	(komplemen-2, 16 bits)

Representasi yang dibahas dalam bagian ini terkadang disebut sebagai titik tetap (*fixed-point*). Ini karena titik radix (titik biner) ditetapkan dan diasumsikan berada di sebelah kanan digit paling kanan. Pemrogram dapat menggunakan representasi yang sama untuk pecahan biner dengan menskalakan angka sehingga titik biner secara implisit diposisikan di beberapa lokasi lain.

6.3. Aritmatika Integer

A. Negasi

Dalam representasi sign-magnitude, aturan untuk membentuk negasi bilangan bulat adalah: membalikkan bit tanda. Dalam notasi Komplemen-2, negasi bilangan bulat dapat dibentuk dengan aturan berikut:

1. Ambil komplemen Boolean dari setiap bit integer (termasuk bit tanda), kemudian ganti masing-masing 1 ke 0 dan setiap 0 ke 1.
2. Memperlakukan hasilnya sebagai bilangan biner *unsigned integer*, tambahkan 1.

Proses dua langkah ini disebut sebagai operasi komplemen berpasangan, atau pengambilan komplemen berpasangan dari sebuah bilangan bulat. Contoh, untuk bilangan +18 desimal, disajikan dalam biner bit menjadi:

$$\begin{array}{rcl}
 +18 & = 0001\ 0010 \\
 \text{bitwise complement} & = \underline{\underline{1110\ 1101}} + & (\text{Komplemen-1}) \\
 & & 1 \\
 & 1110\ 1110 & = -18 \quad (\text{Komplement-2})
 \end{array}$$

Seperti yang diharapkan, negatif dari (-18) akan kembali ke nilai positifnya:

$$\begin{array}{rcl}
 -18 & = & 11101110 \\
 \text{bitwise complement} & = & \underline{\underline{00010001}} + \quad \text{(Komplemen-1)} \\
 & & \quad \quad \quad 1 \\
 & & 00010010 = +18 \quad \text{(Komplemen-2)}
 \end{array}$$

Validitas operasi tersebut dapat dijelaskan dengan menggunakan definisi representasi pelengkap dua-duanya dalam Persamaan (6.2). Sekali lagi, tafsirkan urutan n-bit digit biner $a_{n-1} a_{n-2} \dots a_1 a_0$ sebagai bilangan bulat komplemen-2 A, sehingga nilainya

$$A = -2^{n-1}a_{n-1} + \sum_{i=0}^{n-2} 2^i a_i \quad (6.5)$$

Sekarang bentuk komplemen bitwise $\overline{a_{n-1}} \overline{a_{n-2}} \dots \overline{a_1} \overline{a_0}$, dan perlakuan ini sebagai bilangan *unsigned integer*, tambahkan 1. Terakhir, tafsirkan urutan n-bit yang dihasilkan dari digit biner sebagai komplemen-2 bilangan bulat B, sehingga nilainya adalah:

$$A = -2^{n-1}a_{n-1} + 1 + \sum_{i=0}^{n-2} 2^i \overline{a_i} \quad (6.5)$$

Derivasi sebelumnya mengasumsikan bahwa pertama-tama komplemen bitwise dari A dapat diperlakukan sebagai *unsigned integer* untuk tujuan penambahan 1, dan kemudian memperlakukan hasilnya sebagai bilangan bulat komplemen-2. Ada dua kasus khusus yang perlu dipertimbangkan. Pertama, pertimbangkan $A = 0$. Dalam hal itu, untuk representasi 8-bit:

$$\begin{array}{rcl}
 0 & = & 00000000 \\
 \text{bitwise complement} & = & \underline{\underline{11111111}} + \quad \text{(Komplemen-1)} \\
 & & \quad \quad \quad 1 \\
 & & 100000000 = 0 \quad \text{(Komplemen-2)}
 \end{array}$$

Ada pelaksanaan posisi bit paling signifikan, yang diabaikan. Hasilnya adalah negasi 0 adalah 0, sebagaimana mestinya. Kasus kasus kedua lebih merupakan masalah. Jika mengambil negasi dari pola bit 1 diikuti oleh $n - 1$ nol, maka akan mendapatkan nomor yang sama. Misalnya, untuk *word* 8-bit berikut:

$$\begin{array}{rcl}
 -128 & = & 10000000 \quad \text{(Komplemen-2)} \\
 \text{bitwise complement} & = & \underline{\underline{01111111}} + \\
 & & \quad \quad \quad 1 \\
 & & 10000000 = -128
 \end{array}$$

Anomali tersebut itu tidak dapat dihindari. Jumlah pola bit yang berbeda dalam *word* n-bit adalah 2^n , yang merupakan angka genap. Mewakili bilangan bulat positif dan negatif dan 0. Jika jumlah yang sama bilangan positif dan negatif diwakili (tanda besarnya), maka ada dua representasi untuk 0. Jika hanya ada satu representasi dari 0 (dua pasangan komplemen), maka ada harus berupa angka negatif dan positif yang tidak sama jumlahnya. Dalam kasus dua pasangan komplemen, untuk panjang n-bit, ada representasi untuk -2^{n-1} tetapi tidak untuk $+2^{n-1}$.

B. Penjumlahan dan Pengurangan

Gambar 6.3 mengilustrasikan operasi penjumlahan dalam komplemen-2. Penjumlahan berlangsung seolah-olah kedua angka tersebut adalah bilangan *unsigned integer*. Empat contoh pertama menggambarkan operasi berjalan normal. Jika hasil operasi positif, didapat bilangan positif komplemen-2, yang sama dengan dalam bentuk bilangan *unsigned integer*. Jika hasil operasi negatif,

didapat bilangan negatif dalam bentuk komplemen-2. Perhatikan bahwa, dalam beberapa kasus, ada sedikit *carry* di luar akhir *word*, yang diabaikan.

(a) $(-7) + (+5)$	b. $(-4) + (-4)$
$ \begin{array}{r} 0010 = -7 \\ +\underline{1001} = 5 \\ \hline 1011 = -2 \end{array} $	$ \begin{array}{r} 1100 = -4 \\ +\underline{0010} = 4 \\ \hline 10000 = 0 \end{array} $
(c) $(+3) + (+4)$	(d) $(-7) + (-6)$
$ \begin{array}{r} 0011 = 3 \\ +\underline{0100} = 4 \\ \hline 0111 = 7 \end{array} $	$ \begin{array}{r} 1100 = -4 \\ +\underline{1111} = -1 \\ \hline 11011 = -5 \end{array} $
(e) $(5) + (4)$	(f) $(-7) + (-6)$
$ \begin{array}{r} 0101 = 5 \\ +\underline{0100} = 4 \\ \hline 1001 = (\text{overflow}) \end{array} $	$ \begin{array}{r} 0010 = -7 \\ +\underline{1001} = -6 \\ \hline 10110 = (\text{overflow}) \end{array} $

Sumber: (Stallings, 2016)

Gambar 6.3. Penambahan Bilangan dengan Representasi Komplemen-2

$ \begin{array}{r} 0010 = 2 \\ +\underline{1001} = -7 \\ \hline 1011 = -5 \end{array} $	$ \begin{array}{r} 0101 = 5 \\ +\underline{1110} = -2 \\ \hline 10011 = 3 \end{array} $
(a) $M = 2 = 0010$ $S = 7 = 0111$ $-S = -7 = 1000$	(b) $M = 5 = 0101$ $S = 2 = 0010$ $-S = -2 = 1110$
$ \begin{array}{r} 1011 = -5 \\ +\underline{1110} = -2 \\ \hline 11011 = -7 \end{array} $	$ \begin{array}{r} 0101 = 5 \\ +\underline{0010} = 2 \\ \hline 0111 = 7 \end{array} $
(c) $M = -5 = 1011$ $S = 2 = 0010$ $-S = -2 = 1110$	(d) $M = 5 = 0101$ $S = -2 = 1110$ $-S = 2 = 0010$
$ \begin{array}{r} 0111 = 7 \\ +\underline{0111} = 7 \\ \hline 1110 = (\text{overflow}) \end{array} $	$ \begin{array}{r} 0010 = -6 \\ +\underline{1001} = -4 \\ \hline 10110 = (\text{overflow}) \end{array} $
(e) $M = 7 = 0111$ $S = -7 = 1001$ $-S = 7 = 0111$	(f) $M = -6 = 1010$ $S = 4 = 0100$ $-S = -4 = 1100$

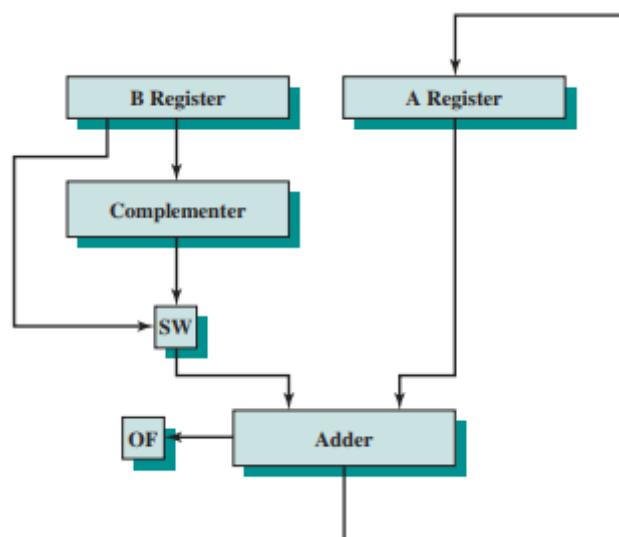
Sumber: (Stallings, 2016)

Gambar 6.4. Pengurangan Bilangan dengan Representasi Komplemen-2 (M-S)

Pada suatu penjumlahan, hasilnya mungkin lebih besar daripada yang bisa ditampung dalam ukuran *word* yang digunakan. Kondisi ini disebut *overflow*. Ketika terjadi *overflow*, ALU harus memberi sinyal kondisi ini sehingga tidak ada upaya yang dilakukan untuk menggunakan hasilnya. Untuk mendeteksi *overflow*, perhatikan aturan berikut: jika dua angka ditambahkan, dan keduanya positif atau keduanya negatif, maka *overflow* terjadi jika dan hanya jika hasilnya memiliki *sign-bit* yang berlawanan. *Overflow* dapat terjadi bila terdapat *carry*.

Pengurangan mudah ditangani dengan aturan berikut: untuk mengurangi satu bilangan (*subtrahend*) dari yang lain (*minuend*), ambil komplementen-2 (negasi) dari *subtrahend* dan tambahkan ke *minuend*. Gambar 6.4 adalah contoh penjumlahan yang menghasilkan *overflow*.

Gambar 6.5 menunjukkan jalur data dan elemen perangkat keras yang diperlukan untuk menyelesaikan penjumlahan dan pengurangan. Elemen utama adalah adder, yang menerima dua angka untuk penjumlahan dan menghasilkan hasil penjumlahan dan indikasi *overflow*.



OF Overflow bit
SW Switch (memilih addition atau subtraction)

Sumber: (Stallings, 2016)

Gambar 6.5. Blok Diagram Hardware untuk Penjumlahan dan Pengurangan

Penambahan biner memperlakukan kedua bilangan tersebut sebagai bilangan unsigned integer. Sebagai tambahan, dua angka disajikan ke adder dari dua register, yang dalam hal ini ditunjuk sebagai register A dan B. Hasilnya mungkin disimpan di salah satu register ini atau di register ketiga. Indikasi *overflow* disimpan dalam bendera *overflow* 1-bit OF (0 = tidak ada *overflow*; 1 = terjadi *overflow*). Untuk pengurangan, subtrahend (register B) dilewatkan melalui *Complementer* sehingga komplementen-2 nya diinputkan ke *adder*. Sinyal kontrol SW diperlukan untuk mengontrol apakah *Complementer* digunakan atau tidak, bergantung pada apakah operasinya adalah penjumlahan atau pengurangan.

C. Perkalian

Unsigned Integers

Gambar 6.6 mengilustrasikan operasi perkalian seperti yang dilakukan secara manual dengan menggunakan kertas dan pensil. Beberapa hal yang digaris bawahi adalah:

1. Perkalian melibatkan tingkatan *partial product*, satu untuk setiap digit dalam *multiplier*. *Partial product* ini kemudian dijumlahkan untuk menghasilkan produk akhir.

1011	Multiplicand (11)
x 1101	Multiplier (13)
1011	
0000	
1011	
+ 1011	
<hr/> 10001111	Product (143)

}

Partial product

Sumber: (Stallings, 2016)

Gambar 6.6. Perkalian Biner

2. *Partial product* mudah didefinisikan. Ketika bit multiplier adalah 0, *partial product* adalah 0. Ketika multiplier adalah 1, *partial product* adalah multiplicand.
3. Total product didapat dengan menjumlahkan *partial product*. Untuk operasi ini, setiap *partial product* berturut-turut digeser satu posisi ke kiri relatif terhadap *partial product* sebelumnya.
4. Perkalian dua bilangan bulat biner n-bit menghasilkan produk dengan panjang hingga 2^n bit (misalnya: $11 \times 11 = 1001$).

Dibandingkan dengan pendekatan manual dengan kertas-dan-pensil, ada beberapa hal yang dapat dilakukan untuk membuat penggandaan terkomputerisasi menjadi lebih efisien. Pertama, dapat melakukan penambahan berjalan pada *partial product* daripada menunggu sampai akhir. Ini menghilangkan kebutuhan untuk penyimpanan semua *partial product*; register lebih sedikit diperlukan. Kedua, dapat menghemat waktu untuk menghasilkan beberapa *partial product*. Untuk setiap 1 pada *multiplier*, operasi tambah dan shift diperlukan; tetapi untuk setiap 0, hanya diperlukan pergeseran.

Two's Complement Multiplication

Pada bahasan sebelumnya, penjumlahan dan pengurangan dapat dilakukan pada angka dalam notasi komplemen-2 berpasangan dengan memperlakukannya sebagai bilangan bulat tak bertanda. Misalnya pada penjumlahan berikut,

$$\begin{array}{r} 1001 \\ +0011 \\ \hline 1100 \end{array}$$

Jika angka-angka ini diasumsikan sebagai *unsigned integer*, maka dapat dilakukan: menambahkan 9 (1001) dengan 3 (0011) untuk mendapatkan 12 (1100). Namun jika diasumsikan sebagai bilangan *signed integer*, menjadi menambahkan -7 (1001) dengan 3 (0011) untuk mendapatkan -4 (1100). Namun skema sederhana ini tidak dapat berlaku pada Perkalian. Sebagai contoh (Gambar 6.7): mengalikan 11 (1011) dengan 13 (1101) untuk mendapatkan 143 (10001111). Jika dua angka tersebut diasumsikan sebagai komplemen, maka operasinya adalah -5 (1011) dikalikan dengan -3 (1101) sama dengan -113 (10001111). Contoh ini menunjukkan bahwa perkalian langsung tidak akan berfungsi jika salah satu atau kedua dari *multiplicand* dan *multiplier* negatif.

1011	Multiplicand (11)
x 1101	Multiplier (13)
00001011	
00000000	
00101100	
+01011000	
<hr/> 10001111	Product (143)

}

Partial product

Gambar 6.7. Perkalian dua bilangan 4-bit *Unsigned Integer* menghasilkan 8 bit

Selanjutnya, penggandaan angka biner dengan 2^n dilakukan dengan menggeser angka itu ke n bit kiri. Dengan mengingat hal ini, Gambar 6.8 menampilkan kembali Gambar 6.7 untuk membuat generasi *partial product* multiplikasi secara eksplisit.

Satu-satunya perbedaan pada Gambar 6.8 adalah: ia mengakui bahwa *partial product* harus dilihat sebagai angka 2^n -bit yang dihasilkan dari *multiplicand* n-bit. Dengan demikian, sebagai bilangan *unsigned integer*, 4-bit *multiplicand* 1011 disimpan dalam *word* 8-bit sebagai 00001011. Setiap *partial product* terdiri dari angka ini yang bergeser ke kiri sebanyak satu digit dari sebelumnya, dengan posisi yang tidak dihuni di sebelah kanan diisi dengan nol (misalnya pergeseran ke kiri dari dua tempat menghasilkan 00101100).

$ \begin{array}{r} 1011 \quad (9) \\ \times 1101 \quad (3) \\ \hline 00001001 \quad 1001 \times 2^0 \\ +00010010 \quad 1001 \times 2^1 \\ \hline 00011011 \quad (27) \end{array} $ <p>(a). Unsigned Integer</p>	$ \begin{array}{r} 1011 \quad (-7) \\ \times 1101 \quad (3) \\ \hline 11111001 \quad (-7) \times 2^0 \\ +11110010 \quad (-7) \times 2^1 \\ \hline 11101011 \quad (-21) \end{array} $ <p>(a). Integer Komplemen-2</p>
---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

Sumber: (Stallings, 2016)

Gambar 6.8. Perbandingan Perkalian *Unsigned Integer* dengan Komplemen-2

Sekarang dapat ditunjukkan bahwa perkalian langsung tidak akan bekerja jika *multiplicand* negatif. Masalahnya adalah bahwa setiap kontribusi dari perkalian negatif sebagai *partial product* harus berupa angka negatif pada bidang 2^n -bit; bit tanda dari *partial product* harus berbaris. Ini ditunjukkan pada Gambar 6.8, yang menunjukkan perkalian biner 1001 dengan 0011. Jika ini diperlakukan sebagai bilangan *unsigned integer*, perkalian menghasilkan $9 * 3 = 27$. Namun, jika 1001 diinterpretasikan sebagai nilai komplemen-dua, menjadi -7, maka setiap *partial product* harus merupakan jumlah negatif dari 2^n (8) bit, seperti yang ditunjukkan pada Gambar 6.8b. Perhatikan bahwa ini dilakukan dengan mengisi setiap *partial product* ke kiri dengan biner 1s.

Jika pengali negatif, perkalian langsung juga tidak akan berfungsi. Alasannya adalah bahwa bit pengganda tidak lagi sesuai dengan pergeseran atau perkalian yang harus terjadi. Sebagai contoh, angka desimal 4-bit -3 ditulis 1101 dalam komplemen-2. Jika hanya mengambil *partial product* berdasarkan setiap posisi bit, maka akan memiliki korespondensi berikut:

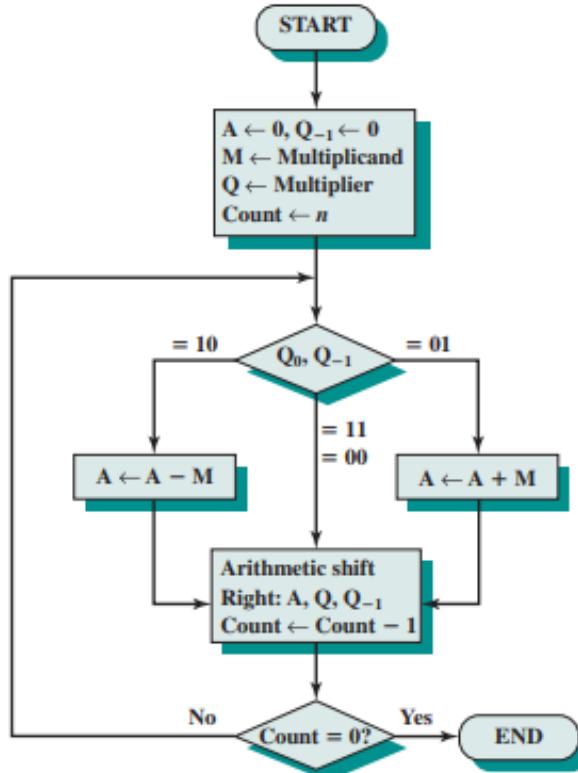
$$1101 \leftrightarrow -(1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0) = -(2^3 + 2^2 + 2^0)$$

Bahkan, yang diinginkan adalah $-(2^3 + 2^2 + 2^0)$. Jadi pengganda ini tidak dapat digunakan secara langsung dengan cara yang telah di gambarkan. Ada sejumlah jalan keluar dari dilema ini. Salah satunya adalah untuk mengubah baik pengali dan multiplikasi ke angka positif, melakukan penggandaan, dan kemudian mengambil hasil komplemen-2 jika dan hanya jika tanda dari dua angka asli berbeda. Pelaksana lebih suka menggunakan teknik yang tidak memerlukan langkah transformasi akhir ini. Salah satu yang paling umum adalah *Algoritma Booth*. Algoritma ini juga memiliki manfaat mempercepat proses multiplikasi, relatif terhadap pendekatan yang lebih mudah.

Gambar 6.9 adalah *flowchart* dari *Algoritma Booth*. Seperti sebelumnya, pengali dan multiplikasi masing-masing ditempatkan dalam register Q dan M. Ada juga register 1-bit yang ditempatkan secara logika di sebelah kanan bit paling signifikan (Q0) dari register Q dan ditunjuk Q-1.

Hasil perkalian akan muncul di register A dan Q. A dan Q-1 diinisialisasi ke 0. Seperti sebelumnya, logika kontrol memindai bit pengali satu per satu. Sekarang, ketika setiap bit diperiksa, bit di sebelah kanannya juga diperiksa. Jika kedua bit itu sama (1-1 atau 0-0), maka semua bit dari register A, Q, dan Q-1 digeser ke bit 1 yang benar. Jika dua bit berbeda, maka multiplicand ditambahkan atau dikurangi dari register A, tergantung pada apakah kedua bit tersebut 0-1 atau 1-0.

Setelah penambahan atau pengurangan, pergeseran yang tepat terjadi. Dalam kedua kasus tersebut, pergeseran kanan sedemikian rupa sehingga bit paling kiri dari A, yaitu An-1, tidak hanya dialihkan ke An-2, tetapi juga tetap di An-1. Ini diperlukan untuk mempertahankan tanda nomor di A dan Q. Ini dikenal sebagai pergeseran aritmatika, karena mempertahankan bit tanda.



Sumber: (Stallings, 2016)

Gambar 6.9. Flowchart Booth's Algorithm untuk Perkalian Komplemen-2

A	Q	Q ₋₁	M		Initial value
0000	0011	0	0111		
1001	0011	0	0111	A ← A-M	First cycle
1100	1001	1	0111	Shift	
1110	0100	1	0111	Shift	Second cycle
0101	0100	1	0111	A ← A+M	
0011	1010	0	0111	Shift	Third cycle
0001	0101	0	0111	Shift	
					Fourth cycle

Sumber: (Stallings, 2016)

Gambar 6.10. Contoh Booth's Algorithm (7x3)

Gambar 6.10 menunjukkan urutan kejadian dalam Algoritma Booth untuk perkalian 7 dengan 3. Secara lebih ringkas, operasi yang sama digambarkan pada Gambar 6.8a. Gambar 6.11 memberikan contoh lain dari algoritma. Seperti dapat dilihat, ini bekerja dengan kombinasi angka positif dan

negatif. Perhatikan juga efisiensi algoritma. Blok 1s atau 0s dilewati, dengan rata-rata hanya satu penambahan atau pengurangan per blok.

$ \begin{array}{r} 0111 \\ \times 1101 \quad (0) \\ \hline 11111001 \quad 1-0 \\ 0000000 \quad 1-1 \\ \hline 000111 \quad 0-1 \\ \hline 00010101 \quad (21) \end{array} $ <p>(a). $7 \times 3 = 21$</p>	$ \begin{array}{r} 0111 \\ \times 1101 \quad (0) \\ \hline 11111001 \quad 1-0 \\ 0000111 \quad 0-1 \\ \hline 111001 \quad 1-0 \\ \hline 11101011 \quad (-21) \end{array} $ <p>(b). $7 \times (-3) = (-21)$</p>
$ \begin{array}{r} 0111 \\ \times 1101 \quad (0) \\ \hline 00000111 \quad 1-0 \\ 0000000 \quad 1-1 \\ \hline 111001 \quad 0-1 \\ \hline 11101011 \quad (21) \end{array} $ <p>(c). $(-7) \times 3 = (-21)$</p>	$ \begin{array}{r} 0111 \\ \times 1101 \quad (0) \\ \hline 00000111 \quad 1-0 \\ 1111001 \quad 0-1 \\ \hline 000111 \quad 1-0 \\ \hline 00010101 \quad (21) \end{array} $ <p>(d). $(-7) \times (-3) = (21)$</p>

Sumber: (Stallings, 2016)

Gambar 6.11. Contoh Penggunaan Booth's Algorithm

D. Pembagian (*Division*)

Gambar 6.12 menunjukkan contoh pembagian panjang bilangan bulat biner yang tidak ditandatangani. Sangat instruktif untuk menggambarkan proses secara rinci. Pertama, bit dari bilangan yang dibagi (*dividen*) diperiksa dari kiri ke kanan, sampai himpunan bit yang diperiksa mewakili angka yang lebih besar dari atau sama dengan pembagi (*divisor*). Sampai peristiwa ini terjadi, 0s ditempatkan dalam hasil bagi dari kiri ke kanan. Ketika peristiwa itu terjadi, 1 ditempatkan di hasil bagi dan pembagi dikurangi dari dividen parsial. Hasilnya disebut sebagai sisa parsial (*partial remainder*).

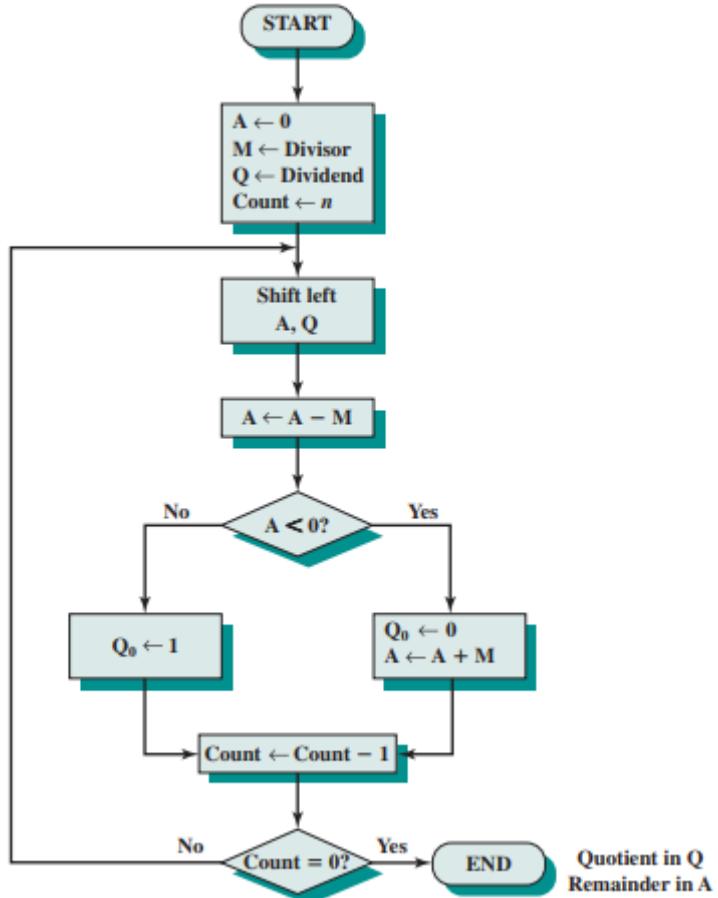
Divisor → 1011	$ \begin{array}{r} 0000 \quad 1101 \quad \leftarrow \text{Quotient} \\ \hline 1011 / \quad 1001 \quad 0011 \quad \leftarrow \text{Dividend} \\ \quad \quad \quad \underline{101 \quad 1} \end{array} $	Parsial remainder → 0011	$ \begin{array}{r} 10 \quad 10 \\ \hline 00 \quad 1111 \end{array} $	\rightarrow	$ \begin{array}{r} 1011 \\ \hline 100 \quad \leftarrow \text{Reminder} \end{array} $
----------------------	---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	--------------------------------	----------------------------------------------------------------------------	---------------	--------------------------------------------------------------------------------------------

Sumber: (Stallings, 2016)

Gambar 6.12. Perkalian Biner

Dari titik ini, pembagian mengikuti pola siklik. Pada setiap siklus, bit tambahan dari *dividen* ditambahkan ke sisa sebagian (*partial remainder*) hingga hasilnya lebih besar dari atau sama dengan *divisor*. Seperti sebelumnya, pembagi dikurangkan dari angka ini untuk menghasilkan sisa parsial baru. Proses berlanjut sampai semua bit dividen habis. Gambar 6.13 menunjukkan algoritma mesin yang sesuai dengan proses pembagian panjang. Pembagi ditempatkan dalam register M, dividen dalam

register Q. Pada setiap langkah, register A dan Q digeser ke kiri 1 bit. M dikurangkan dari A untuk menentukan apakah A membagi sisa sebagian. Jika ya, maka Q_0 mendapat 1bit. Jika tidak, Q_0 mendapat 0 bit dan M harus ditambahkan kembali ke A untuk mengembalikan nilai sebelumnya. Hitungan kemudian dikurangi, dan proses berlanjut untuk n langkah. Pada akhirnya, hasil bagi berada di register Q dan sisanya ada di register A.



Sumber: (Stallings, 2016)

Gambar 6.13. Flowchart untuk Pembagian Biner tidak bertanda

Proses ini dapat diperluas ke angka negatif. Diberikan satu pendekatan untuk dua nomor komplemen. Contoh dari pendekatan ini ditunjukkan pada Gambar 6.14.

Algoritma ini mengasumsikan bahwa *divider* V dan *dividen* D adalah positif dan bahwa $|V| > |D|$. Jika $|V| = |D|$, maka hasil bagi $Q = 1$ dan sisanya $R = 0$. Jika $|V| > |D|$, lalu $Q = 0$ dan $R = D$. Algoritma dapat diringkas sebagai berikut:

1. Muat dua pelengkap dari pembagi ke dalam register M; artinya, register M berisi negatif dari pembagi. Muat dividen ke dalam register A, Q. Dividen harus dinyatakan sebagai angka positif 2n-bit. Jadi, misalnya, 0111 4-bit menjadi 00000111.
2. Shift A, Q ke kiri posisi 1 bit.
3. Lakukan $A \leftarrow A - M$. Operasi ini mengurangi pembagi dari isi A.
4. Dari hasilnya:
 - a. Jika hasilnya tidak negatif (bit paling signifikan dari $A=0$), maka atur $Q_0 \leftarrow 1$.
 - b. Jika hasilnya negatif (bit paling signifikan dari $A=1$), maka atur $Q_0 \leftarrow 0$ dan kembalikan nilai A sebelumnya.
5. Ulangi langkah 2 hingga 4 sebanyak posisi bit pada Q.
6. Sisanya berada di A dan hasil bagi berada di Q.

A	Q	
0000	0111	Initial value
0000	1110	Shift
<u>1101</u>		Menggunakan komplemen dari 0011 untuk pengurangan
1101		Kurangi
0000	1110	Restore, set $Q_0 = 0$
0000	1100	Shift
<u>1101</u>		
1110		Kurangi
0001	1100	Restore, set $Q_0 = 0$
0011	1000	Shift
<u>1101</u>		
0000	1001	Restore, set $Q_0 = 1$
0001	0010	Shift
<u>1101</u>		
1110		Kurangi
0001	0011	Restore, set $Q_0 = 0$

Sumber: (Stallings, 2016)

Gambar 6.14. Contoh restoring pembagian komplemen-2

Untuk menangani angka negatif, maka sisanya ditentukan oleh:

$$D = Q * V + R$$

Artinya, sisanya adalah nilai R yang diperlukan untuk persamaan sebelumnya menjadi valid. Pertimbangkan contoh-contoh pembagian bilangan bulat berikut dengan semua kemungkinan kombinasi tanda D dan V:

$$\begin{array}{llll} D = 7 & V = 3 & \Rightarrow & Q = 2 \quad R = 1 \\ D = 7 & V = -3 & \Rightarrow & Q = -2 \quad R = 1 \\ D = -7 & V = 3 & \Rightarrow & Q = -2 \quad R = -1 \\ D = -7 & V = -3 & \Rightarrow & Q = 2 \quad R = -1 \end{array}$$

Dapat disimpulkan dari Gambar 6.14 bahwa $(-7)/(3)$ dan $(7)/(-3)$ menghasilkan sisa yang berbeda. dapat dilihat bahwa besaran Q dan R tidak terpengaruh oleh tanda bilangan (magnitudes) input dan bahwa tanda bilangan Q dan R mudah diturunkan dari tanda bilangan D dan V. Secara khusus, tanda (R) = tanda (D) dan tanda (Q) = tanda (D) * tanda (V). Oleh karena itu, salah satu cara untuk melakukan pembagian komplemen-2 adalah untuk mengubah *operand* menjadi nilai-nilai yang tidak bertanda (*unsigned*) dan, pada akhirnya, untuk memperhitungkan tandabilangan dengan komplementasi jika diperlukan. Ini adalah metode pilihan untuk memulihkan algoritma pembagian

6.4. Representasi Floating Point

Notasi *fix-point* dapat digunakan untuk mewakili kisaran bilangan bulat positif dan negatif yang berpusat pada atau dekat dengan 0. Dengan mengasumsikan titik biner atau radix tetap, format ini memungkinkan representasi angka dengan komponen fraksional.

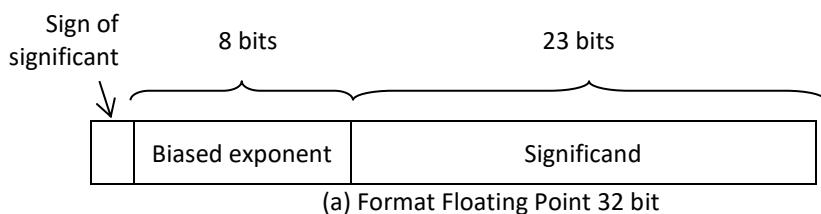
Pendekatan ini memiliki keterbatasan. Angka yang sangat besar tidak dapat diwakili, juga untuk fraksi yang sangat kecil. Selanjutnya, bagian pecahan hasil bagi dalam pembagian dua angka besar bisa hilang.

Untuk angka desimal, batasan ini diatasi dengan menggunakan notasi ilmiah. Misalnya 976.000.000.000.000 dapat diwakili sebagai $9,76 \times 10^{14}$, dan 0,000000000000976 dapat diwakili sebagai $9,76 \times 10^{-14}$. Pada dasarnya, adalah menggeser titik desimal secara dinamis ke lokasi yang

nyaman dan menggunakan eksponen 10 untuk melacak titik desimal itu. Ini memungkinkan kisaran angka yang sangat besar dan sangat kecil untuk diwakilinya hanya dengan beberapa digit. Pendekatan yang sama ini dapat diambil dengan angka biner, dalam bentuk $\pm S * B^{\pm E}$

Bilangan ini dapat disimpan dalam *word* biner dengan tiga bidang:

1. Sign: plus atau minus
2. Signifikan S
3. Eksponen E



(a) Format Floating Point 32 bit

Basis B
 $1.10100001 \times 2^{10100} = 0\ 10010011\ 10100010000000000000000000000000 = 1.6328125 \times 2^{20}$
 $-1.10100001 \times 2^{10100} = 1\ 10010011\ 10100010000000000000000000000000 = -1.6328125 \times 2^{20}$
 $1.10100001 \times 2^{-10100} = 0\ 10010011\ 10100010000000000000000000000000 = 1.6328125 \times 2^{-20}$
 $-1.10100001 \times 2^{-10100} = 1\ 10010011\ 10100010000000000000000000000000 = -1.6328125 \times 2^{-20}$

(b) Contoh

Sumber: (Stallings, 2016)

Gambar 6.15. Format Floating Point 32 bit

Gambar 6.15a menunjukkan format *floating-point* 32-bit yang khas. Bit paling kiri menyimpan tanda angka (0 = positif, 1 = negatif). Prinsip-prinsip yang digunakan dalam merepresentasikan bilangan *floating point* dijelaskan pada contoh seperti gambar 6.15b. Basis B adalah implisit dan tidak perlu disimpan karena sama untuk semua angka. Dalam hal ini, karena bilangan adalah biner, maka basis bilangan adalah 2.

Nilai eksponen disimpan dalam 8 bit berikutnya. Representasi yang digunakan dikenal sebagai representasi bias, disebut juga sebagai eksponen terbiasa. Nilai bias, dikurangkan dari bidang untuk mendapatkan nilai eksponen sejati. Biasanya, biasnya sama dengan $(2^{k-1}-1)$, di mana k adalah jumlah bit dalam eksponen biner. Dalam kasus ini, bidang 8-bit menghasilkan angka 0 hingga 255. Dengan bias 127 (2^7-1), nilai eksponen sebenarnya berada di kisaran -127 hingga +128.

Tabel 6.2 menunjukkan representasi bias untuk bilangan bulat 4-bit. Ketika bit representasi bias diperlakukan sebagai bilangan *unsigned integer*, besaran relatif dari angka tersebut tidak berubah. Misalnya, dalam representasi bias dan *unsigned*, bilangan terbesar adalah 1111 dan bilangan terkecil adalah 0000. Ini tidak berlaku untuk representasi *sign-magnitude* atau komplement-2. Keuntungan dari representasi bias adalah bahwa bilangan *floating-point* nonnegatif dapat diperlakukan sebagai bilangan bulat untuk tujuan perbandingan. Bagian akhir dari *word* (dalam kasus ini 23 bit) adalah signifikansi.

Setiap bilangan *floating-point* dapat diekspresikan dengan banyak cara. Berikut ini adalah bentuk ekivalen, di mana signifikansinya dinyatakan dalam bentuk biner:

$$\begin{aligned} 11000 &= 0,110 \times 2^5 \\ 11000 &= 110 \times 2^2 \\ 11000 &= 0,0110 \times 2^6 \end{aligned}$$

Untuk menyederhanakan operasi pada bilangan *floating-point*, biasanya bilangan tersebut dinormalisasi. Bilangan normal adalah angka di mana digit paling signifikan dari bidang signifikan adalah bukan nol. Untuk representasi basis 2, angka normal adalah satu di mana bit paling signifikan dari *significand* adalah satu. Seperti yang disebutkan, konvensi yang khas adalah bahwa ada satu bit di sebelah kiri titik radix. Jadi, bilangan nol normal adalah satu dalam bentuk:

$$\pm 1.\text{bbb...b} \times 2^{\pm E}$$

b adalah digit biner (0 atau 1).

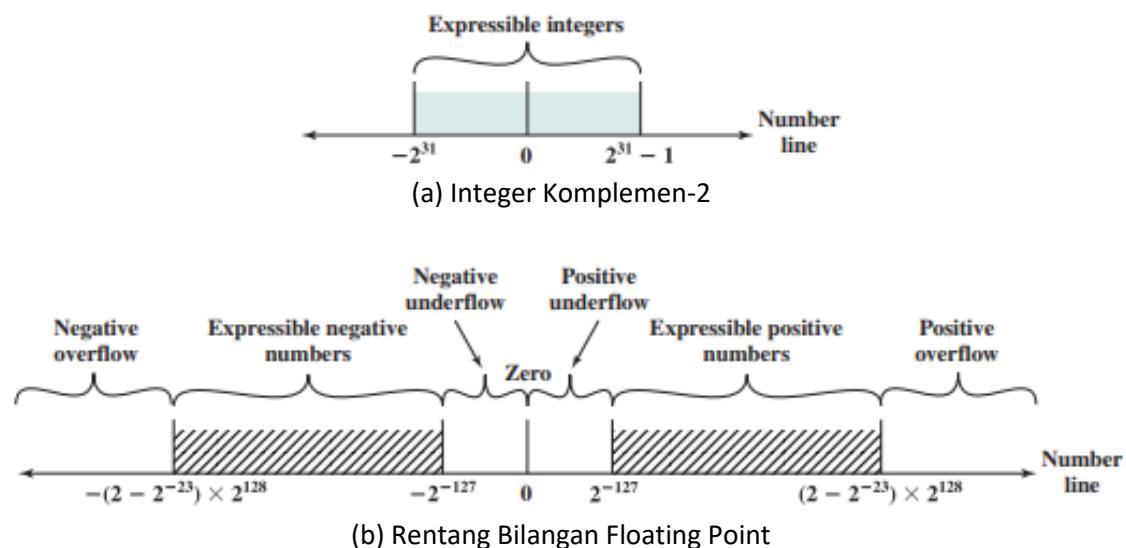
Karena bit yang paling signifikan selalu satu, maka tidak perlu menyimpan bit ini (dinyatakan secara implisit). Dengan demikian, bidang 23-bit digunakan untuk menyimpan signifikansi 24-bit dengan angka yang paling signifikan dinyatakan secara implisit.

Gambar 6.16b memberikan beberapa contoh untuk format ini. Untuk setiap contoh, di sebelah kiri adalah nomor biner; di tengah adalah pola bit yang sesuai; di sebelah kanan adalah nilai desimal. Perhatikan fitur-fitur berikut:

- Tanda disimpan di bit pertama *word*.
- Bit pertama yang paling signifikan (MSB) selalu 1 dan tidak perlu disimpan dalam bidang *significand*.
- Nilai 127 ditambahkan ke eksponen sebenarnya untuk disimpan dalam bidang *biased exponent*.
- Basisnya 2.

Sebagai perbandingan, Gambar 6.16 menunjukkan kisaran bilangan yang dapat direpresentasikan dalam *word* 32-bit. Menggunakan dua representasi integer komplemen, semua bilangan bulat dari -2^{31} sampai $2^{31}-1$ dapat diwakili, dengan total 2^{32} bilangan yang berbeda. Dengan contoh format *floating point* pada Gambar 6.16b, rentang angka berikut dimungkinkan:

- Angka negatif antara $-(2 - 2^{-23}) \times 2^{128}$ dan -2^{-127}
- Angka positif antara 2^{-127} dan $(2 - 2^{-23}) \times 2^{128}$



Sumber: (Stallings, 2016)

Gambar 6.16. Bilangan yang disajikan pada format 32 bit

Lima bilangan pada garis bilangan tidak termasuk dalam rentang ini adalah:

1. Angka negatif kurang dari $-(2 - 2^{-23}) \times 2^{128}$, disebut negatif *overflow*
2. Angka negatif lebih besar dari 2^{-127} , yang disebut negatif *underflow*
3. Nol

4. Angka positif kurang dari 2^{-127} , disebut aliran bawah positif
5. Angka positif lebih besar dari $(2 - 2^{-23}) \times 2^{128}$, disebut positive overflow

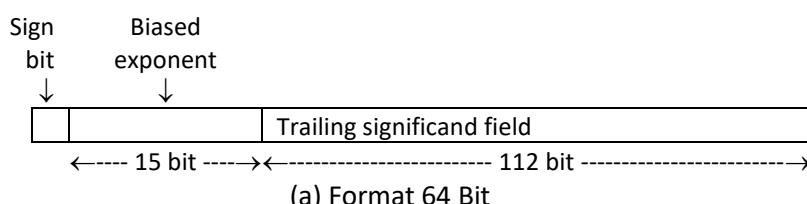
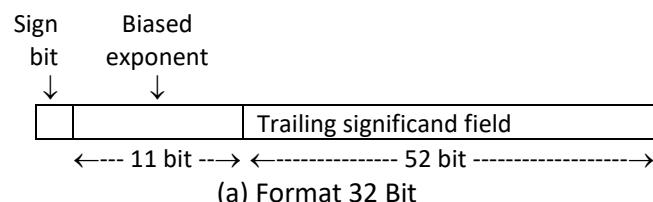
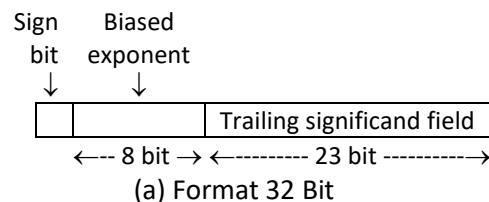
Overflow terjadi ketika operasi aritmatika menghasilkan nilai absolut yang lebih besar daripada yang dapat diekspresikan dengan eksponen 128 (misalnya: $2^{120} \times 2^{100} = 2^{220}$). *Underflow* terjadi ketika besarnya fraksional terlalu kecil (misalnya: $2^{-120} \times 2^{-100} = 2^{-220}$). *Underflow* adalah masalah yang kurang serius karena hasilnya secara umum dapat diasumsikan sama dengan 0.

Standar IEEE untuk Representasi *Floating-point* Biner

Representasi *floating-point* didefinisikan dalam Standar IEEE 754, mulai diadopsi pada tahun 1985 dan direvisi pada tahun 2008. Standar ini dikembangkan untuk memfasilitasi portabilitas program dari satu prosesor ke yang lain dan untuk mendorong pengembangan program yang canggih, yang berorientasi numerik. Standar ini telah diadopsi secara luas dan digunakan pada hampir semua prosesor dan koprosesor aritmatika kontemporer. IEEE 754-2008 mencakup representasi *floating-point* biner dan desimal. Pembahasan ini hanya melibatkan representasi biner.

IEEE 754-2008 menetapkan jenis format *floating-point* sebagai berikut:

1. Format aritmatika: Semua operasi wajib yang ditentukan oleh standar didukung oleh format. Format dapat digunakan untuk mewakili *operand* titik-mengambang atau hasil untuk operasi yang dijelaskan dalam standar.
2. Format dasar: Format ini mencakup lima representasi *floating-point*, tiga biner dan dua desimal, yang pengkodeannya ditentukan oleh standar, dan yang dapat digunakan untuk aritmatika. Setidaknya salah satu format dasar diimplementasikan dalam implementasi yang sesuai.
3. Format pertukaran: Suatu pengkodean biner dengan panjang tetap yang ditentukan yang memungkinkan pertukaran data antara berbagai *platform* dan yang dapat digunakan untuk penyimpanan (*extended precision*).



Sumber: (Stallings, 2016)
Gambar 6.17. Format Bilangan Foating Point IEEE 754

Tiga format dasar biner memiliki panjang 32 bit (*single precision*), 64 bit (*double precision*), dan 128 bit (*extended single precision*), dengan eksponen masing-masing 8, 11, dan 15 bit (Gambar 6.17). Tabel 6.3 merangkum karakteristik dari ketiga format tersebut. Dua format desimal dasar memiliki panjang bit 64 dan 128 bit. Semua format dasar juga merupakan tipe format aritmatika (dapat digunakan untuk operasi aritmatika) dan tipe format *interchange* (platform independent).

Tabel 6.3. Parameter Format IEEE 754

Parameter	Format		
	32 bit	64 bit	128 bit
Lebar data (bit)	32	64	128
Lebar exponent (bit)	8	11	15
Bias eksponen	127	1023	16383
Maksimum eksponen	127	1023	16383
Minimum eksponen	-126	-1022	-16382
Perkiraan range bilangan normal (desimal)	$10^{-38}, 10^{+38}$	$10^{-308}, 10^{+308}$	$10^{-4932}, 10^{+4932}$
Lebar significant (bit)	23	52	112
Banyaknya eksponen	254	2046	32766
Banyaknya fraction	2^{23}	2^{52}	2^{112}
Banyaknya bilangan	1.98×2^{31}	1.99×2^{63}	1.99×2^{128}
Bilangan normal positif terkecil	2^{-126}	2^{-1022}	2^{-16362}
Bilangan normal positif terbesar	$2^{128} - 2^{104}$	$2^{1024} - 2^{971}$	$2^{1634} - 2^{16721}$
Subnormal magnitude terkecil	2^{-149}	2^{-1074}	2^{-16271}

Sumber: (Stallings, 2016)

Beberapa format lain ditentukan dalam standar. Format binary 16 hanya format pertukaran dan dimaksudkan untuk penyimpanan nilai ketika presisi yang lebih tinggi tidak diperlukan. Format biner {k} dan format desimal {k} adalah format *interchange* dengan total panjang k bit dan dengan panjang yang ditentukan untuk signifikansi dan eksponen. Format harus kelipatan 32 bit; dengan demikian format didefinisikan untuk k = 160, 192, dan sebagainya. Dua keluarga format ini juga merupakan format aritmatika. Selain itu, standar mendefinisikan format presisi yang diperluas, yang memperluas format dasar yang didukung dengan memberikan bit tambahan dalam eksponen (jangkauan yang diperluas) dan dalam signifikansi (perluasan presisi). Format yang tepat tergantung pada implementasi, tetapi standar menempatkan batasan tertentu pada panjang eksponen dan signifikansi. Format ini adalah tipe format aritmatika tetapi bukan tipe format interchange. Format yang diperluas harus digunakan untuk perhitungan menengah. Dengan ketepatan yang lebih besar, format yang diperluas mengurangi kemungkinan hasil akhir yang telah terkontaminasi oleh kesalahan pembulatan yang berlebihan; dengan jangkauan mereka yang lebih besar, mereka juga mengurangi kemungkinan *overflow* untuk membatalkan perhitungan yang hasil akhirnya akan dapat direpresentasikan dalam format dasar. Motivasi tambahan untuk format yang diperluas adalah memberikan beberapa manfaat format dasar yang lebih besar tanpa menimbulkan penalti waktu yang biasanya dikaitkan dengan presisi yang lebih tinggi.

Akhirnya, IEEE 754-2008 mendefinisikan format *extendable precision* sebagai format dengan presisi dan rentang yang ditentukan di bawah kendali pengguna. Format ini dapat digunakan untuk perhitungan menengah, tetapi standar tidak menempatkan kendala atau format atau panjang. Tabel 6.4 menunjukkan hubungan antara format yang ditentukan dan jenis format.

Tabel 6.4. Format IEEE 754-2008

Format	Format Type		
	Arithmetic	Basic	Intrchange
Binery 16			x
Binery 32	x	x	x
Binery 64	x	x	x
Binery 128	x	x	x
Binery (k) ($k=n*32, n>4$)	x		x
Desimal 64	x	x	x
Desimal 128	x	x	x
Desimal (k) ($k=n*32, n>4$)	x		x
Extenden precision	x		
Extendable precision	x		

Sumber: (Stallings, 2016)

Tidak semua pola bit dalam format IEEE ditafsirkan dengan cara yang biasa; sebaliknya, beberapa pola bit digunakan untuk mewakili nilai-nilai khusus. Tabel 6.5 menunjukkan nilai yang ditetapkan untuk berbagai pola bit. Nilai eksponen semua nol (0 bit) dan semua yang (1 bit) menentukan nilai khusus. Kelas-kelas angka berikut diwakili:

- Untuk nilai eksponen dalam kisaran 1 hingga 254 untuk format 32-bit, 1 hingga 2046 untuk format 64-bit, dan 1 hingga 16382, angka-angka titik mengambang normal bukan nol diwakili. Eksponen bias, sehingga kisaran eksponen adalah -126 hingga +127 untuk format 32-bit, dan seterusnya. Angka normal memerlukan 1 bit di sebelah kiri titik biner; bit ini tersirat, memberikan signifikansi 24-bit, 53-bit, atau 113-bit yang efektif. Karena salah satu bit tersirat, bidang yang sesuai dalam format biner disebut sebagai bidang Trailing Significand.
- Eksponen nol bersama dengan sebagian kecil dari nol mewakili positif atau negatif nol, tergantung pada bit tanda. Seperti yang disebutkan, adalah berguna untuk memiliki nilai tetap 0 yang diwakili.
- Eksponen semua yang bersama dengan sebagian kecil dari nol mewakili tak terhingga positif atau negatif, tergantung pada bit tanda. Ini juga berguna untuk memiliki representasi infinity. Ini tergantung pada pengguna untuk memutuskan apakah memperlakukan *overflow* sebagai kondisi kesalahan atau untuk membawa nilai dan melanjutkan dengan program apa pun yang sedang dijalankan.
- Eksponen nol bersama dengan fraksi bukan nol mewakili angka subnormal. Dalam hal ini, bit di sebelah kiri titik biner adalah nol dan eksponen sebenarnya adalah -126 atau -1022. Jumlahnya positif atau negatif tergantung pada bit tanda.
- Eksponen semua yang bersama dengan fraksi bukan nol diberi nilai NaN, yang berarti Bukan Angka, dan digunakan untuk memberi sinyal berbagai kondisi pengecualian.

Tabel 6.5 Interpretasi Bilangan Floating Point dari IEEE

(a) Binary 32 Format

	Sign	Biased Exponent	Fraction	Value
Positive zero	0	0	0	0
Negative zero	1	0	0	- 0
Plus infinity	0	All 1s	0	∞
Minus infinity	1	All 1s	0	- ∞
Quiet NaN	0 or 1	All 1s	$\neq 0$; first bit = 1	qNaN
Signaling NaN	0 or 1	All 1s	$\neq 0$; first bit = 0	sNaN
Positive normal	0	0	f	$2^{e-127}(1.f)$
Negative normal	1	0	f	- $2^{e-127}(1.f)$
Positive subnormal	0	0	$f \neq 0$	$2^{e-127}(0.f)$
Negative subnormal	1	0	$f \neq 0$	- $2^{e-127}(0.f)$

(b) Binary 64 Format

	Sign	Biased Exponent	Fraction	Value
Positive zero	0	0	0	0
Negative zero	1	0	0	- 0
Plus infinity	0	All 1s	0	∞
Minus infinity	1	All 1s	0	- ∞
Quiet NaN	0 or 1	All 1s	$\neq 0$; first bit = 1	qNaN
Signaling NaN	0 or 1	All 1s	$\neq 0$; first bit = 0	sNaN
Positive normal	0	$0 < e < 2047$	f	$2^{e-1023}(1.f)$
Negative normal	1	$0 < e < 2047$	f	- $2^{e-1023}(1.f)$
Positive subnormal	0	0	$f \neq 0$	$2^{e-1022}(0.f)$
Negative subnormal	1	0	$f \neq 0$	- $2^{e-1022}(0.f)$

(b) Binary 64 Format

	Sign	Biased Exponent	Fraction	Value
Positive zero	0	0	0	0
Negative zero	1	0	0	- 0
Plus infinity	0	All 1s	0	∞
Minus infinity	1	All 1s	0	- ∞
Quiet NaN	0 or 1	All 1s	$\neq 0$; first bit = 1	qNaN
Signaling NaN	0 or 1	All 1s	$\neq 0$; first bit = 0	sNaN
Positive normal	0	All 1s	f	$2^{e-16383}(1.f)$
Negative normal	1	All 1s	f	- $2^{e-16383}(1.f)$
Positive subnormal	0	0	$f \neq 0$	$2^{e-16383}(0.f)$
Negative subnormal	1	0	$f \neq 0$	- $2^{e-16383}(0.f)$

Sumber: (Stallings, 2016)

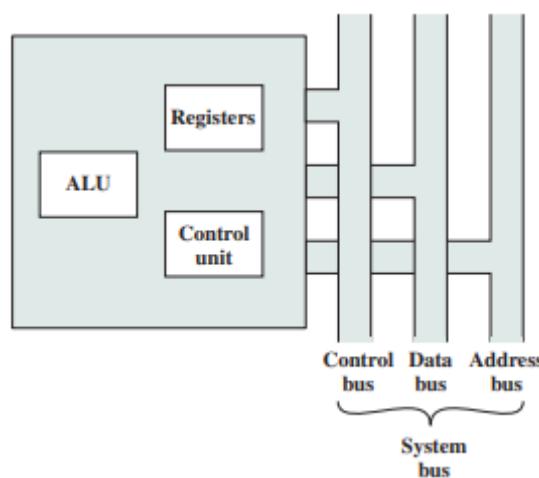
7. Struktur dan Fungsi Prosesor

7.1 Organisasi Prosesor

Berikut adalah hal-hal yang dilakukan oleh prosesor, yang akan menentukan organisinya:

- 1) Mengambil instruksi (*fetch instruction*): Prosesor membaca instruksi dari memori (register, *cache*, memori utama).
- 2) Interpretasi instruksi (*interpret instruction*): Instruksi diterjemahkan untuk menentukan tindakan apa yang diperlukan.
- 3) Mengambil data (*fetch data*): Pada saat eksekusi instruksi mungkin memerlukan untuk membaca data dari memori atau modul I/O.
- 4) Memproses data (*data processing*): Eksekusi suatu instruksi mungkin memerlukan beberapa operasi aritmatika atau logika pada data.
- 5) Menulis data (*write data*): Hasil eksekusi mungkin memerlukan penulisan data ke memori atau modul I/O.

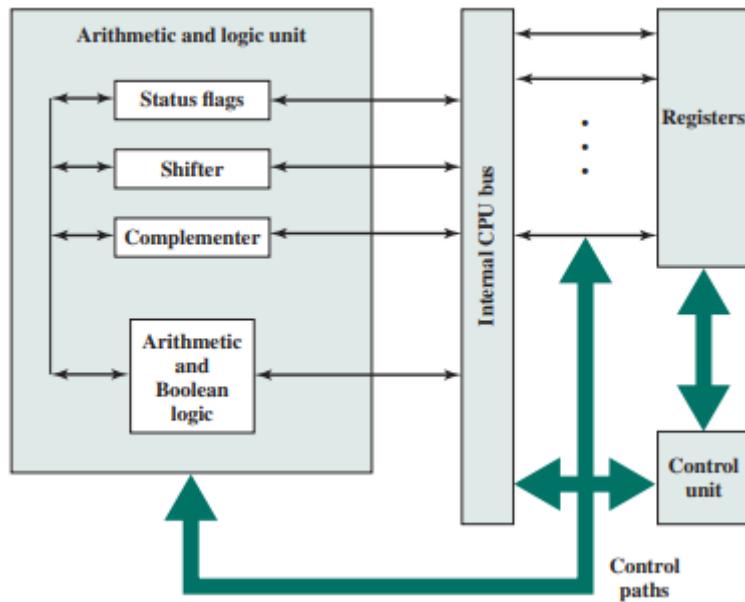
Untuk melakukan hal-hal ini, prosesor perlu menyimpan sementara beberapa data, harus mengingat lokasi instruksi terakhir sehingga bisa menentukan di mana mendapatkan instruksi selanjutnya. Prosesor juga perlu menyimpan instruksi dan data sementara saat instruksi sedang dijalankan. Dengan kata lain, prosesor membutuhkan memori internal yang kecil, yaitu register.



Sumber: (Stallings, 2016)
Gambar 7.1. CPU dengan System Bus

Gambar 7.1 adalah bagan sederhana prosesor, yang menunjukkan koneksinya ke seluruh sistem melalui bus sistem. Antarmuka diperlukan untuk setiap struktur interkoneksi ke perangkat eksternal. Komponen utama prosesor adalah *Aritmatic and Logic Unit* (ALU) dan *Control Unit* (CU). ALU melakukan perhitungan atau pemrosesan data yang sebenarnya. CU mengontrol pergerakan data dan instruksi masuk dan keluar dari prosesor dan mengontrol operasi ALU. Selain itu, gambar 7.1

menunjukkan memori internal minimal, terdiri dari satu set lokasi penyimpanan, yang disebut register. Gambar 7.2 adalah tampilan prosesor yang sedikit lebih rinci. Transfer data dan jalur kontrol logika ditunjukkan, termasuk elemen yang diberi label bus prosesor internal. Elemen ini diperlukan untuk mentransfer data antara berbagai register dan ALU karena ALU sebenarnya hanya beroperasi pada data dalam memori prosesor internal. Gambar ini juga menunjukkan elemen dasar khas ALU. Perhatikan kesamaan antara struktur internal komputer secara keseluruhan dan struktur internal prosesor. Dalam kedua kasus, ada koleksi kecil elemen utama (komputer: prosesor, I/O, memori; prosesor: *Control Unit*, ALU, register) yang dihubungkan oleh jalur data.



Sumber: (Stallings, 2016)
Gambar 7.2. Struktur Internal CPU

7.2. Organisasi Register

Pada hierarki memori dari tingkat yang lebih tinggi berlaku sifat: memori lebih cepat, lebih kecil, dan lebih mahal (per bit). Di dalam prosesor, ada satu set register yang berfungsi sebagai tingkat memori di atas memori utama dan *cache* dalam hierarki memorinya. Register dalam prosesor dikelompokkan berdasarkan perannya, yaitu:

- 1) Register yang terlihat pengguna (*user visible register*): mengaktifkan pemrogram bahasa mesin atau perakitan untuk meminimalkan referensi memori utama dengan mengoptimalkan penggunaan register.
- 2) Kontrol dan register status: Digunakan oleh *Control Unit* untuk mengontrol operasi prosesor dan oleh program sistem operasi yang istimewa untuk mengontrol pelaksanaan program.

A. *User Visible Register*

User visible register adalah register yang dapat direferensikan oleh programmer melalui bahasa mesin yang dijalankan oleh prosesor. Pada umumnya *user visible register* terdiri dari: *general purpose register*, *data register*, *address register*, dan *condition code register*.

General Purpose Register

Programmer dapat menugaskan General Purpose Register untuk berbagai keperluan. Semua General Purpose Register apapun dapat berisi *operand* untuk opcode apa pun. Ini menyediakan penggunaan register tujuan umum yang sebenarnya. Namun, seringkali ada batasan. Misalnya, mungkin ada register khusus untuk operasi *floating-point* dan stack.

Data register, dapat digunakan hanya untuk menyimpan data dan tidak dapat digunakan dalam perhitungan alamat *operand*.

Address register dapat disebut sebagai general purpose, atau dapat ditujukan untuk mode pengalamatan tertentu. Contohnya termasuk yang berikut ini:

- **Segment Pointer**: Dalam sebuah mesin dengan pengalamatan segmented (lihat Bagian 8.3), register segmen menyimpan alamat basis segmen. Mungkin ada beberapa register: misalnya, satu untuk sistem operasi dan satu untuk proses saat ini.
- **Indeks register**: Ini digunakan untuk pengalamatan yang diindeks dan mungkin diindeks otomatis.
- **Stack pointer register**: Jika ada penumpukan yang terlihat oleh pengguna, maka biasanya ada register khusus yang menunjuk ke bagian atas *stack*. Ini memungkinkan pengalamatan implisit; yaitu, push, pop, dan instruksi stack lainnya tidak perlu berisi *operand* stack eksplisit.

Flag Register. Kategori terakhir register, yang setidaknya sebagian terlihat oleh pengguna, memegang kode kondisi (juga disebut sebagai flag). Kode kondisi adalah bit yang ditetapkan oleh perangkat keras prosesor sebagai hasil dari operasi. Misalnya, operasi aritmatika dapat menghasilkan nilai positif, negatif, nol, atau *overflow*. Selain hasil operasi itu sendiri disimpan dalam register atau memori, kode kondisi yang merespon hasil operasi juga diatur dan disimpan dalam flag register. Kode selanjutnya dapat diuji sebagai bagian dari operasi cabang bersyarat.

Bit kode kondisi dikumpulkan ke dalam satu atau beberapa register. Biasanya, mereka membentuk bagian dari register kontrol. Secara umum, instruksi mesin memungkinkan bit-bit ini untuk dibaca oleh referensi implisit, tetapi programmer tidak dapat mengubahnya.

Di beberapa mesin, *subroutine call* akan menghasilkan penghematan otomatis semua register yang terlihat oleh pengguna, yang akan dikembalikan saat return. Prosesor melakukan penyimpanan dan pemulihannya sebagai bagian dari pelaksanaan instruksi *call* dan *return*. Ini memungkinkan setiap subrutin untuk menggunakan register yang terlihat oleh pengguna secara independen. Di mesin lain, merupakan tanggung jawab programmer untuk menyimpan konten register yang tidak terlihat oleh pengguna sebelum panggilan subrutin, dengan memasukkan instruksi untuk tujuan ini dalam program.

B. Kontrol dan Status Register

Ada berbagai register prosesor yang digunakan untuk mengontrol operasi prosesor. Sebagian besar dari ini, pada kebanyakan mesin, tidak terlihat oleh pengguna. Beberapa di antaranya mungkin terlihat oleh instruksi mesin yang dijalankan dalam mode kontrol atau sistem operasi.

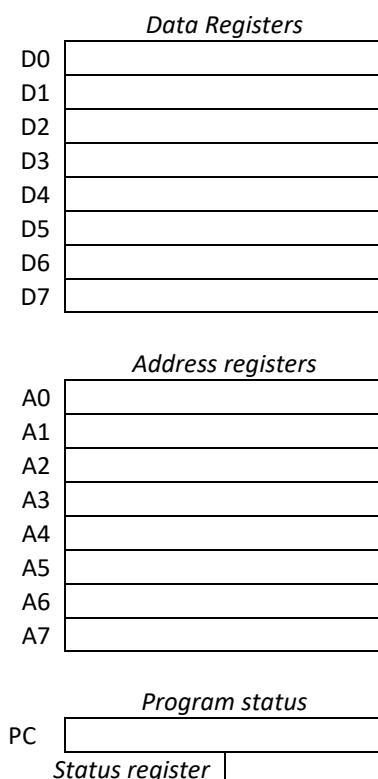
Gambar 7.3 adalah contoh Organisasi Register dari tiga Mikroprosesor berbeda. Mesin yang berbeda akan memiliki organisasi register yang berbeda dan menggunakan terminologi yang berbeda. Empat register sangat penting untuk pelaksanaan instruksi adalah:

- **Program Counter** (PC): Berisi alamat instruksi yang akan diambil.
- **Instruction Register** (IR): Berisi instruksi yang paling baru diambil.
- **Memory Address Register** (MAR): Berisi alamat lokasi dalam memori.
- **Memory Buffer Register** (MBR): Berisi *word* data yang akan ditulis ke memori atau *word* yang paling baru dibaca.

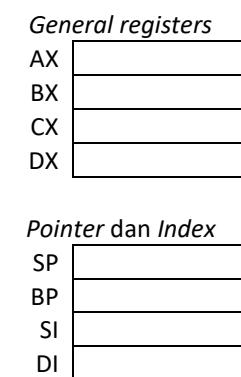
Tidak semua prosesor memiliki register internal yang ditunjuk sebagai MAR dan MBR, tetapi beberapa mekanisme *buffering* yang setara sangat diperlukan, dimana bit yang akan ditransfer ke bus sistem dilepaskan dan bit yang akan dibaca dari bus data disimpan sementara.

Biasanya, prosesor memperbarui register PC setelah setiap instruksi diambil sehingga register PC selalu menunjuk ke instruksi berikutnya yang akan dieksekusi. Instruksi cabang atau lompatan juga akan mengubah konten PC.

Instruksi yang diambil dimuat ke IR, untuk menentukan opcode dan *operand* yang dibutuhkan. Data dipertukarkan dengan memori menggunakan MAR dan MBR. Dalam sistem bus-terorganisir, MAR terhubung langsung ke bus alamat, dan MBR terhubung langsung ke bus data. User visible register, pada gilirannya, bertukar data dengan MBR. Keempat register yang disebutkan tadi digunakan untuk pergerakan data antara prosesor dan memori. Di dalam prosesor, data harus disajikan kepada ALU untuk diproses. ALU mungkin memiliki akses langsung ke MBR dan register yang terlihat oleh pengguna. Atau, mungkin ada register buffering tambahan di bawah ALU; register ini berfungsi sebagai register *input* dan *output* untuk ALU dan bertukar data dengan MBR dan register yang terlihat oleh pengguna.

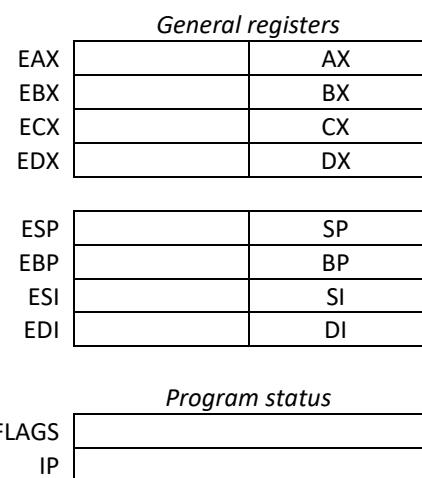


(a) MC68000



Program status
FLAGS
IP

(b) Intel 8086



(c) Intel 80x86 - Pentium 4

Sumber: (Stallings, 2016)
Gambar 7.3. Contoh Organisasi Register Mikroprosesor

Banyak desain prosesor termasuk register atau serangkaian register, sering dikenal sebagai *Program Status Word* (PSW), yang berisi informasi status. PSW biasanya berisi kode kondisi plus informasi status lainnya. Bidang atau bendera umum mencakup yang berikut:

- **Sign**: Berisi sedikit tanda dari hasil operasi aritmatika terakhir.
- **Zerro**: Di-set saat hasilnya 0.
- **Carry**: Mengatur apakah operasi menghasilkan *carry* (penambahan) ke dalam atau meminjam (mengurangi) bit orde tinggi. Digunakan untuk operasi aritmatika multi word.
- **Equal**: Setel jika hasil perbandingan logik adalah benar (*true*).
- **Overflow**: Digunakan untuk mengindikasikan *overflow* aritmatika.

- **Interrupt Enable/Disable:** Digunakan untuk mengaktifkan atau menonaktifkan interupsi.
- **Supervisor:** Mengindikasikan apakah prosesor mengeksekusi dalam mode supervisor atau pengguna. Instruksi khusus tertentu dapat dieksekusi hanya dalam mode supervisor, dan area memori tertentu dapat diakses hanya dalam mode supervisor.

Sejumlah register lain yang berkaitan dengan status dan kontrol dapat ditemukan dalam desain prosesor tertentu. Mungkin ada penunjuk ke blok memori yang berisi informasi status tambahan (misalnya *Process Block Control*). Dalam mesin yang menggunakan *vector interrupt*, register vektor interupsi dapat disediakan. Jika *stack* digunakan untuk mengimplementasikan fungsi tertentu (misalnya Panggilan subrutin), maka penunjuk *stack* sistem diperlukan. Penunjuk tabel halaman digunakan dengan sistem memori virtual. Akhirnya, register dapat digunakan dalam kontrol operasi I/O.

7.3. Siklus Instruksi

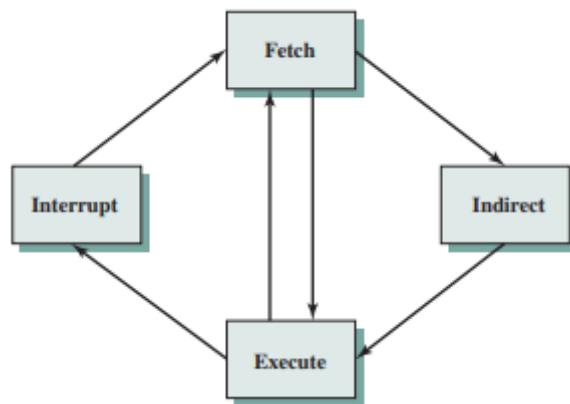
siklus instruksi yang sederhana meliputi tahapan-tahapan berikut:

- 1) *Fetch:* Baca instruksi selanjutnya dari memori ke dalam prosesor.
- 2) *Execute:* Menafsirkan *opcode* dan melakukan operasi yang ditunjukkan.
- 3) *Interrupt:* Jika interupsi diaktifkan dan interupsi telah terjadi, simpan status proses saat ini dan layanan interupsi.

A. Siklus Tidak Langsung (*Indirect Cycle*)

Pelaksanaan instruksi dapat melibatkan satu atau lebih *operand* dalam memori, yang masing-masing memerlukan akses memori. Lebih lanjut, jika pengalaman tidak langsung digunakan, maka akses memori tambahan diperlukan.

Dapat diasumsikan bahwa pengambilan alamat tidak langsung sebagai satu tahap instruksi lagi. Hasilnya ditunjukkan pada Gambar 7.4. Garis utama kegiatan terdiri dari pengambilan instruksi secara bergantian dan aktivitas pelaksanaan instruksi. Setelah instruksi diambil, itu diperiksa untuk menentukan apakah ada pengalaman tidak langsung yang terlibat. Jika demikian, *operand* yang diperlukan diambil menggunakan pengalaman tidak langsung. Setelah eksekusi, interupsi dapat diproses sebelum instruksi berikutnya diambil.

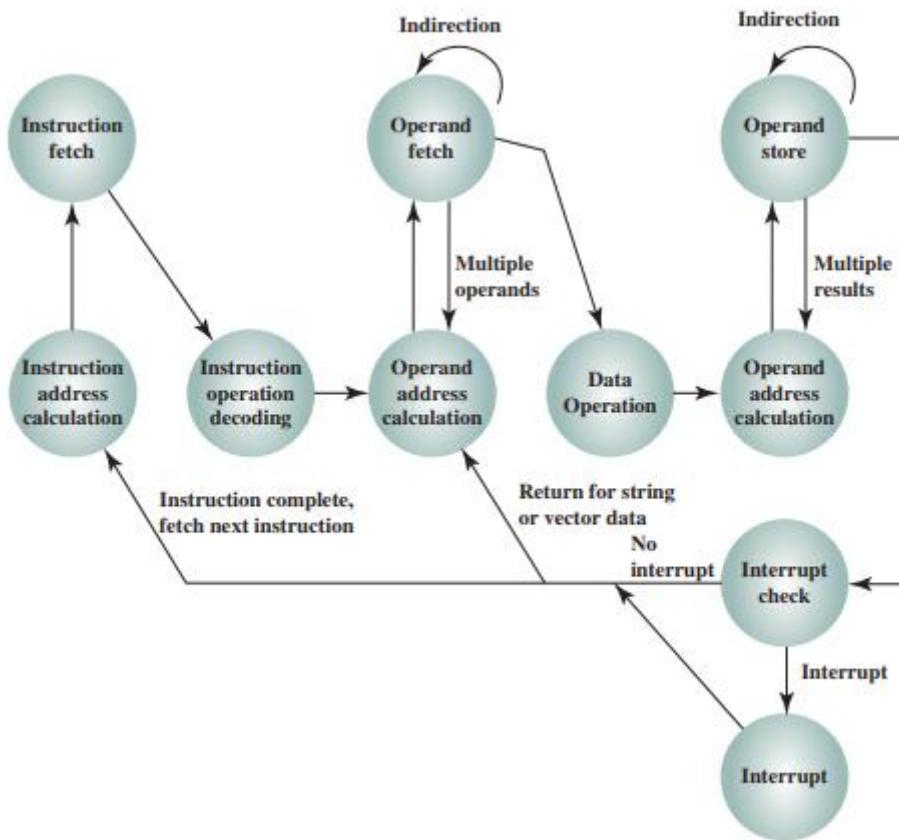


Sumber: (Stallings, 2016)

Gambar 7.4. Siklus Instruksi *Indirect*

Cara lain untuk melihat proses ini ditunjukkan pada Gambar 7.5, yang merupakan versi revisi Gambar 1.20. Ini menggambarkan lebih tepat sifat dari siklus instruksi. Setelah instruksi diambil, penentu *operand* harus diidentifikasi. Setiap *operand input* dalam memori kemudian diambil, dan proses ini mungkin memerlukan pengalaman tidak langsung. *Operand* berbasis register tidak perlu diambil.

Setelah opcode dieksekusi, proses serupa mungkin diperlukan untuk menyimpan hasilnya di memori utama.

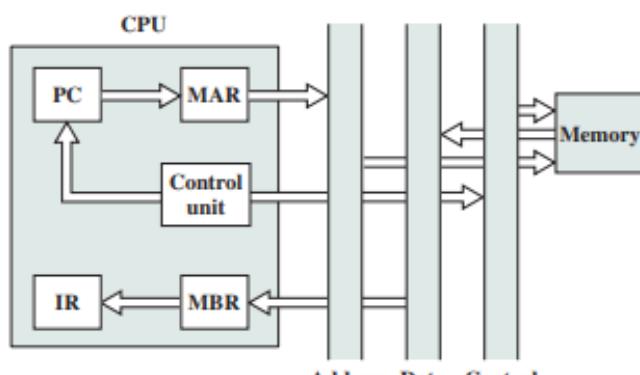


Sumber: (Stallings, 2016)

Gambar 7.5. State Diagram dari Siklus Instruksi

B. Aliran Data Pada Siklus Intruksi

Urutan kejadian yang tepat selama siklus instruksi tergantung pada desain prosesor. Diasumsikan bahwa prosesor yang menggunakan *Memory Address Register (MAR)*, *Memory Buffer Register (MBR)*, *Program Counter (PC)* dan *Instruction Register (IR)*.



MBR = Memory buffer register

MAR = Memory address register

IR = Instruction register

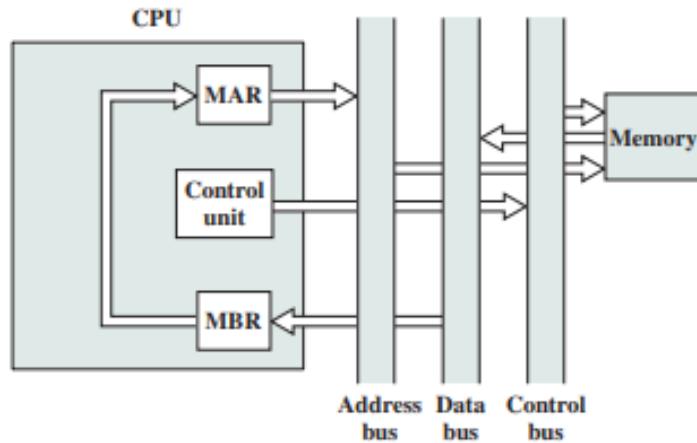
PC = Program counter

Sumber: (Stallings, 2016)

Gambar 7.6. Aliran data dari Fetch Cycle

Selama siklus pengambilan, instruksi dibaca dari memori. Gambar 7.6 menunjukkan aliran data selama siklus ini. PC berisi alamat instruksi berikutnya yang harus diambil. Alamat ini dipindahkan ke **MAR** dan ditempatkan di bus alamat. **Control Unit (CU)** meminta memori dibaca, dan hasilnya ditempatkan pada bus data dan disalin ke **MBR** dan kemudian dipindahkan ke **IR**. Sementara itu, **PC** bertambah 1, persiapan untuk pengambilan berikutnya.

Setelah siklus pengambilan selesai, **CU** akan memeriksa konten **IR** untuk menentukan apakah ia berisi *specifier operand* menggunakan pengalaman tidak langsung. Jika demikian, siklus tidak langsung dilakukan. Seperti yang ditunjukkan pada Gambar 7.7, ini adalah siklus sederhana. Bit N paling kanan dari **MBR**, yang berisi referensi alamat, ditransfer ke **MAR**. Kemudian **Control Unit** meminta memori dibaca, untuk mendapatkan alamat *operand* yang diinginkan ke dalam **MBR**.

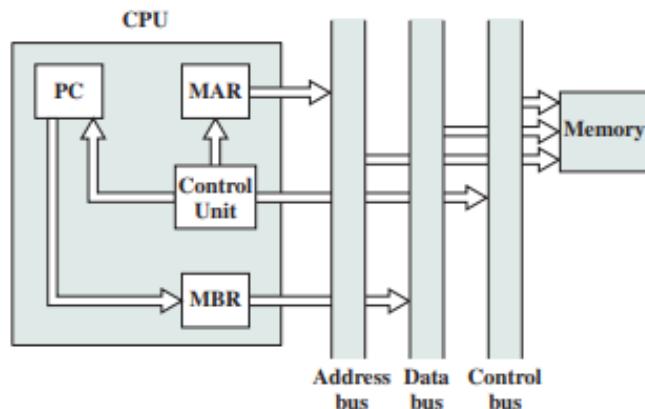


Sumber: (Stallings, 2016)

Gambar 7.7. Aliran data dari *Indirect Cycle*

Siklus pengambilan dan tidak langsung sederhana dan dapat diprediksi. Siklus eksekusi memiliki banyak bentuk; bentuknya tergantung pada instruksi mesin mana yang ada di IR. Siklus ini mungkin melibatkan transfer data di antara register, baca atau tulis dari memori atau I/O, dan/atau dari **ALU**.

Gambar 7.8 menunjukkan aliran data pada siklus interupsi. Isi **PC** saat ini harus disimpan sehingga prosesor dapat melanjutkan aktivitas normal setelah interupsi. Dengan demikian, isi **PC** ditransfer ke **MBR** untuk ditulis ke dalam memori. Lokasi memori khusus yang disediakan untuk tujuan ini dimuat ke dalam **MAR** dari **Control Unit**. Mungkin, misalnya, menjadi penunjuk *stack*. **PC** dimuat dengan alamat rutin interupsi. Akibatnya, siklus instruksi berikutnya akan dimulai dengan mengambil instruksi yang sesuai.



Sumber: (Stallings, 2016)

Gambar 7.8. Aliran Data dari *Interrupt Cycle*

7.4. Prosesor Keluarga x86

A. Organisasi Register

Organisasi register meliputi jenis register berikut (Tabel 7.1):

- 1) **General register:** Ada delapan register tujuan umum 32-bit (lihat Gambar 7.3c). Ini dapat digunakan untuk semua jenis instruksi x86; mereka juga dapat menahan *operand* untuk perhitungan alamat. Selain itu, beberapa register ini juga melayani tujuan khusus. Sebagai contoh, instruksi string menggunakan isi register ECX, ESI, dan EDI sebagai *operand* tanpa harus merujuk register ini secara eksplisit dalam instruksi. Hasilnya, sejumlah instruksi dapat dikodekan secara lebih kompak. Dalam mode 64-bit, ada enam belas register tujuan umum 64-bit.
- 2) **Segmen register:** Enam register segmen 16-bit berisi pemilih segmen, yang indeks ke dalam tabel segmen. Register segmen kode (CS) merujuk segmen yang berisi instruksi yang sedang dieksekusi. Register stack segment (SS) mereferensikan segmen yang berisi stack yang terlihat oleh pengguna. Register segmen yang tersisa (DS, ES, FS, GS) memungkinkan pengguna untuk referensi hingga empat segmen data yang terpisah sekaligus.
- 3) **Flags register:** Register EFLAGS 32-bit berisi kode kondisi dan berbagai bit mode. Dalam mode 64-bit, register ini diperpanjang hingga 64 bit dan disebut sebagai RFLAGS. Dalam definisi arsitektur saat ini, 32 bit atas RFLAGS tidak digunakan.
- 4) **Instruktion Pointer register:** Berisi alamat instruksi saat ini.
- 5) Ada juga register yang dikhususkan untuk unit *floating-point*:
- 6) **Numeric register:** Setiap register memegang angka *floating-point* 80-bit presisi-tinggi. Ada delapan register yang berfungsi sebagai *stack*, dengan operasi *push* dan *pop* tersedia dalam set instruksi.
- 7) **Control register:** Register kontrol 16-bit berisi bit yang mengontrol operasi unit titik-mengambang, termasuk jenis kontrol pembulatan; presisi tunggal, ganda, atau diperpanjang; dan bit untuk mengaktifkan atau menonaktifkan berbagai kondisi pengecualian.
- 8) **Status register:** Register status 16-bit berisi bit yang mencerminkan keadaan saat ini dari unit titik-mengambang, termasuk penunjuk 3-bit ke bagian atas *stack*; kode kondisi yang melaporkan hasil operasi terakhir; dan bendera pengecualian.
- 9) **Tag Word register:** Register 16-bit ini berisi tag 2-bit untuk setiap register numerik *floating-point*, yang menunjukkan sifat konten register yang sesuai. Keempat nilai yang mungkin adalah valid, nol, khusus (NaN, tak terhingga, didenormalkan), dan kosong. Tag ini memungkinkan program untuk memeriksa konten register numerik tanpa melakukan decoding kompleks dari data aktual dalam register. Misalnya, ketika sakelar konteks dibuat, prosesor tidak perlu menyimpan register titik-mengambang yang kosong.

Register EFLAGS (Gambar 7.9) menunjukkan kondisi prosesor dan membantu mengendalikan operasinya. Ini mencakup enam kode kondisi yang merespon hasil operasi integer, yaitu: *carry* C, parity P, auxiliary A, nol Z, sign S, dan *overflow* O. Selain itu, ada bit dalam register yang dapat disebut sebagai bit kontrol:

- 1) **Trap Flag (TF):** Ketika diatur, menyebabkan interupsi setelah eksekusi setiap instruksi. Ini digunakan untuk debugging.
- 2) **Interrupt Flag (IF):** Saat diset 1, prosesor akan mengenali interupsi eksternal.
- 3) **Direction Flag (DF):** Menentukan apakah instruksi pemrosesan string menambah atau mengurangi SI 16-bit setengah register dan DI (untuk operasi 16-bit) atau 32-bit register ESI dan EDI (untuk operasi 32-bit).
- 4) **I/O Privilege (IOPL):** Ketika diatur, menyebabkan prosesor menghasilkan pengecualian pada semua akses ke perangkat I/O selama operasi mode terlindungi.

- 5) **Resume Flag** (RF): Memungkinkan programmer untuk menonaktifkan pengecualian debug sehingga instruksi dapat dimulai kembali setelah pengecualian debug tanpa segera menyebabkan pengecualian debug lainnya.
- 6) **Alignment Check** (AC): Aktif jika *word* atau *doubleword* dialamatkan pada batas *nonword* atau *nondoubleword*.
- 7) **Identification Flag** (ID): Jika bit ini dapat diatur dan dihapus, maka prosesor ini mendukung instruksi prosesorID. Instruksi ini memberikan informasi tentang vendor, keluarga, dan model.

Tabel 7.1. Organisasi Register Keluarga Intel x86

(a) Unit Integer pada Mode 32-bit

Type	Jumlah	Length (bits)	Tujuan
General	8	32	General-purpose user register
Segment	6	16	Contain segment selectors
EFLAGS	1	32	Status and control bits
Instruction Pointer	1	32	Instruction pointer

(b) Unit Integer pada Mode 64-bit

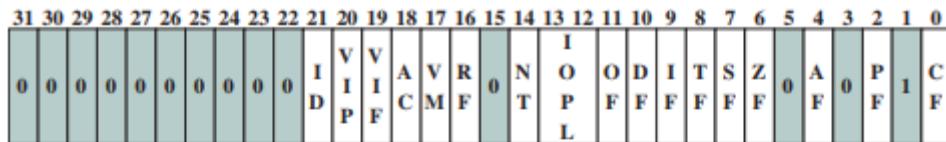
Type	Jumlah	Length (bits)	Tujuan
General	16	32	General-purpose user register
Segment	6	16	Contain segment selectors
RFLAGS	1	64	Status and control bits
Instruction Pointer	1	64	Instruction pointer

(c) Unit *Floating-point*

Type	Jumlah	Length (bits)	Tujuan
Numeric	8	80	Hold floating point numbers
Control	1	16	Control bits
Status	1	16	Status bits
Tag Word	1	16	Specifies contents of numerical registers
Instruction Pointer	1	48	Point to instruction interrupted by exception
Data Pointer	1	48	Points to <i>operand</i> interrupted by exception

Sumber: (Stallings, 2016)

Selain itu, ada 4 bit yang berhubungan dengan mode operasi. Bendera **Nested Task** (NT) menunjukkan bahwa tugas saat ini bersarang dalam tugas lain dalam operasi metode terproteksi. Bit **Virtual Mode** (VM) memungkinkan programmer untuk mengaktifkan atau menonaktifkan mode virtual 8086, yang menentukan apakah prosesor berjalan sebagai mesin 8086. Bendera **Virtual Interrupt Flag** (VIF) dan **Virtual Interrupt Pending** (VIP) digunakan dalam lingkungan *multitasking*.



X ID = Identification flag
 X VIP = Virtual interrupt pending
 X VIF = Virtual interrupt flag
 X AC = Alignment check
 X VM = Virtual 8086 mode
 X RF = Resume flag
 X NT = Nested task flag
 X IOPL = I/O privilege level
 S OF = Overflow flag

C DF = Direction flag
 X IF = Interrupt enable flag
 X TF = Trap flag
 S SF = Sign flag
 S ZF = Zero flag
 S AF = Auxiliary carry flag
 S PF = Parity flag
 S CF = Carry flag

S indicates a status flag.
 C indicates a control flag.
 X indicates a system flag.
 Shaded bits are reserved.

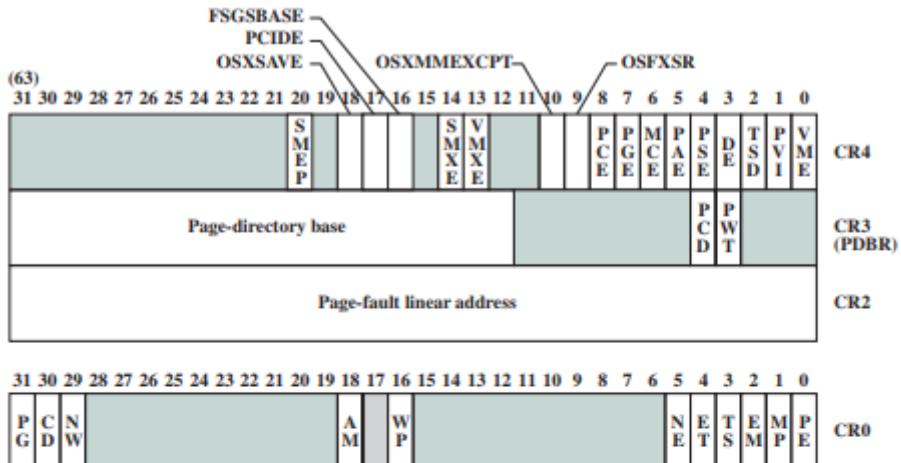
Sumber: (Stallings, 2016)
Gambar 7.9. Register EFLAGS x86

B. Control Register

X86 menggunakan empat register kontrol (register CR1 tidak digunakan) untuk mengontrol berbagai aspek operasi prosesor (Gambar 7.10). Semua register kecuali CR0 panjangnya 32 bit atau 64 bit, tergantung pada apakah implementasinya mendukung arsitektur x86 64-bit. Register CR0 berisi flag kontrol sistem, yang mengontrol mode atau menunjukkan status yang berlaku secara umum untuk prosesor dan bukan pada pelaksanaan tugas individu. Bendera adalah sebagai berikut:

- 1) *Protection Enable* (PE): Mengaktifkan/menonaktifkan mode operasi yang dilindungi.
- 2) *Monitor Coprocessor* (MP): Hanya menarik ketika menjalankan program dari mesin sebelumnya di x86; ini berkaitan dengan keberadaan coprocessor aritmatika.
- 3) *Emulation* (EM): Setelah saat prosesor tidak memiliki unit titik-mengambang, dan menyebabkan interupsi ketika upaya dilakukan untuk menjalankan instruksi titik-mengambang.
- 4) *Task Switched* (TS): Menunjukkan bahwa prosesor telah berpindah tugas.
- 5) *Extension Type* (ET): Tidak digunakan pada Pentium dan mesin yang lebih baru; digunakan untuk menunjukkan dukungan instruksi coprocessor matematika pada mesin sebelumnya.
- 6) *Numeric Error* (NE): Mengaktifkan mekanisme standar untuk melaporkan kesalahan *floating point* pada jalur bus eksternal.
- 7) *Write Protect* (WP): Ketika bit ini jelas, halaman tingkat pengguna hanya bisa ditulis oleh proses penyelia. Fitur ini berguna untuk mendukung pembuatan proses di beberapa sistem operasi.
- 8) *Alignment Mask* (AM): Mengaktifkan/menonaktifkan pemeriksaan penyelarasian.
- 9) *Not Write Through* (NW): Memilih mode operasi *cache* data. Ketika bit ini disetel, *cache* data dihambat dari operasi write-through *cache*.
- 10) *Cache Disable* (CD): Mengaktifkan/menonaktifkan mekanisme pengisian *cache* internal.
- 11) *Paging* (PG): Mengaktifkan/menonaktifkan *paging*.

Saat paging diaktifkan, register CR2 dan CR3 valid. Register CR2 memegang alamat linear 32-bit dari halaman terakhir yang diakses sebelum *page fault interrupt*. 20 bit paling kiri dari CR3 menampung 20 bit paling signifikan dari alamat dasar direktori halaman; sisa alamat berisi nol. Dua bit CR3 digunakan untuk menggerakkan pin yang mengontrol operasi *cache* eksternal. Penonaktifan *cache* pagelevel (PCD) mengaktifkan atau menonaktifkan *cache* eksternal, dan kontrol tingkat *Page Write Transparant* (PWT) menulis melalui *cache* eksternal. CR4 berisi bit kontrol tambahan.



Shaded area indicates reserved bits.

OSXSAVE	= XSAVE enable bit	VME	= Virtual 8086 mode extensions
PCIDE	= Enables process-context identifiers	PCD	= Page-level cache disable
FSGSBASE	= Enables segment base instructions	PWT	= Page-level writes transparent
SMXE	= Enable safer mode extensions	PG	= Paging
VMXE	= Enable virtual machine extensions	CD	= Cache disable
OSXMMEXCPT	= Support unmasked SIMD FP exceptions	NW	= Not write through
OSFXSR	= Support FXSAVE, FXSTOR	AM	= Alignment mask
PCE	= Performance counter enable	WP	= Write protect
PGE	= Page global enable	NE	= Numeric error
MCE	= Machine check enable	ET	= Extension type
PAE	= Physical address extension	TS	= Task switched
PSE	= Page size extensions	EM	= Emulation
DE	= Debug extensions	MP	= Monitor coprocessor
TSD	= Time stamp disable	PE	= Protection enable
PVI	= Protected mode virtual interrupt		

Sumber: (Stallings, 2016)

Gambar 7.10. Register Control x86

Sembilan bit kontrol tambahan didefinisikan di CR4:

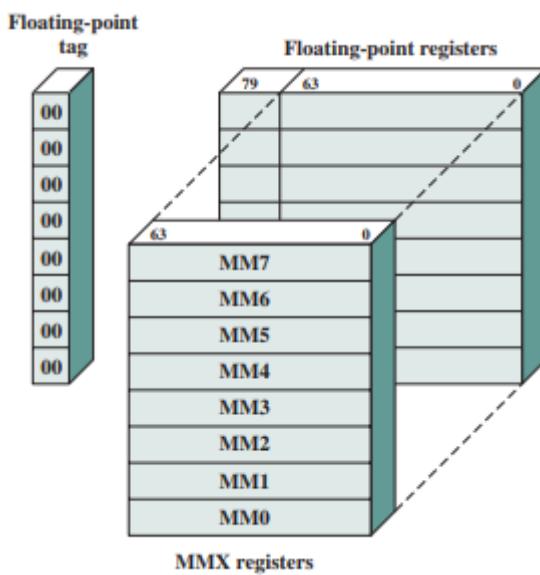
- 1) Virtual-8086 Mode Extension (VME): Mengaktifkan dukungan untuk bendera interupsi virtual dalam mode virtual-8086.
- 2) Protected-mode Virtual Interrupts (PVI): Mengaktifkan dukungan untuk bendera interupsi virtual dalam mode terlindungi.
- 3) Time Stamp Disable (TSD): Menonaktifkan instruksi baca dari penghitung cap waktu (RDTSC), yang digunakan untuk tujuan debugging.
- 4) Debugging Extensions (DE): Mengaktifkan breakpoint I / O; hal ini memungkinkan prosesor untuk melakukan interupsi pada I / O pembacaan dan penulisan.
- 5) Page Size Extensions (PSE): Mengaktifkan ukuran halaman besar (halaman 2 atau 4 MByte) saat disetel; membatasi halaman hingga 4 KByte jika kosong.
- 6) Physical Address Extension (PAE): Mengaktifkan baris alamat A35 hingga A32 setiap kali mode pengalamatan baru khusus, yang dikontrol oleh PSE, diaktifkan.
- 7) Machine Check Enable (MCE): Mengaktifkan interupsi pemeriksaan mesin, yang terjadi saat kesalahan paritas data terjadi selama siklus bus baca atau saat siklus bus tidak berhasil diselesaikan.
- 8) Page Global Enable (PGE): Memungkinkan penggunaan halaman global. Ketika PGE = 1 dan sakelar tugas dijalankan, semua entri TLB dihapus dengan pengecualian yang ditandai global.
- 9) Performance Counter Enable (PCE): Mengaktifkan eksekusi instruksi RDPMC (read performance counter) pada tingkat hak istimewa apa pun. Dua penghitung kinerja digunakan untuk mengukur durasi jenis peristiwa tertentu dan jumlah kemunculan jenis peristiwa tertentu.

C. MMX Register

Salah satu kemampuan Intel x86 MMX adalah memanfaatkan beberapa tipe data 64-bit. Instruksi MMX menggunakan bidang alamat register 3-bit, sehingga delapan register MMX didukung. Bahkan, prosesor tidak termasuk register MMX tertentu. Sebaliknya, prosesor menggunakan teknik aliasing (Gambar 7.11). Register *floating-point* yang ada digunakan untuk menyimpan nilai MMX.

Secara khusus, 64 bit urutan rendah (mantissa) dari setiap register *floating-point* digunakan untuk membentuk delapan register MMX. Dengan demikian, arsitektur x86 32-bit yang lebih lama mudah diperluas untuk mendukung kemampuan MMX. Beberapa karakteristik utama dari penggunaan MMX register ini adalah sebagai berikut:

- Ingat bahwa register titik-mengambang diperlakukan sebagai *stack* untuk operasi titik-mengambang. Untuk operasi MMX, register yang sama ini diakses secara langsung.
- Pertama kali instruksi MMX dijalankan setelah operasi *floating-point*, word tag FP ditandai sah. Ini mencerminkan perubahan dari operasi stack ke pengalaman register langsung.
- Instruksi EMMS (Empty MMX State) menetapkan bit word tag FP untuk menunjukkan bahwa semua register kosong. Adalah penting bahwa programmer memasukkan instruksi ini di akhir blok kode MMX sehingga operasi floatingpoint berikutnya berfungsi dengan baik.
- Ketika sebuah nilai ditulis ke register MMX, bit [79:64] dari register FP yang sesuai (bit sign dan eksponen) disetel ke semua. Ini menetapkan nilai dalam register FP ke NaN (bukan angka) atau tak terhingga ketika dilihat sebagai nilai floatingpoint. Ini memastikan bahwa nilai data MMX tidak akan terlihat seperti nilai *floating-point* yang valid.



Sumber: (Stallings, 2016)

Gambar 7.11. Mapping Register MMX ke Register Floating Point

D. Pemrosesan Interupsi

Pemrosesan interupsi dalam prosesor adalah fasilitas yang disediakan untuk mendukung sistem operasi. Ini memungkinkan program aplikasi untuk ditangguhkan, agar berbagai kondisi interupsi dapat diperbaiki dan kemudian dilanjutkan.

interupsi dan pengecualian Dua kelas peristiwa menyebabkan x86 untuk menunda eksekusi aliran instruksi saat ini dan menanggapi peristiwa: interupsi dan pengecualian. Dalam kedua kasus tersebut, prosesor menyimpan konteks proses saat ini dan mentransfer ke rutin yang telah ditentukan untuk memperbaiki kondisi tersebut. Interupsi dihasilkan oleh sinyal dari perangkat keras, dan itu dapat

terjadi secara acak selama pelaksanaan suatu program. Pengecualian dihasilkan dari perangkat lunak, dan diprovokasi oleh pelaksanaan instruksi. Ada dua sumber interupsi dan dua sumber pengecualian:

- 1) *Interrupt*
 - a. *Maskable Interrupt*: Diterima pada pin INTR prosesor. Prosesor tidak mengenali interupsi yang dapat ditutup kecuali jika interrupt enable flag (IF) diatur.
 - b. *Nonmaskable Interrupt*: Diterima pada pin NMI prosesor. permintaan Interupsi tersebut tidak dapat dicegah.
- 2) *Exception*
- 2) Pengecualian yang terdeteksi prosesor: Hasil saat prosesor menemukan kesalahan saat mencoba menjalankan instruksi.
- 3) Pengecualian terprogram: Ini adalah instruksi yang menghasilkan pengecualian (misalnya: INTO, INT3, INT, dan BOUND).

E. Tabel Vektor Interupsi

Pemrosesan interupsi pada x86 menggunakan tabel vektor interupsi. Setiap jenis interupsi diberi nomor, dan nomor ini digunakan untuk mengindeks ke tabel vektor interupsi. Tabel ini berisi 256 vektor interupsi 32-bit, yang merupakan alamat (*segment* dan *offset*) dari layanan rutin interupsi untuk nomor interupsi itu.

Tabel 7.2 menunjukkan penugasan angka dalam tabel vektor interupsi; entri yang diarsir mewakili interupsi, sedangkan entri yang tidak diarsir adalah pengecualian. Interupsi perangkat keras NMI adalah tipe 2. Interupsi perangkat keras INTR ditetapkan angka dalam kisaran 32 hingga 255; ketika interupsi INTR dibuat, itu harus disertai pada bus dengan nomor vektor interupsi untuk interupsi ini. Angka vektor yang tersisa digunakan untuk pengecualian.

Jika lebih dari satu pengecualian atau interupsi tertunda, prosesor akan melayani mereka dalam urutan yang dapat diprediksi. Lokasi angka vektor dalam tabel tidak mencerminkan prioritas. Sebaliknya, prioritas di antara pengecualian dan interupsi diorganisasikan ke dalam lima kelas. Dalam urutan prioritas yang menurun, ini adalah:

- Kelas 1: Jebakan pada instruksi sebelumnya (nomor vektor 1)
- Kelas 2: Interupsi eksternal (2, 32–255)
- Kelas 3: Kesalahan saat mengambil instruksi selanjutnya (3, 14)
- Kelas 4: Kesalahan saat memecahkan kode instruksi selanjutnya (6, 7)
- Kelas 5: Kesalahan dalam menjalankan instruksi (0, 4, 5, 8, 10-14, 16, 17)

F. Interrupt Handling

Seperti halnya transfer eksekusi menggunakan instruksi CALL, transfer ke rutin penanganan interupsi menggunakan *stack* sistem untuk menyimpan status prosesor. Saat interupsi terjadi dan dikenali oleh prosesor, serangkaian peristiwa terjadi:

- 1) Jika transfer melibatkan perubahan tingkat hak istimewa, maka register segmen stack saat ini dan register stack extended extended (ESP) didorong ke stack.
- 2) Nilai register EFLAGS saat ini didorong ke stack.
- 3) Interrupt Flag (IF) dan Trap (TF) dihapus. Ini menonaktifkan INTR interupsi dan fitur trap atau single-step.
- 4) Code Segment Pointer saat ini (CS) dan instruction Pointer saat ini (IP atau EIP) didorong ke stack.
- 5) Jika interupsi disertai dengan kode kesalahan, maka kode kesalahan didorong ke *stack*.
- 6) Isi vektor interupsi diambil dan dimuat ke dalam register CS dan IP atau EIP. Eksekusi berlanjut dari rutinitas layanan interupsi.

Untuk kembali dari interupsi, rutinitas layanan interupsi mengeksekusi instruksi IRET. Ini menyebabkan semua nilai yang disimpan di *stack* dikembalikan; eksekusi dilanjutkan dari titik interupsi.

Tabel 7.2. Tabel *Vector Interrupt and Exception* keluarga x86

Nomor Vector	Deskripsi
0	Divide error; overflow pada pembagian atau pembagian dengan nol
1	Debug exception; menyertakan berbagai kesalahan (fault) dan jebakan (traps) yang terkait dengan debugging.
2	NMI pin interrupt; sinyal pada pin NMI
3	Breakpoint; disebabkan oleh instruksi INT 3, yang merupakan instruksi 1-byte yang berguna untuk debugging
4	INTO-detected overflow; terjadi ketika prosesor mengeksekusi INTO dengan set flag OF
5	Rentang BOUND terlampaui; instruksi BOUND membandingkan register dengan batas-batas yang disimpan dalam memori dan menghasilkan interupsi jika isi register di luar batas.
6	Opcode tidak ditentukan
7	Perangkat tidak tersedia; upaya untuk menggunakan instruksi ESC atau WAIT gagal karena kurangnya perangkat eksternal
8	Kesalahan ganda; dua interupsi terjadi selama instruksi yang sama dan tidak dapat ditangani secara serial
9	Dicadangkan (Reserved)
10	Invalid task state segment; segmen yang menjelaskan tugas yang diminta tidak diinisialisasi atau tidak valid
11	Segmen tidak ada; segmen yang dibutuhkan tidak ada
12	Stack fault; batas stack segment terlampaui atau tidak ada
13	General protection; pelanggaran perlindungan yang tidak menyebabkan pengecualian lain (misalnya: menulis ke read-only segment)
14	Page fault
15	Dicadangkan (Reserved)
16	Kesalahan <i>floating-point</i> ; dihasilkan oleh instruksi aritmatika <i>floating-point</i>
17	Alignment check; akses ke word yang disimpan di alamat byte ganjil atau doubleword yang disimpan di alamat bukan kelipatan 4
18	Machine check; model tertentu
19-31	Dicadangkan (Reserved)
32-255	User interrupt vectors; isediakan saat sinyal INTR diaktifkan

Sumber: (Stallings, 2016)

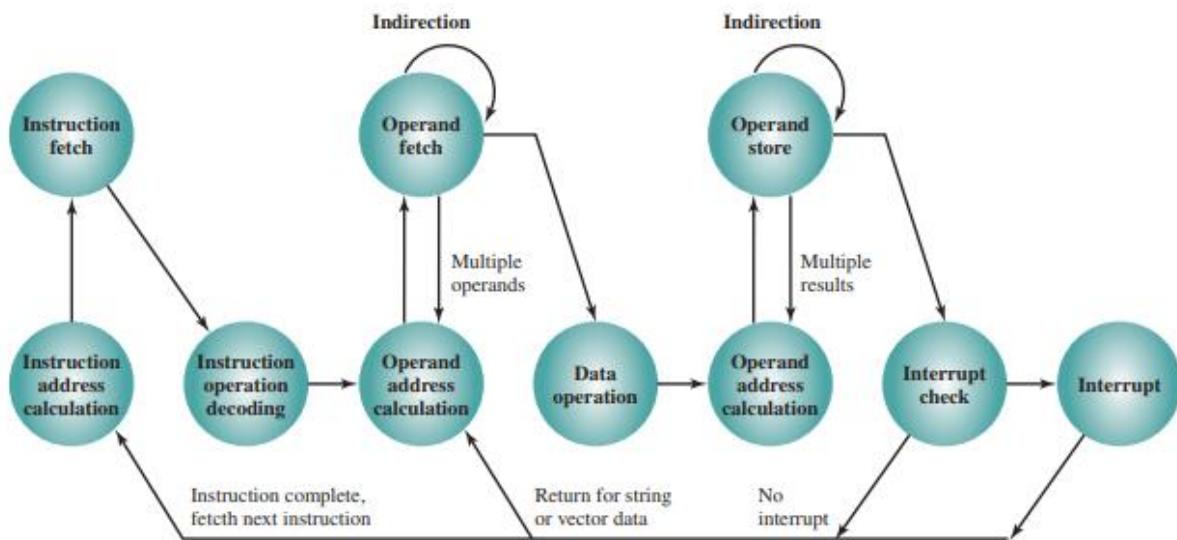
7.5. Pipelining Instruksi

Instruksi *pipelining* mirip dengan penggunaan jalur perakitan di pabrik. Jalur perakitan mengambil keuntungan dari kenyataan bahwa suatu produk melewati berbagai tahap (*stage*) produksi. Dengan meletakkan proses produksi di jalur perakitan, produk pada berbagai tahap dapat dikerjakan secara bersamaan. Proses ini juga disebut sebagai *pipelining*, karena, seperti dalam pipa, *input* baru diterima di satu ujung sebelum *input* yang sebelumnya diterima muncul sebagai *output* di ujung lainnya.

Untuk menerapkan konsep ini pada eksekusi instruksi, suatu instruksi memiliki sejumlah tahapan (*stage*). Gambar 7.12, misalnya, memecah siklus instruksi menjadi 10 tugas, yang terjadi secara berurutan. Jelas, harus ada beberapa peluang untuk *pipelining*.

Sebagai pendekatan sederhana, pertimbangkan untuk membagi pemrosesan instruksi menjadi dua stage: ambil (*fetch*) instruksi dan jalankan (*execute*) instruksi. Ada saat selama pelaksanaan instruksi

ketika memori utama tidak diakses. Waktu ini dapat digunakan untuk mengambil instruksi berikutnya secara paralel dengan pelaksanaan yang sekarang.



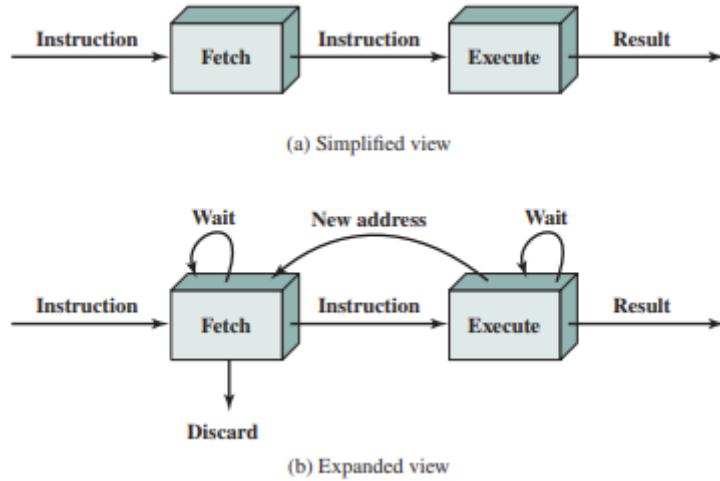
Sumber: (Stallings, 2016)

Gambar 7.12. State Diagram Siklus Instruksi

Gambar 7.13a menggambarkan pendekatan ini. Pipa memiliki dua tahap independen. Tahap pertama mengambil instruksi dan buffer itu. Ketika tahap kedua bebas, tahap pertama melewati instruksi *buffered*. Sementara tahap kedua menjalankan instruksi, tahap pertama mengambil keuntungan dari siklus memori yang tidak digunakan untuk mengambil dan buffer instruksi berikutnya. Ini disebut instruksi *prefetch* atau *fetch overlap*. Perhatikan bahwa pendekatan ini, yang melibatkan buffer instruksi, membutuhkan lebih banyak register. Secara umum *pipelining* membutuhkan register untuk menyimpan data antar tahap (stage).

Jika tahap *fetch* dan *execute* memiliki durasi yang sama, waktu siklus instruksi akan berkurang setengahnya. Namun, jika dilihat lebih dekat pada *pipa* ini (Gambar 7.13b), didapat bahwa penggandaan laju eksekusi ini tidak mungkin karena dua alasan:

- 1) Waktu *execute* umumnya akan lebih lama dari waktu *fetch*. Eksekusi akan melibatkan membaca dan menyimpan *operand* dan kinerja beberapa operasi. Jadi, tahap *fetch* mungkin harus menunggu beberapa saat sebelum dapat mengosongkan buffernya.
- 2) Instruksi cabang bersyarat membuat alamat instruksi berikutnya tidak diketahui. Dengan demikian, tahap *fetch* harus menunggu hingga menerima alamat instruksi berikutnya dari tahap *execute*. Tahap *execute* mungkin harus menunggu sementara instruksi berikutnya diambil.



Sumber: (Stallings, 2016)

Gambar 7.13. Dua Stage Pipeline Instruksi

Ketika instruksi cabang bersyarat diteruskan dari *fetch* ke tahap *execute*, tahap *fetch* mengambil instruksi berikutnya dalam memori setelah instruksi cabang. Kemudian, jika cabang tidak diambil, tidak ada waktu yang hilang. Jika cabang diambil, instruksi yang diambil harus dibuang dan instruksi yang baru diambil. Sementara faktor-faktor ini mengurangi keefektifan potensial dari pipa dua-tahap, beberapa percepatan terjadi. Untuk mendapatkan percepatan lebih lanjut, pipa harus memiliki lebih banyak tahapan (stage). Perhatikan dekomposisi berikut dari pemrosesan instruksi.

- **Fetch Instruction (FI)**: Baca instruksi selanjutnya yang diharapkan, simpan ke dalam buffer.
- **Dekode Instrucion (DI)**: Menentukan opcode dan *operand*.
- **Calculate Operand (CO)**: Hitung alamat efektif dari setiap *source operand*. Ini mungkin melibatkan *displacement*, *register indirect*, *indirect*, atau bentuk kalkulasi alamat lainnya.
- **Fetch Operand (FO)**: Ambil *operand* dari memori, *operand* dalam register tidak perlu diambil.
- **Execute Instrucion (EI)**: Lakukan operasi yang ditunjukkan dan simpan hasilnya, jika ada, di lokasi *operand tujuan* yang ditentukan.
- **Write Operand (WO)**: Menyimpan hasilnya dalam memori.

Dengan dekomposisi ini, berbagai tahapan akan memiliki durasi yang hampir sama. Untuk ilustrasi, setiap stage diasumsikan memiliki durasi yang sama. Dengan menggunakan asumsi ini, Gambar 7.14 menunjukkan bahwa pipa enam stage dapat mengurangi waktu pelaksanaan untuk 9 instruksi dari 54 unit waktu menjadi 14 unit waktu.

Time →

	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Instruction 1	FI	DI	CO	FO	EI	WO								
Instruction 2		FI	DI	CO	FO	EI	WO							
Instruction 3			FI	DI	CO	FO	EI	WO						
Instruction 4				FI	DI	CO	FO	EI	WO					
Instruction 5					FI	DI	CO	FO	EI	WO				
Instruction 6						FI	DI	CO	FO	EI	WO			
Instruction 7							FI	DI	CO	FO	EI	WO		
Instruction 8								FI	DI	CO	FO	EI	WO	
Instruction 9									FI	DI	CO	FO	EI	WO

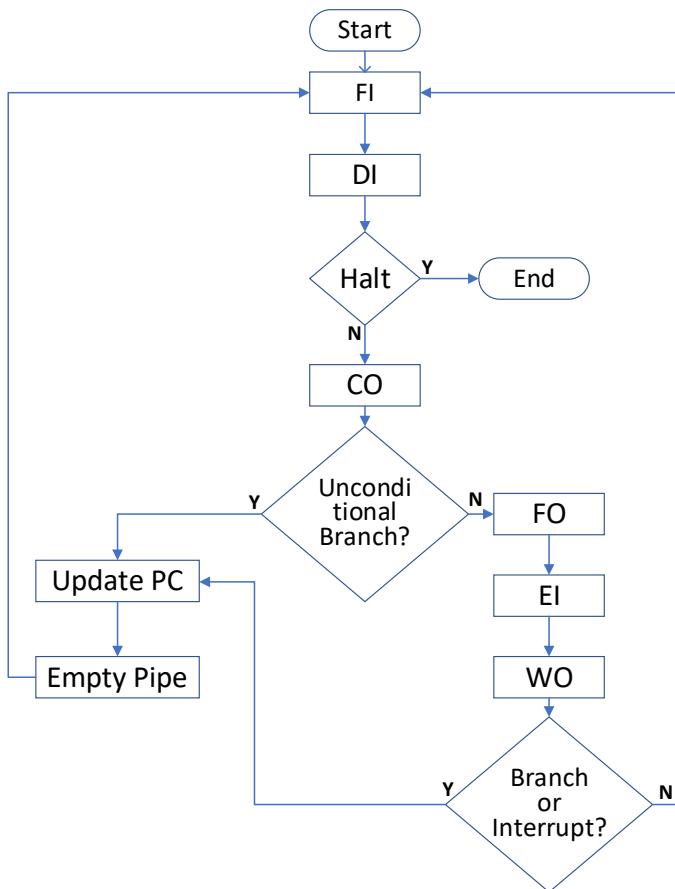
Sumber: (Stallings, 2016)

Gambar 7.14. Time Diagram Untuk Operasi Pipeline Instruksi Dengan Enam Stage

Diagram pada gambar 7.14 mengasumsikan bahwa setiap instruksi melewati semua enam stage pada *pipeline*. Ini tidak selalu terjadi. Misalnya, instruksi LOAD tidak memerlukan tahap WO. Namun, untuk menyederhanakan perangkat keras *pipeline*, waktunya diatur dengan asumsi bahwa setiap instruksi memerlukan enam stage. Selain itu, diagram mengasumsikan bahwa semua stage dapat dilakukan secara paralel. Secara khusus, diasumsikan bahwa tidak ada konflik memori. Misalnya, tahapan FI, FO, dan WO melibatkan akses memori. Diagram menyiratkan bahwa semua akses ini dapat terjadi secara bersamaan. Sebagian besar sistem memori tidak akan mengizinkannya. Namun, nilai yang diinginkan mungkin dalam *cache*, atau tahap FO atau WO mungkin nol. Dengan demikian, sebagian besar waktu, konflik memori tidak akan memperlambat pipeline.

Beberapa faktor lain berfungsi membatasi peningkatan kinerja. Jika enam stage tidak memiliki durasi waktu yang sama, akan ada beberapa waiting yang terlibat di setiap stage. Kesulitan lain adalah instruksi cabang bersyarat, yang dapat membatalkan beberapa pengambilan instruksi. Peristiwa serupa yang tidak dapat diprediksi adalah Interupsi. Gambar 7.15 mengilustrasikan efek dari cabang bersyarat. Asumsikan instruksi-3 adalah cabang bersyarat untuk instruksi-15. Sampai instruksi dieksekusi, tidak ada cara untuk mengetahui instruksi yang akan datang berikutnya. Pipa, dalam contoh ini, cukup memuat instruksi berikutnya secara berurutan (instruksi 4) dan hasil.

Pada Gambar 7.14, tidak ada operasi percabangan, sehingga didapatkan kinerja penuh dari peningkatan tersebut. Pada Gambar 7.16, terdapat operasi percabangan. Ini tidak ditentukan sampai akhir waktu unit waktu ke 7. Pada titik ini, pipa harus dibersihkan dari instruksi yang tidak berguna. Selama unit waktu ke 8, instruksi 15 memasuki *pipeline*. Tidak ada instruksi yang lengkap selama unit waktu ke 9 hingga 12; ini adalah penalti kinerja yang dihasilkan karena tidak dapat mengantisipasi operasi percabangan. Gambar 7.15 menunjukkan logika yang diperlukan pada pipelining untuk memperhitungkan operasi percabangan dan interupsi.



Sumber: (Stallings, 2016)

Gambar 7.15 Flowchart 6 Stage Pipeline Instruksi

	Time														Branch penalty	
	1	2	3	4	5	6	7	8	9	10	11	12	13	14		
Instruction 1	FI	DI	CO	FO	EI	WO										
Instruction 2		FI	DI	CO	FO	EI	WO									
Instruction 3			FI	DI	CO	FO	EI	WO								
Instruction 4				FI	DI	CO	FO									
Instruction 5					FI	DI	CO									
Instruction 6						FI	DI									
Instruction 7							FI									
Instruction 15								FI	DI	CO	FO	EI	WO			
Instruction 16								FI	DI	CO	FO	EI	WO			

Sumber: (Stallings, 2016)

Gambar 7.16. Efek dari Cabang Bersyarat pada Operasi Pipeline Instruksi

Untuk memperjelas operasi *pipeline*, Gambar 7.14 dan 7.16 menunjukkan perkembangan waktu secara horizontal melintasi waktu (time), dengan setiap baris menunjukkan kemajuan instruksi

individu. Gambar 7.17 menunjukkan urutan kejadian yang sama, dengan waktu bergerak secara vertikal ke bawah gambar, dan setiap baris menunjukkan keadaan pipa pada titik waktu tertentu. Pada Gambar 7.17a (yang sesuai dengan Gambar 7.14), pipa penuh pada waktu 6, dengan 6 instruksi berbeda dalam berbagai tahap pelaksanaan, dan tetap penuh sepanjang waktu 9; diasumsikan bahwa instruksi I9 adalah instruksi terakhir yang akan dieksekusi. Pada Gambar 7.17b, (yang sesuai dengan Gambar 7.14), pipa penuh pada waktu 6 dan 7. Pada waktu 7, instruksi 3 sedang dalam tahap eksekusi dan mengeksekusi cabang ke instruksi 15. Pada titik ini, instruksi I4 hingga I7 memerlukan waktu eksekusi yang lama, sehingga pada waktu 8, hanya dua instruksi dalam pipa, I3 dan I15.

The diagram consists of two tables, (a) and (b), showing the state of a pipeline over 14 time steps. A vertical arrow on the left labeled 'Time' points downwards, indicating the progression of time from step 1 to 14. The columns represent stages: FI (Instruction Fetch), DI (Instruction Decode), CO (Control Output), FO (Forwarding), EI (Execution), and WO (Write-Back).

(a) No branches:

Time	FI	DI	CO	FO	EI	WO
1	I1					
2	I2	I1				
3	I3	I2	I1			
4	I4	I3	I2	I1		
5	I5	I4	I3	I2	I1	
6	I6	I5	I4	I3	I2	I1
7	I7	I6	I5	I4	I3	I2
8	I8	I7	I6	I5	I4	I3
9	I9	I8	I7	I6	I5	I4
10		I9	I8	I7	I6	I5
11			I9	I8	I7	I6
12				I9	I8	I7
13					I9	I8
14						I9

(b) With conditional branch:

Time	FI	DI	CO	FO	EI	WO
1	I1					
2	I2	I1				
3	I3	I2	I1			
4	I4	I3	I2	I1		
5	I5	I4	I3	I2	I1	
6	I6	I5	I4	I3	I2	I1
7	I7	I6	I5	I4	I3	I2
8	I15					I3
9	I16	I15				
10		I16	I15			
11			I16	I15		
12				I16	I15	
13					I16	I15
14						I16

Sumber: (Stallings, 2016)

Gambar 7.17. Alternatif Penggambaran Pipeline

Dari diskusi sebelumnya, mungkin tampak bahwa semakin besar jumlah tahapan dalam pipa, semakin cepat laju eksekusi. Beberapa perancang IBM S/360 menunjukkan dua faktor yang menggagalkan pola yang tampaknya sederhana ini untuk desain berkinerja tinggi, dan mereka tetap merupakan elemen yang masih harus dipertimbangkan perancang:

- 1) Pada setiap tahap pipa, ada beberapa *overhead* yang terlibat dalam memindahkan data dari buffer ke buffer dan dalam melakukan berbagai fungsi persiapan dan pengiriman. *Overhead* ini dapat memperpanjang total waktu eksekusi dari satu instruksi. Ini penting ketika instruksi berurutan tergantung secara logika, baik melalui penggunaan percabangan yang berat atau melalui dependensi akses memori.
- 2) Jumlah logika kontrol yang diperlukan untuk menangani memori dan register dependensi dan untuk mengoptimalkan penggunaan pipa meningkat sangat besar dengan jumlah tahapan. Hal ini dapat mengarah pada situasi di mana logika mengendalikan gerbang antara tahap lebih kompleks daripada tahap yang dikendalikan.

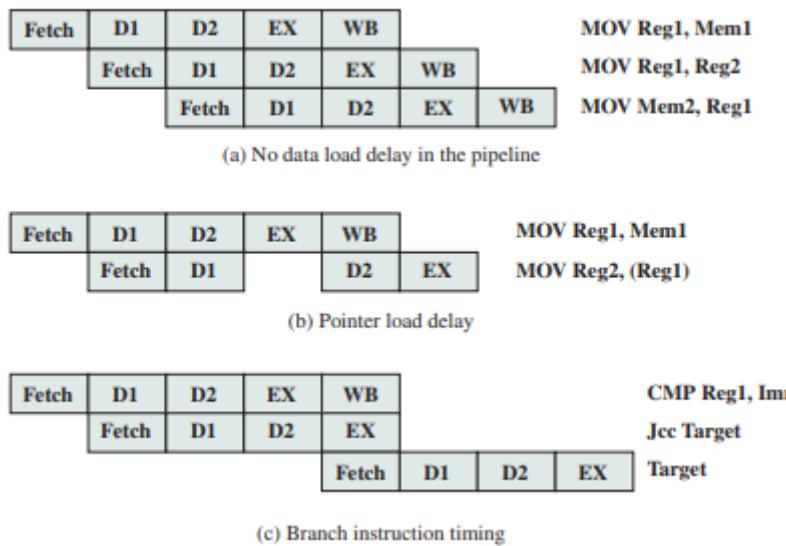
Pertimbangan lain adalah penundaan penguncian: Dibutuhkan waktu untuk penyangga pipa untuk beroperasi dan ini menambah waktu siklus instruksi.

Intel 80486 Pipelining

Contoh instruktif dari *pipeline* instruksi adalah dari Intel 80486, yang mengimplementasikan lima stage *pipeline*:

- 1) **Fetch:** Instruksi diambil dari *cache* atau dari memori eksternal dan ditempatkan ke dalam salah satu dari dua buffer prefetch 16-byte. Tujuan dari *fetch stage* adalah untuk mengisi buffer prefetch dengan data baru segera setelah data lama telah dikonsumsi oleh decoder instruksi. Karena instruksi memiliki panjang variabel (dari 1 hingga 11 byte tidak termasuk awalan), status prefetcher relatif terhadap stage *pipeline* lainnya bervariasi dari instruksi ke instruksi. Rata-rata, sekitar lima instruksi diambil dengan masing-masing 16 byte.
- 2) **Load.** *Load stage* beroperasi secara independen dari stage lainnya untuk menjaga buffer prefetch penuh.
- 3) **Decode 1:** Semua informasi opcode dan mode pengalamanan diterjemahkan dalam stage D1. Informasi yang diperlukan, serta informasi panjang instruksi, termasuk paling banyak 3 byte pertama instruksi. Oleh karena itu, 3 byte dilewatkan ke stage D1 dari buffer prefetch. Dekoder D1 kemudian dapat mengarahkan stage D2 untuk menangkap sisa instruksi (perpindahan dan data langsung), yang tidak terlibat dalam dekode D1.
- 4) **Decode 2:** Stage D2 memperluas setiap opcode menjadi sinyal kontrol untuk ALU. Ini juga mengontrol perhitungan mode pengalamanan yang lebih kompleks.
- 5) **Jalankan:** Stage ini mencakup operasi ALU, akses *cache*, dan pembaruan register.
- 6) **Tulis kembali:** Stage ini, jika perlu, memperbarui register dan tanda status yang dimodifikasi selama stage eksekusi sebelumnya. Jika instruksi saat ini memperbarui memori, nilai yang dihitung dikirim ke *cache* dan ke buffer penulisan antarmuka-bus pada saat yang sama.

Dengan menggunakan dua stage decode, *pipeline* dapat mempertahankan throughput mendekati satu instruksi per siklus *clock*. Instruksi kompleks dan cabang kondisional dapat memperlambat laju ini.



Sumber: (Stallings, 2016)
Gambar 7.18. Contoh *Pipeline* Instruksi 80486

Gambar 7.18 menunjukkan contoh operasi *pipeline*. Gambar 7.18a menunjukkan bahwa tidak ada penundaan yang dimasukkan ke dalam pipa ketika akses memori diperlukan. Namun, seperti yang ditunjukkan Gambar 7.18b, mungkin ada penundaan untuk nilai yang digunakan untuk menghitung alamat memori. Yaitu, jika suatu nilai dimuat dari memori ke dalam register dan register itu kemudian

digunakan sebagai register dasar dalam instruksi berikutnya, prosesor akan berhenti selama satu siklus. Dalam contoh ini, prosesor mengakses *cache* pada tahap EX dari instruksi pertama dan menyimpan nilai yang diambil dalam register selama tahap WB. Namun, instruksi selanjutnya membutuhkan register ini dalam tahap D2. Ketika tahap D2 sejalan dengan tahap WB dari instruksi sebelumnya, jalur sinyal pintas memungkinkan tahap D2 untuk memiliki akses ke data yang sama yang digunakan oleh tahap WB untuk menulis, menghemat satu tahap pipa. Gambar 7.18c mengilustrasikan waktu instruksi cabang, dengan asumsi bahwa cabang diambil. Instruksi perbandingan memperbaharui kode kondisi dalam tahap WB, dan jalur bypass membuatnya tersedia untuk tahap EX dari instruksi lompatan pada saat yang sama. Secara paralel, prosesor menjalankan siklus pengambilan spekulatif ke target lompatan selama tahap EX dari instruksi lompatan. Jika prosesor menentukan kondisi cabang palsu, ia membuang prefetch ini dan melanjutkan eksekusi dengan instruksi berurutan berikutnya (sudah diambil dan didekodekan).

8. Set Instruksi

8.1 Karakteristik Instruksi Mesin

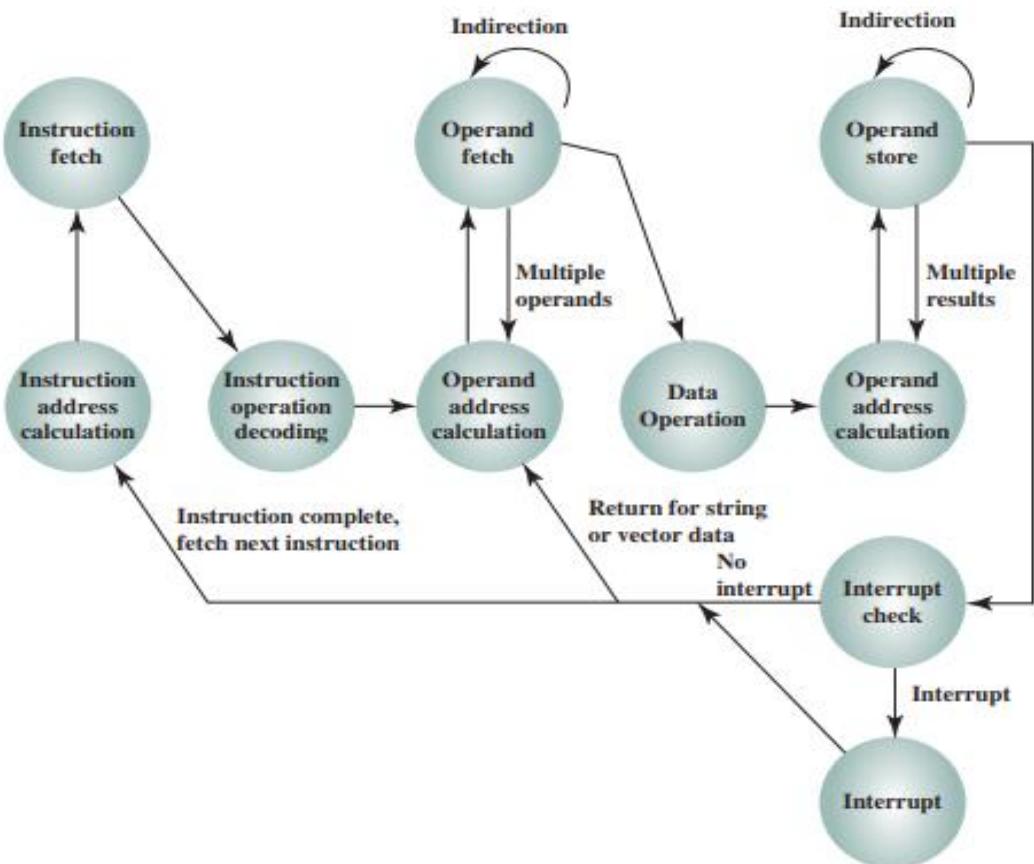
Pengoperasian prosesor ditentukan oleh instruksi yang dijalankannya, disebut sebagai instruksi mesin. Kumpulan instruksi berbeda yang dapat dijalankan prosesor disebut instruction set. Setiap instruksi harus mengandung informasi yang diperlukan oleh prosesor untuk eksekusi. Gambar 8.1, menunjukkan langkah-langkah yang terlibat dalam eksekusi instruksi dan implikasinya, serta mendefinisikan elemen-elemen dari instruksi mesin. Elemen-elemen ini adalah sebagai berikut:

- 1) **Operation Code**: Menentukan operasi yang akan dilakukan (misalnya: ADD, I/O). Operasi ditentukan oleh kode biner, yang dikenal sebagai kode operasi, atau opcode.
- 2) **Source operand reference**: Operasi dapat melibatkan satu atau lebih *operand* sumber, yaitu *operand* yang merupakan *input* untuk operasi.
- 3) **Result operand reference**: Operasi dapat menghasilkan hasil.
- 4) **Next instruction reference**: Ini memberi tahu prosesor tempat mengambil instruksi selanjutnya setelah eksekusi instruksi ini selesai.

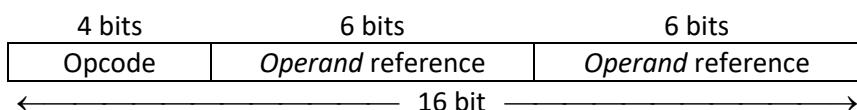
Alamat instruksi selanjutnya yang akan diambil bisa berupa alamat asli atau alamat virtual, tergantung arsitekturnya. Secara umum, perbedaannya transparan untuk arsitektur set instruksi. Dalam kebanyakan kasus, instruksi selanjutnya yang harus diambil segera mengikuti instruksi saat ini. Dalam kasus tersebut, tidak ada referensi eksplisit untuk instruksi selanjutnya. Ketika referensi eksplisit diperlukan, maka memori utama atau alamat memori virtual harus diberikan.

Operand sumber dan hasil dapat di salah satu dari empat bidang:

- 1) **Memori utama atau virtual**: Seperti dengan referensi instruksi berikutnya, alamat memori utama atau virtual harus diberikan.
- 2) **Register prosesor**: Dengan pengecualian yang jarang, prosesor berisi satu atau lebih register yang mungkin dirujuk oleh instruksi mesin. Jika hanya ada satu register, referensi untuk itu mungkin implisit. Jika ada lebih dari satu register, maka setiap register diberi nama atau nomor yang unik, dan instruksi harus berisi nomor register yang diinginkan.
- 3) **Segera (*Immediate*)**: Nilai *operand* terdapat dalam bidang dalam instruksi yang dieksekusi.
- 4) **Perangkat I/O**: Instruksi harus menentukan modul dan perangkat I/O untuk operasi. Jika I/O yang dipetakan dengan memori digunakan, ini hanyalah alamat memori utama atau virtual lainnya.



Sumber: (Stallings, 2016)
Gambar 8.1. State Diagram Siklus Instruksi



Sumber: (Stallings, 2016)
Gambar 8.2. Format isntruksi 16 bit sederhana

A. Representasi Instruksi

Di dalam komputer, setiap instruksi diwakili oleh urutan bit. Instruksi dibagi menjadi bidang-bidang, sesuai dengan elemen-elemen penyusun instruksi. Contoh sederhana dari format instruksi ditunjukkan pada Gambar 8.2. Sebagai contoh lain, format instruksi IAS ditunjukkan pada Gambar 1.9. Dengan sebagian besar set instruksi, menggunakan lebih dari satu format instruksi.

Selama eksekusi instruksi, instruksi dibaca ke dalam *Instruction Register* (IR) dalam prosesor. Prosesor harus dapat mengekstraksi data dari berbagai bidang instruksi untuk melakukan operasi yang diperlukan. Sulit bagi programmer dan pembaca buku teks untuk berurusan dengan representasi biner dari instruksi mesin. Dengan demikian, sudah menjadi praktik umum untuk menggunakan representasi simbolis dari instruksi mesin. Contoh ini digunakan untuk set instruksi IAS, pada Tabel 1.1.

Opcode diwakili oleh singkatan, yang disebut mnemonik, yang menunjukkan operasi. Contoh opcode:

ADD	Add
	Subtract

MUL	Multiply
DIV	Divide
LOAD	Load data from memory
STOR	Store data to memory

Operand juga direpresentasikan secara simbolis. Sebagai contoh,

ADD R,Y

instruksi tersebut dapat berarti menambahkan nilai yang terkandung dalam lokasi data Y ke isi register R. Dalam contoh ini, Y merujuk ke alamat lokasi dalam memori, dan R mengacu pada register tertentu. Perhatikan bahwa operasi dilakukan pada konten lokasi, bukan pada alamatnya.

Dengan demikian, dimungkinkan untuk menulis program bahasa mesin dalam bentuk simbolis. Setiap opcode simbolik memiliki representasi biner tetap, dan programmer menentukan lokasi setiap *operand* simbolik. Sebagai contoh, programmer dapat memulai dengan daftar definisi:

X = 513
Y = 514

dan seterusnya. Sebuah program sederhana akan menerima *input* simbolik ini, mengonversi opcodes dan referensi *operand* ke bentuk biner, dan membuat instruksi mesin biner. Pemrogram bahasa mesin jarang sampai tidak ada. Sebagian besar program saat ini ditulis dalam bahasa tingkat tinggi atau, jika tidak, bahasa assembly. Namun, bahasa mesin simbolik tetap merupakan alat yang berguna untuk menggambarkan instruksi mesin, dan akan menggunakannya untuk tujuan itu.

B. Type Instruksi

Pertimbangkan instruksi bahasa tingkat tinggi yang dapat diekspresikan dalam bahasa seperti BASIC atau FORTRAN. Sebagai contoh,

X = X + Y

Pernyataan ini menginstruksikan komputer untuk menambahkan nilai yang disimpan dalam Y ke nilai yang disimpan dalam X dan memasukkan hasilnya dalam X. Diasumsikan bahwa variabel X dan Y sesuai dengan lokasi 513 dan 514. Untuk set instruksi mesin yang sederhana, operasi ini dapat diselesaikan dengan tiga instruksi mesin:

- 1) Muat register dengan isi lokasi memori 513.
- 2) Tambahkan konten lokasi memori 514 ke register.
- 3) Simpan konten register di lokasi memori 513.

Instruksi BASIC tunggal mungkin memerlukan tiga instruksi mesin. Ini khas dari hubungan antara bahasa tingkat tinggi dan bahasa mesin. Bahasa tingkat tinggi mengekspresikan operasi dalam bentuk aljabar singkat, menggunakan variabel. Bahasa mesin mengekspresikan operasi dalam bentuk dasar yang melibatkan perpindahan data ke atau dari register.

Komputer harus memiliki seperangkat instruksi yang memungkinkan pengguna untuk merumuskan tugas pemrosesan data apa pun. Cara lain untuk melihatnya adalah dengan mempertimbangkan kemampuan bahasa pemrograman tingkat tinggi. Setiap program yang ditulis dalam bahasa tingkat tinggi harus diterjemahkan ke dalam bahasa mesin untuk dieksekusi. Dengan demikian, seperangkat instruksi mesin harus cukup untuk mengekspresikan setiap instruksi dari bahasa tingkat tinggi. Dengan mengingat hal ini jenis instruksi dapat dikategorikan sebagai berikut:

- 1) Pemrosesan data (*Data processing*): Aritmatika dan instruksi logika
- 2) Penyimpanan data (*Data Storage*): Pergerakan data ke dalam atau ke luar dari register dan atau lokasi memori

- 3) Pergerakan data (*Data movement*): instruksi I/O
- 4) Kontrol aliran program (*Program flow control*): Test dan instruksi cabang

Instruksi aritmatika memberikan kemampuan komputasi untuk memproses data numerik. Instruksi logika (Boolean) beroperasi pada bit *word* sebagai bit daripada sebagai angka; dengan demikian, mereka menyediakan kemampuan untuk memproses semua jenis data lain yang pengguna mungkin ingin gunakan. Operasi ini dilakukan terutama pada data dalam register prosesor. Oleh karena itu, harus ada instruksi memori untuk memindahkan data antara memori dan register. Instruksi I/O diperlukan untuk mentransfer program dan data ke dalam memori dan hasil perhitungan kembali ke pengguna.

Instruksi test digunakan untuk menguji nilai *word* data atau status perhitungan. Instruksi cabang kemudian digunakan untuk melakukan cabang ke sejumlah instruksi berbeda tergantung pada keputusan yang dibuat.

C. Jumlah Alamat

Salah satu cara tradisional untuk menggambarkan arsitektur prosesor adalah dalam hal jumlah alamat yang terkandung dalam setiap instruksi. Dimensi ini menjadi kurang signifikan dengan meningkatnya kompleksitas desain prosesor. Berapa jumlah maksimum alamat yang diperlukan dalam suatu instruksi tergantung pada instruksi pada set instruksi. Instruksi aritmatika dan logika akan membutuhkan *operand* terbanyak. Hampir semua operasi aritmatika dan logika adalah unary (*operand* satu sumber) atau biner (dua *operand* sumber). Dengan demikian, dibutuhkan maksimal dua alamat untuk referensi *operand* sumber. Hasil operasi harus disimpan, menyarankan alamat ketiga, yang menentukan *operand* tujuan. Akhirnya, setelah menyelesaikan suatu instruksi, instruksi selanjutnya harus diambil, dan alamatnya diperlukan.

Alur penalaran ini menunjukkan bahwa suatu instruksi masuk akal mungkin diperlukan untuk memuat empat referensi alamat: dua *operand* sumber, satu *operand* tujuan, dan alamat instruksi berikutnya. Di sebagian besar arsitektur, sebagian besar instruksi memiliki satu, dua, atau tiga alamat *operand*, dengan alamat instruksi berikutnya yang tersirat (diperoleh dari register *Program Counter*). Sebagian besar arsitektur juga memiliki beberapa *special purpose instruction* dengan *operand* yang lebih banyak.

Tabel 8.1 membandingkan instruksi khas satu, dua, dan tiga alamat yang dapat digunakan untuk menghitung $Y=(A-B)/(C+DE)$.

Tabel 8.1. Program untuk mengeksekusi $Y=(A-B)/(C+DE)$

(a) Instruksi Tiga Alamat		(b) Instruksi Dua Alamat		(c) Instruksi Satu Alamat	
Instruction	Comment	Instruction	Comment	Instruction	Comment
SUB Y,A,B	$Y \leftarrow A - B$	MOVE Y,A	$Y \leftarrow A$	LOAD D	$AC \leftarrow D$
MPY T,D,E	$T \leftarrow D \times E$	SUB Y,B	$Y \leftarrow Y - B$	MPY E	$AC \leftarrow AC \times E$
ADD T,T,C	$T \leftarrow T + C$	MOVE T,D	$T \leftarrow D$	ADD C	$AC \leftarrow AC + C$
DIV Y,Y,T	$Y \leftarrow Y : T$	MPY T,E	$T \leftarrow T \times E$	STOR Y	$Y \leftarrow AC$
		ADD T,C	$T \leftarrow T + C$	LOAD A	$AC \leftarrow A$
		DIV Y,T	$Y \leftarrow Y : T$	SUB B	$AC \leftarrow AC - B$
				DIV Y	$AC \leftarrow AC : Y$

Sumber: (Stallings, 2016)

Dengan tiga alamat, setiap instruksi menentukan dua lokasi *operand* sumber dan lokasi *operand* tujuan. Karena nilai dari salah satu lokasi *operand* tidak berubah, lokasi sementara T, digunakan untuk menyimpan beberapa hasil antara. Terdapat empat instruksi dan ekspresi asli memiliki lima *operand*. Format instruksi tiga alamat ini tidak umum karena mereka memerlukan format instruksi yang relatif panjang untuk menampung referensi tiga alamat.

Dengan instruksi dua alamat, dan untuk operasi biner, satu alamat harus melakukan tugas ganda baik sebagai *operand* maupun hasilnya. Jadi, instruksi SUB Y, B melakukan perhitungan Y - B, dan menyimpan hasilnya di Y. Format instruksi ini mengurangi kebutuhan ruang tetapi juga memperkenalkan beberapa kecanggungan. Untuk menghindari mengubah nilai *operand*, instruksi MOVE digunakan untuk memindahkan salah satu nilai ke hasil atau lokasi sementara sebelum melakukan operasi. Dengan format dua alamat, program sampel bertambah hingga enam instruksi.

Lebih sederhana lagi adalah instruksi satu alamat. Agar ini berfungsi, alamat kedua harus implisit. Ini umum di mesin sebelumnya, dengan alamat tersirat menjadi register prosesor yang dikenal sebagai register *Acumulator* (AC). Akumulator berisi salah satu *operand* dan digunakan untuk menyimpan hasilnya. Dalam contoh diatas, delapan instruksi diperlukan untuk menyelesaikan tugas.

Pada kenyataannya, dimungkinkan untuk puas dengan nol alamat untuk beberapa instruksi. Instruksi alamat-nol berlaku untuk organisasi memori khusus, yang disebut *stack*. *Stack* adalah struktur memori *First In Last Out*. *Stack* berada di lokasi yang sudah ditentukan sebelumnya dan, seringkali, paling tidak dua elemen teratas ada di register prosesor.

Tabel 8.2. Pemanfaatan Alamat Instruksi Bukan Percabangan

Jumlah Alamat	Representasi Simbolik	Interpretasi
3	OP A,B,C	A \leftarrow B OP C
2	OP A,B	A \leftarrow A OP B
1	OP A	AC \leftarrow AC OP A
0	OP	A \leftarrow (T-1) OP T

Ket:
 OP = Opcode
 AC = Akumulator
 T = puncak stack
 (T-1) = elemen kedua dari stack
 A, B, C = lokasi memori atau register

Sumber: (Stallings, 2016)

Tabel 8.2 merangkum interpretasi untuk ditempatkan pada instruksi dengan nol, satu, dua, atau tiga alamat. Dalam setiap kasus dalam tabel, diasumsikan bahwa alamat instruksi selanjutnya adalah implisit, dan bahwa satu operasi dengan dua *operand* sumber dan satu *operand* hasil harus dilakukan. Jumlah alamat per instruksi adalah keputusan desain dasar. Lebih sedikit alamat per instruksi menghasilkan instruksi yang lebih primitif, membutuhkan prosesor yang kurang kompleks. Ini juga menghasilkan instruksi dengan panjang yang lebih pendek. Di sisi lain, program mengandung lebih banyak instruksi total, yang secara umum menghasilkan waktu eksekusi yang lebih lama dan lebih lama, program yang lebih kompleks. Juga, ada ambang batas yang penting antara instruksi satu alamat dan banyak alamat. Dengan instruksi satu alamat, programmer umumnya hanya memiliki satu register tujuan umum, akumulator. Dengan instruksi banyak alamat, biasanya memiliki beberapa register tujuan umum. Ini memungkinkan beberapa operasi dilakukan hanya pada register. Karena referensi register lebih cepat daripada referensi memori, ini mempercepat eksekusi. Untuk alasan fleksibilitas dan kemampuan untuk menggunakan banyak register, kebanyakan mesin kontemporer menggunakan campuran instruksi dua dan tiga alamat.

Ada masalah yang harus dipecahkan, apakah alamat referensi merupakan lokasi memori atau register. Jika memilih register sebagai referensi alamat, dengan jumlah register lebih sedikit, sehingga membutuhkan bit lebih sedikit untuk referensi register. Sebuah mesin mungkin menawarkan beragam mode pengalaman, dan spesifikasi mode membutuhkan satu atau lebih bit. Hasilnya adalah sebagian besar desain prosesor melibatkan berbagai format instruksi.

8.2. Type Operand.

Kategori umum data yang paling penting pada instruksi mesin adalah:

- 1) Alamat
- 2) Angka
- 3) Karakter
- 4) Data logika

Bidang alamat dalam *operand* pada instruksi, pada kenyataannya adalah suatu bentuk data. Dalam banyak kasus, beberapa perhitungan harus dilakukan pada referensi *operand* dalam instruksi untuk menentukan alamat memori utama atau virtual. Dalam konteks ini, alamat dapat dianggap sebagai bilangan bulat tidak bertanda (*unsigned integer*).

Tipe data umum lainnya adalah angka, karakter, dan data logika, dan masing-masing diperiksa secara singkat di bagian ini. Selain itu, beberapa mesin mendefinisikan tipe data khusus atau struktur data. Misalnya, mungkin ada operasi mesin yang beroperasi secara langsung pada register atau serangkaian karakter.

A. Type Data Numerik atau Angka

Semua bahasa mesin termasuk tipe data numerik. Bahkan dalam pemrosesan data nonnumerik, ada kebutuhan untuk angka untuk bertindak sebagai penghitung, lebar bidang, dan sebagainya. Perbedaan penting antara angka yang digunakan dalam matematika biasa dan angka yang disimpan dalam komputer adalah bahwa yang terakhir terbatas. Ini benar dalam dua pengertian. Pertama, ada batas untuk besarnya angka yang dapat diwakili pada mesin dan kedua, dalam kasus angka *floating point*, batas ketepatannya. Oleh karena itu, programmer dihadapkan dengan pemahaman konsekuensi dari pembulatan, luapan, dan luapan. Tiga jenis data numerik umum di komputer:

- 1) Biner integer atau *fixed point biner*
- 2) *Binary floating point*
- 3) Desimal

Meskipun semua operasi komputer internal bersifat biner, pengguna manusia dari sistem ini berurusan dengan angka desimal. Jadi, ada kebutuhan untuk mengkonversi dari desimal ke biner pada *input* dan dari biner ke desimal pada *output*. Untuk aplikasi di mana ada banyak I/O dan relatif sedikit, perhitungan yang relatif sederhana, lebih disukai untuk menyimpan dan beroperasi pada angka dalam bentuk desimal. Representasi yang paling umum untuk tujuan ini adalah ***packed decimal***.

Dengan ***packed decimal***, setiap digit desimal 0 sampai 9 direpresentasikan oleh kode 4-bit, sebagai berikut: 0=0000; 1=0001, 2=0010,..., 8=1000, dan 9=1001. Dua digit desimal akan disimpan dalam satu *byte*. Perhatikan bahwa ini adalah kode yang agak tidak efisien karena hanya 10 dari 16 nilai 4-bit yang mungkin digunakan. Untuk membentuk angka, kode 4-bit dirangkai, biasanya dalam kelipatan 8 bit. Dengan demikian, kode untuk 246 adalah 0000 0010 0100 0110. Kode ini jelas kurang kompak daripada representasi biner normal, tetapi menghindari *overhead* konversi. Angka negatif dapat direpresentasikan dengan menyertakan digit tanda 4-bit di ujung kiri atau kanan rangkaian angka desimal yang dikemas. Nilai tanda standar adalah 1100 untuk positif dan 1101 untuk negatif. Banyak mesin memberikan instruksi aritmatika untuk melakukan operasi langsung pada angka desimal yang dikemas.

B. Type Data Karakter

Kode karakter yang paling umum digunakan dalam *International Reference Alphabet* (IRA) adalah ASCII (*American Standard Code for Information Interchange*). Kode ASCII mewakili karakter karakter alfanumerik dalam memori sistem komputer. Pada awalnya format data yang digunakan adalah 7 bit dengan bit ke-8 sebagai MSB (*Most Significant Bit*) yang digunakan untuk memuat parity dalam beberapa sistem, bit diatur sedemikian rupa sehingga jumlah biner 1 di setiap oktet selalu ganjil

(paritas ganjil) atau selalu genap (paritas genap). Jika data ASCII digunakan dengan sebuah printer, maka MSB adalah 0 pada pencetakan alfanumerik dan 1 pada pencetakan grafik.

Kumpulan karakter ASCII menggunakan kode 00H-7FH atau 0-127 Desimal, seperti disajikan pada tabel 8.3. Kemudian Kode ASCII diperluas lagi menjadi Extended ASCII dengan penambahan kode 80H-FFH atau 128-255 Desimal, seperti ditunjukkan pada tabel 8.4. Karakter Extended ASCII menyimpan huruf-huruf asing dan tanda baca, karakter Greek (Yunani) karakter matematika, karakter *box-drawing*, serta karakter khusus lainnya. Kode selanjutnya 0100H-FFFFH disebut dengan *universal code (unicode)* digunakan untuk menyimpan karakter khusus dari semua kumpulan karakter di dunia.

Tabel 8.3. Karakter ASCII

Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
0	00	Null	32	20	Space	64	40	Ø	96	60	`
1	01	Start of heading	33	21	!	65	41	A	97	61	a
2	02	Start of text	34	22	"	66	42	B	98	62	b
3	03	End of text	35	23	#	67	43	C	99	63	c
4	04	End of transmit	36	24	\$	68	44	D	100	64	d
5	05	Enquiry	37	25	%	69	45	E	101	65	e
6	06	Acknowledge	38	26	&	70	46	F	102	66	f
7	07	Audible bell	39	27	'	71	47	G	103	67	g
8	08	Backspace	40	28	(72	48	H	104	68	h
9	09	Horizontal tab	41	29)	73	49	I	105	69	i
10	0A	Line feed	42	2A	*	74	4A	J	106	6A	j
11	0B	Vertical tab	43	2B	+	75	4B	K	107	6B	k
12	0C	Form feed	44	2C	,	76	4C	L	108	6C	l
13	0D	Carriage return	45	2D	-	77	4D	M	109	6D	m
14	0E	Shift out	46	2E	.	78	4E	N	110	6E	n
15	0F	Shift in	47	2F	/	79	4F	O	111	6F	o
16	10	Data link escape	48	30	Ø	80	50	P	112	70	p
17	11	Device control 1	49	31	1	81	51	Q	113	71	q
18	12	Device control 2	50	32	2	82	52	R	114	72	r
19	13	Device control 3	51	33	3	83	53	S	115	73	s
20	14	Device control 4	52	34	4	84	54	T	116	74	t
21	15	Neg. acknowledge	53	35	5	85	55	U	117	75	u
22	16	Synchronous idle	54	36	6	86	56	V	118	76	v
23	17	End trans. block	55	37	7	87	57	W	119	77	w
24	18	Cancel	56	38	8	88	58	X	120	78	x
25	19	End of medium	57	39	9	89	59	Y	121	79	y
26	1A	Substitution	58	3A	:	90	5A	Z	122	7A	z
27	1B	Escape	59	3B	;	91	5B	[123	7B	{
28	1C	File separator	60	3C	<	92	5C	\	124	7C	
29	1D	Group separator	61	3D	=	93	5D]	125	7D	}
30	1E	Record separator	62	3E	>	94	5E	^	126	7E	~
31	1F	Unit separator	63	3F	?	95	5F	_	127	7F	□

Sumber: (Brey, 2009)

Secara umum karakter yang ada pada ASCII dikelompokan menjadi:

- 1) Karakter kendali, adalah karakter tidak tercetak yang berfungsi melaksanakan fungsi kontrol dalam sistem komputer, termasuk clear screen, back space, line feed, dan lainnya.

- 2) Karakter alfanumerik, terdiri dari karakter simbol bilangan desimal dan abjad a-z serta A-Z.
- 3) Spesial karakter, meliputi simbol yang digunakan pada tanda baca dan matematika.
- 4) Karakter *drawing*, merupakan pengembangan pada extended ASCII meliputi simbol-simbol yang digunakan untuk membentuk kotak (box-drawing) dan karakter icon yang sering digunakan secara umum.

Kode lain yang digunakan untuk menyandikan karakter adalah *Extended Binary Coded Decimal Interchange Code* (EBCDIC). EBCDIC digunakan pada mainframe IBM. Ini adalah kode 8-bit. Seperti halnya IRA, EBCDIC kompatibel dengan desimal paket. Dalam kasus EBCDIC, kode 11110000 hingga 11111001 mewakili angka 0 hingga 9.

Tabel 8.4. Karakter Extended ASCII.

Dec	Hex	Char									
128	80	ç	160	A0	á	192	C0	ł	224	E0	α
129	81	ü	161	A1	í	193	C1	ł	225	E1	ß
130	82	é	162	A2	ó	194	C2	τ	226	E2	Γ
131	83	à	163	A3	ú	195	C3	†	227	E3	π
132	84	ä	164	A4	ñ	196	C4	—	228	E4	Σ
133	85	à	165	A5	Ñ	197	C5	+	229	E5	σ
134	86	ã	166	A6	²	198	C6	ƒ	230	E6	µ
135	87	ç	167	A7	°	199	C7	॥	231	E7	τ
136	88	ê	168	A8	ç	200	C8	ܼ	232	E8	Φ
137	89	ë	169	A9	߱	201	C9	߱	233	E9	߰
138	8A	è	170	AA	߱	202	CA	߲	234	EA	߳
139	8B	ି	171	AB	ୟ୍ୟ	203	CB	ୟ୍ୟ	235	EB	ୟ୍ୟ
140	8C	ି	172	AC	ୟ୍ୟ	204	CC	ୟ୍ୟ	236	EC	ୟ୍ୟ
141	8D	ି	173	AD	ି	205	CD	=	237	ED	ୟ୍ୟ
142	8E	ା	174	AE	ୟ୍ୟ	206	CE	ୟ୍ୟ	238	EE	ୟ୍ୟ
143	8F	ା	175	AF	ୟ୍ୟ	207	CF	ୟ୍ୟ	239	EF	ୟ୍ୟ
144	90	ି	176	B0	ୟ୍ୟ	208	DO	ୟ୍ୟ	240	F0	ୟ୍ୟ
145	91	ାୟ	177	B1	ୟ୍ୟ	209	D1	ୟ୍ୟ	241	F1	ୟ୍ୟ
146	92	ାୟ	178	B2	ୟ୍ୟ	210	D2	ୟ୍ୟ	242	F2	ୟ୍ୟ
147	93	ାୟ	179	B3	ି	211	D3	ୟ୍ୟ	243	F3	ୟ୍ୟ
148	94	ାୟ	180	B4	ି	212	D4	ୟ୍ୟ	244	F4	ି
149	95	ାୟ	181	B5	ି	213	D5	ୟ୍ୟ	245	F5	ି
150	96	ାୟ	182	B6	ି	214	D6	ୟ୍ୟ	246	F6	ି
151	97	ାୟ	183	B7	ି	215	D7	ି	247	F7	ି
152	98	ାୟ	184	B8	ି	216	D8	ି	248	F8	ି
153	99	ାୟ	185	B9	ି	217	D9	ି	249	F9	ି
154	9A	ୟ୍ୟ	186	BA	ି	218	DA	ି	250	FA	ି
155	9B	ାୟ	187	BB	ି	219	DB	ି	251	FB	ି
156	9C	ାୟ	188	BC	ି	220	DC	ି	252	FC	ି
157	9D	ାୟ	189	BD	ି	221	DD	ି	253	FD	ି
158	9E	ୟ୍ୟ	190	BE	ି	222	DE	ି	254	FE	ି
159	9F	ି	191	BF	ି	223	DF	ି	255	FF	ି

Sumber: (Brey, 2009)

C. Type Data Logika

Biasanya, setiap *word* atau unit *addressable unit* lainnya (*byte*, *halfword*, dan sebagainya) diperlakukan sebagai satu unit data. Namun, kadang-kadang berguna untuk mempertimbangkan unit n-bit yang terdiri dari n item 1-bit data, setiap item memiliki nilai 0 atau 1. Ketika data dilihat dengan cara ini, mereka dianggap sebagai data logika.

Ada dua keuntungan dari tampilan berorientasi bit. Pertama, terkadang mungkin ingin menyimpan array item data Boolean atau biner, di mana setiap item hanya dapat mengambil nilai 1 (benar) dan 0 (salah). Dengan data logika, memori dapat digunakan paling efisien untuk penyimpanan ini. . Kedua, ada saat-saat ketika ingin memanipulasi bit-item data. Sebagai contoh, jika operasi *floating-point* diimplementasikan dalam perangkat lunak, harus dapat menggeser bit yang signifikan dalam beberapa operasi. Contoh lain: Untuk mengkonversi dari IRA ke desimal paket, maka perlu mengekstrak 4 bit paling kanan dari setiap *byte*.

Perhatikan bahwa, dalam contoh sebelumnya, data yang sama kadang-kadang diperlakukan sebagai logika dan kadang-kadang sebagai angka atau teks. "Jenis" unit data ditentukan oleh operasi yang dilakukan di atasnya. Meskipun ini biasanya tidak terjadi dalam bahasa tingkat tinggi, hampir selalu demikian halnya dengan bahasa mesin.

8.3. Type Operasi

Jumlah opcode sangat bervariasi dari mesin ke mesin, namun jenis operasi khas yang sama, ditemukan pada hampir semua mesin. Kategorisasi tipe operasi yang khas adalah sebagai berikut:

- 1) Data transfer
- 2) Aritmatika
- 3) Logika
- 4) Konversi
- 5) I/O
- 6) Transfer kendali
- 7) Kontrol sistem

Tabel 8.5 menyajikan type instruksi yang umum pada masing-masing kategori.

Tabel 8.5. Type Operasi Dasar

Type	Nama Operasi	Deskripsi
Data transfer	<i>Move (Transfer)</i>	Mentransfer word atau block dari sumber ke tujuan
	<i>Store</i>	Mentransfer word dari prosesor ke memori
	<i>Exchange</i>	Menukar isi (konten) dari sumber ke tujuan
	<i>Clear (Reset)</i>	Mentransfer word 0 ke tujuan
	<i>Set</i>	Mentransfer word 1 ke tujuan
	<i>Push</i>	Mentransfer word dari sumber ke puncak stack memori
	<i>Pop</i>	Mentransfer word dari puncak stack memori ke sumber
Aritmatika	<i>Add</i>	Menghitung jumlah dari dua <i>operand</i>
	<i>Subtract</i>	Menghitung selisih dari dua <i>operand</i>
	<i>Multiply</i>	Menghitung hasil kali dua <i>operand</i>
	<i>Divide</i>	Menghitung pembagian dua <i>operand</i>
	<i>Absolute</i>	Mengganti isi <i>operand</i> dengan nilai absolute
	<i>Negate</i>	Mengubah tanda dari <i>operand</i>
	<i>Increment</i>	Menambahkan 1 pada <i>operand</i>
	<i>Decrement</i>	Mengurangi 1 dari <i>operand</i>
Logika	<i>AND</i>	Membentuk operasi logika AND
	<i>OR</i>	Membentuk operasi logika OR

	<i>NOT (complement)</i>	Membentuk operasi logika NOT
	<i>XOR</i>	Membentuk operasi logika XOR
	<i>Test</i>	Tes kondisi yang ditentukan; atur bendera berdasarkan hasil
	<i>Compare</i>	Buat perbandingan logika atau aritmatika dari dua atau lebih <i>operand</i> ; atur bendera berdasarkan hasil
	<i>Set Control Variable</i>	Kelas instruksi untuk mengatur kontrol untuk tujuan perlindungan, penanganan interupsi, kontrol <i>timer</i> , dll.
	<i>Shift</i>	menggeser isi <i>operand</i> ke kiri/kanan, menampilkan konstanta akhir yang dihasilkan
	<i>Rotate</i>	Memutar isi <i>operand</i> , dengan ujung sampul
Konversi	<i>Translate</i>	Menerjemahkan nilai dalam bagian memori berdasarkan pada tabel korespondensi.
	<i>Convert</i>	Mengonversi konten <i>word</i> dari satu bentuk ke bentuk lainnya (misalnya: packed-Desimal menjadi biner).
I/O	<i>Input (Read)</i>	Mentransfer data dari I/O port atau perangkat tertentu ke tujuan (misalnya: Memori utama atau register prosesor).
	<i>Output (Write)</i>	Transfer data dari sumber tertentu ke I/O port atau perangkat.
	<i>Start I/O</i>	Transfer instruksi ke prosesor I/O untuk memulai operasi I/O
	<i>Test I/O</i>	Mentransfer informasi status dari sistem I/O ke tujuan yang ditentukan
Transfer of Control	<i>Jump (branch)</i>	Transfer tanpa syarat; memuat PC dengan alamat yang ditentukan
	<i>Jump Conditional</i>	Tes kondisi yang ditentukan; baik memuat register <i>Program Counter</i> dengan alamat yang ditentukan atau tidak melakukan apa-apa, berdasarkan kondisi
	<i>Jump to Subroutine</i>	Tempatkan informasi kontrol program saat ini di lokasi yang diketahui pada <i>stack</i> ; lompat ke alamat yang ditentukan.
	<i>Return</i>	Ganti isi register PC dan register lainnya dari lokasi yang diketahui (<i>stack</i>)
	<i>Execute</i>	Ambil <i>operand</i> dari lokasi yang ditentukan dan jalankan sebagai instruksi; jangan modifikasi register <i>Program Counter</i>
	<i>Skip</i>	Increment register <i>Program Counter</i> untuk melewati instruksi selanjutnya
	<i>Skip Conditional</i>	Tes kondisi yang ditentukan; baik melewati atau melakukan apa pun berdasarkan kondisi
	<i>Halt</i>	Hentikan eksekusi program
	<i>Wait (hold)</i>	Hentikan eksekusi program; uji kondisi yang ditentukan berulang kali; melanjutkan eksekusi ketika kondisinya terpenuhi
	<i>No Operation</i>	Tidak ada operasi yang dilakukan, tetapi eksekusi program dilanjutkan.

Sumber: (Stallings, 2016)

A. Transfer data

Jenis instruksi mesin yang paling mendasar adalah instruksi transfer data. Instruksi transfer data harus menentukan beberapa hal. Pertama, lokasi *operand* sumber dan tujuan harus ditentukan. Setiap lokasi dapat berupa memori, register, atau bagian atas *stack*. Kedua, panjang data yang akan ditransfer harus ditunjukkan. Ketiga, seperti halnya semua instruksi dengan *operand*, mode pengalamatan untuk setiap *operand* harus ditentukan.

Pilihan instruksi transfer data untuk dimasukkan dalam set instruksi mencontohkan jenis pertukaran yang harus dibuat oleh perancang. Sebagai contoh, lokasi umum (memori atau register) dari suatu *operand* dapat ditunjukkan dalam spesifikasi opcode atau *operand*. Tabel 8.6 menunjukkan contoh instruksi transfer data IBM EAS/390 yang paling umum. Perhatikan bahwa ada varian untuk menunjukkan jumlah data yang akan ditransfer (8, 16, 32, atau 64 bit). Juga, ada instruksi berbeda

untuk register ke register, Memori ke memori, memori ke register, dan transfer memori ke memori. Sebaliknya, VAX memiliki instruksi move (MOV) dengan varian untuk jumlah data yang berbeda untuk dipindahkan, tetapi itu menentukan apakah register atau memori sebagai bagian dari *operand*. Pendekatan VAX agak lebih mudah bagi programmer, yang memiliki lebih sedikit mnemonik untuk ditangani. Namun, itu juga agak kurang kompak daripada pendekatan IBM EAS/390 karena lokasi (register versus memori) dari masing-masing *operand* harus ditentukan secara terpisah dalam instruksi.

Dalam hal eksekusi instruksi prosesor, operasi transfer data mungkin merupakan tipe yang paling sederhana. Jika sumber dan tujuan adalah register, maka prosesor hanya menyebabkan data ditransfer dari satu register ke register lainnya; ini adalah operasi internal ke prosesor. Jika satu atau kedua *operand* ada di memori, maka prosesor harus melakukan beberapa atau semua tindakan berikut:

- 1) Hitung alamat memori, berdasarkan mode alamat.
- 2) Jika alamat merujuk ke memori virtual, terjemahkan dari alamat virtual ke memori nyata.
- 3) Tentukan apakah item yang dialamatkan dalam cache.
- 4) Jika tidak, keluarkan perintah ke modul memori.

Tabel 8.6. Contoh Operasi Transfer Data dari IBM EAS/390

Mnemonik	Nama	Jumlah Bit Yang Ditransfer	Deskripsi
L	Load	32	Transfer dari memori ke register
LH	Load halfword	16	Transfer dari memori ke register
LR	Load	32	Transfer dari memori ke register
LER	Load (short)	32	Transfer dari floating point register ke <i>floating point</i> register
LE	Load (short)	32	Transfer dari memori ke <i>floating point</i> register
LDR	Load (long)	64	Transfer dari memori ke <i>floating point</i> register
LD	Load (long)	64	Transfer dari memori ke <i>floating point</i> register
ST	Store	32	Transfer dari register ke memori
STH	Store halfword	16	Transfer dari register ke memori
STC	Store character	8	Transfer dari register ke memori
STE	Store (short)	32	Transfer dari <i>floating point</i> register ke memori
STD	Store (long)	64	Transfer dari <i>floating point</i> register ke memori

Sumber: (Stallings, 2016)

B. Aritmatika

Sebagian besar mesin menyediakan operasi aritmatika dasar ADD, SUBTRACTION, MULTIPLY, dan DIVISION. Ini selalu disediakan untuk angka *signed integer (fix point)*. Seringkali mereka juga disediakan untuk *floating-point* dan angka desimal paket. Kemungkinan operasi lain termasuk berbagai instruksi *operand* tunggal; sebagai contoh,

- 1) **Absolute**: Ambil nilai absolut dari *operand*.
- 2) **Negate**: Negasikan *operand*.
- 3) **Increment**: Tambahkan 1 ke *operand*.
- 4) **Decrement**: Kurangi 1 dari *operand*.

C. Logika

Sebagian besar mesin juga menyediakan berbagai operasi *bitwise*, yaitu operasi logika untuk memanipulasi bit pada sebuah *word* atau unit yang dapat dialamatkan, sering disebut sebagai “*bit twiddling*”.

Tabel 8.7. Tabel Kebenaran Logika Dasar

P	Q	NOT P	P AND Q	P OR Q	P XOR Q	P=Q
0	0	1	0	0	0	1
0	1	1	0	1	1	0
1	0	0	0	1	1	0
1	1	0	1	1	0	1

Sumber: (Penulis, 2020)

Beberapa operasi logika dasar yang dapat dilakukan pada biner ditunjukkan pada Tabel 8.7. Operasi AND, OR, dan Exclusive-OR (XOR) adalah fungsi logika paling umum dengan dua *operand*, sedangkan operasi NOT hanya dengan satu *operand*. EQUAL adalah test logika biner yang digunakan untuk membuat keputusan. Operasi logika ini dapat diterapkan ke n-bit unit data logika. Misalnya, jika dua register R1 dan R2 berisi data:

$$(R1) = 10100101$$

$$(R2) = 00001111$$

Maka $(R1) \text{ AND } (R2) = 00000101$

di mana notasi (X) berarti isi lokasi X. Dengan demikian, operasi AND dapat digunakan sebagai *mask* yang memilih bit tertentu dalam *word* dan nol keluar bit yang tersisa. Sebagai contoh lain, jika dua register berisi

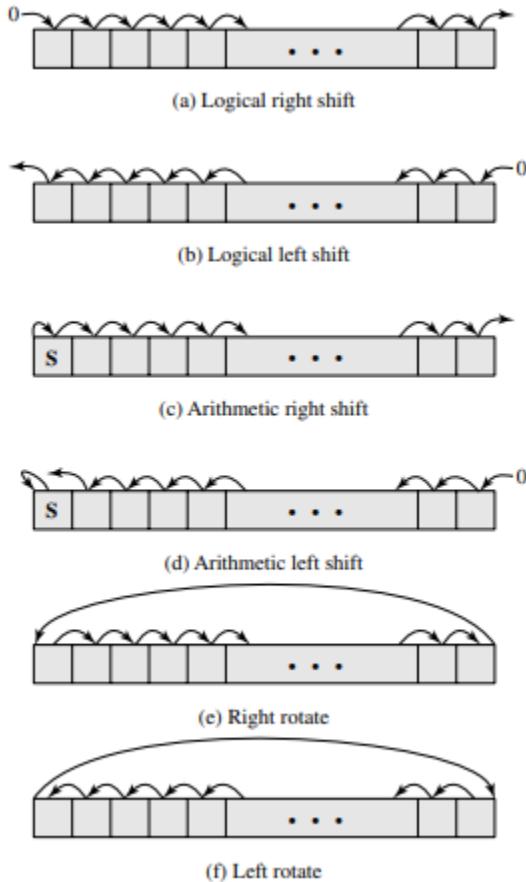
$$(R1) = 10100101$$

$$(R2) = 11111111$$

Maka $(R1) \text{ XOR } (R2) = 01011010$

Dengan satu *word* diset ke biner 1, operasi XOR membalikkan semua bit di *word* lain (komplemen satu).

Selain operasi logika *bitwise*, sebagian besar mesin menyediakan berbagai fungsi *shifting* dan *rotating*. Operasi paling dasar diilustrasikan dalam Gambar 8.3. Dengan perubahan logika, bit *word* digeser ke kiri atau kanan. Di satu sisi, bit yang bergeser hilang. Di sisi lain, 0 digeser masuk. Pergeseran logika berguna terutama untuk mengisolasi bidang dalam sebuah *word*. Os yang digeser menjadi *word* menggeser informasi yang tidak diinginkan yang digeser dari ujung lainnya.



Sumber: (Stallings, 2016)
Gambar 8.3. Operasi *Shift* dan *Rotate*

Sebagai contoh, misalkan sebuah data akan dikirimkan ke perangkat I/O satu karakter sekaligus. Jika setiap *word* memori memiliki panjang 16 bit dan berisi dua karakter, maka karakter harus dipecah sebelum dapat dikirim. Yang dilakukan untuk mengirim dua karakter dalam satu *word* adalah:

- 1) Masukkan *word* ke dalam register.
- 2) Isi register digeser ke kanan delapan kali. Ini menggeser karakter yang tersisa ke bagian kanan register.
- 3) Lakukan I/O. Modul I/O membaca urutan 8-bit dari bus data.

Langkah-langkah sebelumnya menghasilkan pengiriman karakter kiri. Untuk mengirim karakter sebelah kanan langkahnya adalah:

- 1) Muat *word* lagi ke dalam register.
- 2) AND dengan 0000000011111111. Ini menutupi karakter di sebelah kiri.
- 3) Lakukan I/O.

Shift, adalah operasi aritmatika yang memperlakukan data sebagai bilangan signed integer dan tidak menggeser bit tanda. Pada pergeseran aritmatika kanan, bit tanda direplikasi ke posisi bit di sebelah kanannya. Pada pergeseran aritmatika kiri, pergeseran logika kiri dilakukan pada semua bit tetapi bit tanda, yang dipertahankan. Operasi ini dapat mempercepat operasi aritmatika tertentu. Dengan angka dalam notasi Komplemen-2, *shift* aritmatika kanan sesuai dengan pembagian dengan 2, dengan pemotongan untuk angka ganjil. Baik *shift* kiri aritmatika dan *shift* kiri logika sesuai dengan perkalian dengan 2 saat tidak ada *overflow*. Jika *overflow* terjadi, operasi aritmatika dan *shift* kiri logika menghasilkan hasil yang berbeda, tetapi *shift* kiri aritmatika tetap menggunakan tanda angka tersebut.

Tabel 8.8. Contoh dari Operasi *Shift* dan *Rotate*

Input	Operasi	Hasil
1010 0110	<i>Logical right shift (3 bits)</i>	0001 0100
1010 0110	<i>Logical left shift (3 bits)</i>	0011 0000
1010 0110	<i>Arithmetic right shift (3 bits)</i>	1111 0100
1010 0110	<i>Arithmetic left shift (3 bits)</i>	1011 0000
1010 0110	<i>Right rotate (3bit)</i>	1101 0100
1010 0110	<i>Left rotate (3bit)</i>	0011 0101

Sumber: (Penulis, 2020)

Rotate (*shift cyclic*), adalah operasi mempertahankan semua bit yang dioperasikan. Salah satu penggunaan *rotate* adalah untuk membawa setiap bit secara berturut-turut ke dalam bit paling kiri, di mana ia dapat diidentifikasi dengan menguji tanda data (diperlakukan sebagai angka). Seperti operasi aritmatika, operasi logika melibatkan aktivitas ALU dan mungkin melibatkan operasi transfer data. Tabel 8.8 memberikan contoh semua operasi *shift* dan *rotasi* yang dibahas.

D. Kendali sistem

Instruksi kontrol sistem adalah instruksi yang hanya dapat dieksekusi ketika prosesor berada dalam status istimewa tertentu atau menjalankan program di area memori khusus istimewa. Biasanya, instruksi ini dicadangkan untuk penggunaan sistem operasi.

Beberapa contoh operasi kontrol sistem adalah sebagai berikut. Instruksi kontrol sistem dapat membaca atau mengubah register kontrol. Contoh lainnya adalah instruksi untuk membaca atau memodifikasi kunci perlindungan penyimpanan, seperti yang digunakan dalam sistem memori EAS/390. Contoh lain adalah akses ke blok kontrol proses dalam sistem *multiprogramming*.

E. Transfer kendali

Untuk semua jenis operasi yang dibahas sejauh ini, instruksi selanjutnya yang harus dilakukan adalah yang segera mengikuti, dalam memori, instruksi saat ini. Namun, sebagian kecil dari instruksi dalam setiap program memiliki fungsinya mengubah urutan pelaksanaan instruksi. Untuk instruksi ini, operasi yang dilakukan oleh prosesor adalah memperbarui register *Program Counter* untuk memuat alamat beberapa instruksi dalam memori.

Ada sejumlah alasan mengapa operasi *transfer-of-control* diperlukan. Di antara yang paling penting adalah sebagai berikut:

- 1) Dalam penggunaan praktis komputer, penting untuk dapat menjalankan setiap instruksi lebih dari sekali dan mungkin ribuan kali. Mungkin memerlukan ribuan atau mungkin jutaan instruksi untuk mengimplementasikan aplikasi. Ini tidak akan terpikirkan jika setiap instruksi harus ditulis secara terpisah. Jika tabel atau daftar item akan diproses, *loop* program diperlukan. Satu urutan instruksi dieksekusi berulang kali untuk memproses semua data.
- 2) Hampir semua program melibatkan pengambilan keputusan. Bila menginginkan komputer melakukan satu hal jika satu syarat berlaku, dan satu lagi jika syarat lain berlaku. Misalnya, urutan instruksi menghitung akar kuadrat dari angka. Di awal urutan, tanda nomor diuji. Jika jumlahnya negatif, perhitungan tidak dilakukan, tetapi kondisi kesalahan dilaporkan.
- 3) Untuk menyusun dengan benar program komputer berukuran besar atau bahkan sedang adalah tugas yang sangat sulit. Ini membantu jika ada mekanisme untuk memecah tugas menjadi potongan-potongan kecil yang dapat dikerjakan satu per satu.

Operasi transfer-of-control paling umum yang ditemukan dalam set instruksi: branch, skip, dan procedure call.

Instruksi Peracabangan.

Instruksi branch, juga disebut instruksi jump, salah satu *operand* nya menyimpan alamat instruksi berikutnya yang akan dieksekusi. Paling sering, instruksi branch adalah instruksi cabang bersyarat. Yaitu, cabang dibuat (perbarui register PC ke alamat yang ditentukan dalam *operand*) hanya jika kondisi tertentu terpenuhi. Jika tidak, instruksi selanjutnya secara berurutan dieksekusi (increment register PC seperti normal). Sebuah instruksi cabang di mana cabang selalu diambil adalah cabang tanpa syarat.

Ada dua cara umum untuk menghasilkan kondisi yang akan diuji dalam instruksi cabang bersyarat. Pertama, sebagian besar mesin menyediakan kode kondisi 1-bit atau beberapa-bit yang ditetapkan sebagai hasil dari beberapa operasi. Kode ini dapat dianggap sebagai register singkat yang dapat dilihat pengguna. Sebagai contoh, operasi aritmatika (ADD, SUBTRACT, dan sebagainya) dapat menetapkan kode kondisi 2-bit dengan salah satu dari berikut ini

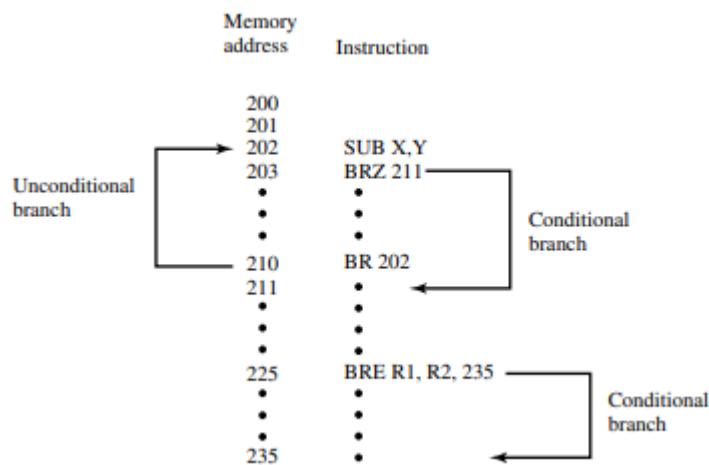
empat nilai: 0, positif, negatif, meluap. Pada mesin seperti itu, mungkin ada empat instruksi cabang bersyarat yang berbeda:

- BRP X Cabang ke lokasi X jika hasilnya positif.
- BRN X Cabang ke lokasi X jika hasilnya negatif.
- BRZ X Cabang ke lokasi X jika hasilnya nol.
- BRO X Cabang ke lokasi X jika terjadi *overflow*.

Dalam semua kasus ini, hasil yang dimaksud adalah hasil dari operasi terbaru yang menetapkan kode kondisi. Pendekatan lain yang dapat digunakan dengan format instruksi tiga alamat adalah dengan melakukan perbandingan dan menentukan cabang dalam instruksi yang sama. Sebagai contoh:

BRE R1, R2, X Branch to X if contents of R1 contents of R2

Gambar 8.4 menunjukkan contoh operasi ini. Perhatikan bahwa cabang dapat maju (instruksi dengan alamat yang lebih tinggi) atau mundur (alamat lebih rendah).



Sumber: (Stallings, 2016)
Gambar 8.4. Instruksi Percabangan

Contoh menunjukkan bagaimana cabang tanpa syarat dan bersyarat dapat digunakan untuk membuat *loop* berulang instruksi. Instruksi di lokasi 202 hingga 210 akan dieksekusi berulang kali hingga hasil pengurangan Y dari X adalah 0.

Skip Instructions.

Bentuk lain dari instruksi *transfer-of-control* adalah instruksi *skip*. Biasanya, lompatan menyiratkan bahwa satu instruksi dilewati; dengan demikian, alamat tersirat sama dengan alamat instruksi berikutnya ditambah satu panjang instruksi.

Karena instruksi *skip* tidak memerlukan bidang alamat tujuan, maka bebas untuk melakukan hal-hal lain. Contoh tipikal adalah instruksi *increment-and-skip-if-zero* (ISZ). Pertimbangkan potongan program berikut:

```
301
:
:
309 ISZ R1
310 BR 301
311
```

Dalam kasus ini, dua instruksi *transfer-of-control* digunakan untuk mengimplementasikan *loop* berulang. R1 diatur dengan negatif dari jumlah iterasi yang akan dilakukan. Pada akhir *loop*, R1 bertambah. Jika bukan 0, program bercabang kembali ke awal *loop*. Jika tidak, cabang dilewati, dan program melanjutkan dengan instruksi berikutnya setelah akhir dari *loop*.

Procedure Call Instructions.

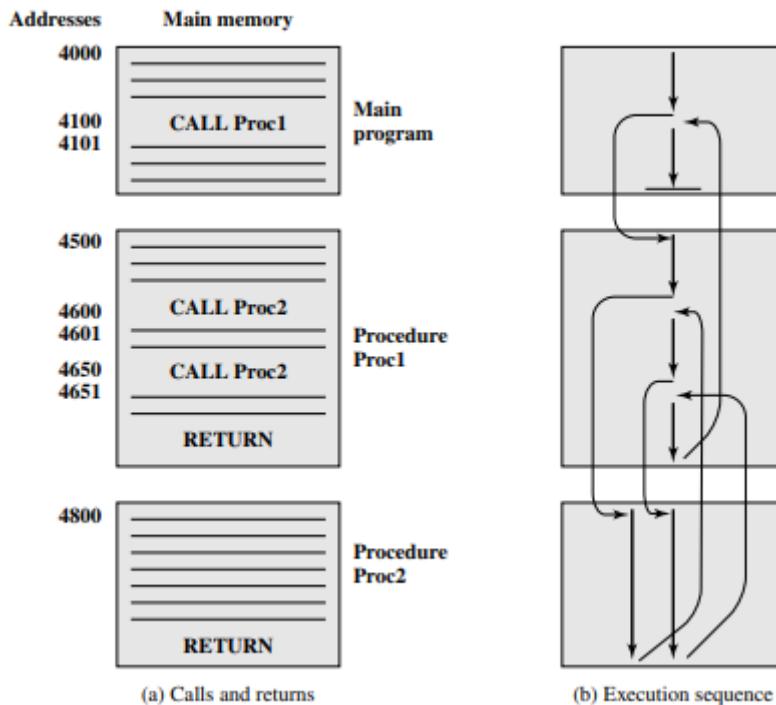
Mungkin inovasi yang paling penting dalam pengembangan bahasa pemrograman adalah prosedurnya. Suatu prosedur adalah program komputer mandiri yang dimasukkan ke dalam program yang lebih besar. Kapan saja dalam program ini prosedur dapat dipanggil. Prosesor diinstruksikan untuk pergi dan menjalankan seluruh prosedur dan kemudian kembali ke titik di mana panggilan berlangsung.

Dua alasan utama untuk penggunaan prosedur adalah ekonomi dan modularitas. Suatu prosedur memungkinkan potongan kode yang sama digunakan berulang kali. Ini penting untuk penghematan dalam upaya pemrograman dan untuk membuat penggunaan ruang penyimpanan yang paling efisien dalam sistem (program harus disimpan). Prosedur juga memungkinkan tugas pemrograman besar untuk dibagi lagi menjadi unit yang lebih kecil. Penggunaan modularitas ini sangat memudahkan tugas pemrograman.

Mekanisme prosedur melibatkan dua instruksi dasar: instruksi panggilan yang bercabang dari lokasi sekarang ke prosedur, dan instruksi pengembalian yang kembali dari prosedur ke tempat dari mana ia dipanggil. Keduanya adalah bentuk instruksi percabangan.

Gambar 8.5a menggambarkan penggunaan prosedur untuk membangun program. Dalam contoh ini, ada program utama mulai dari lokasi 4000. Program ini mencakup panggilan ke prosedur PROC1, mulai dari lokasi 4500. Ketika instruksi panggilan ini ditemui, prosesor menunda pelaksanaan program utama dan memulai pelaksanaan PROC1 dengan mengambil instruksi berikutnya dari lokasi 4500. Dalam PROC1, ada dua panggilan ke PROC2 di lokasi 4800. Dalam setiap kasus, eksekusi PROC1 ditunda dan PROC2 dieksekusi. Pernyataan RETURN menyebabkan prosesor untuk kembali ke program panggilan dan melanjutkan eksekusi sesuai instruksi setelah instruksi CALL yang sesuai. Perilaku ini diilustrasikan pada Gambar 8.5b. Tiga poin perlu diperhatikan:

- 1) Suatu prosedur dapat dipanggil dari beberapa lokasi.
- 2) Panggilan prosedur dapat muncul dalam suatu prosedur. Ini memungkinkan prosedur bersarang ke kedalaman yang sewenang-wenang.
- 3) Setiap panggilan prosedur dicocokkan dengan pengembalian dalam program yang dipanggil



Sumber: (Stallings, 2016)
Gambar 8.5. Prosedur Nested

Karena prosedur dapat dipanggil dari berbagai titik, prosesor harus bagaimana menyimpan alamat pengirim sehingga pengembalian dapat dilakukan dengan tepat. Ada tiga tempat umum untuk menyimpan alamat pengirim:

- Register
- Mulai dari prosedur yang disebut
- Bagian atas stack

Pertimbangkan instruksi bahasa mesin $\text{CALL } X$, yang merupakan singkatan dari prosedur panggilan di lokasi X . Jika pendekatan register digunakan, $\text{CALL } X$ menyebabkan tindakan berikut:

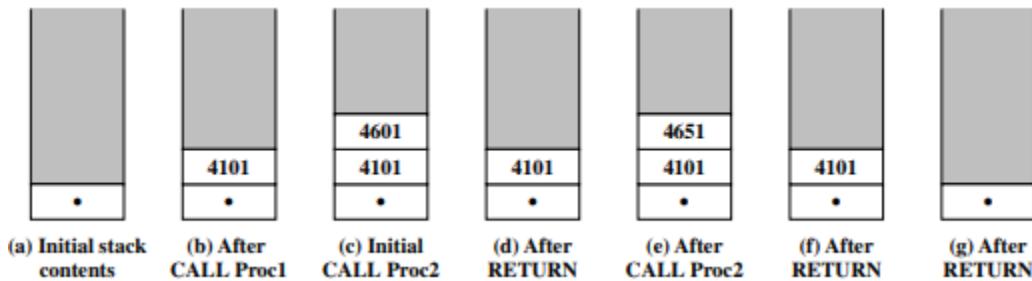
$$\begin{aligned} \text{RN} &\leftarrow \text{PC} + \text{C} \\ \text{PC} &\leftarrow X \end{aligned}$$

di mana RN adalah register yang selalu digunakan untuk tujuan ini, PC adalah penghitung program, dan merupakan panjang instruksi. Prosedur yang dipanggil sekarang dapat menyimpan konten RN untuk digunakan untuk pengembalian nanti. Kemungkinan kedua adalah menyimpan alamat pengirim di awal prosedur. Dalam hal ini, $\text{CALL } X$ menyebabkan:

$$\begin{aligned} X &\leftarrow \text{PC} + \text{C} \\ \text{PC} &\leftarrow X + 1 \end{aligned}$$

Ini sangat berguna. Alamat pengirim telah disimpan dengan aman.

Kedua pendekatan sebelumnya bekerja dan telah digunakan. Satu-satunya batasan dari pendekatan ini adalah mereka mempersulit penggunaan prosedur *reentrant*. Prosedur *reentrant* adalah prosedur yang memungkinkan beberapa panggilan terbuka secara bersamaan. Prosedur rekursif (prosedur yang menyebut dirinya sendiri) adalah contoh penggunaan fitur ini.

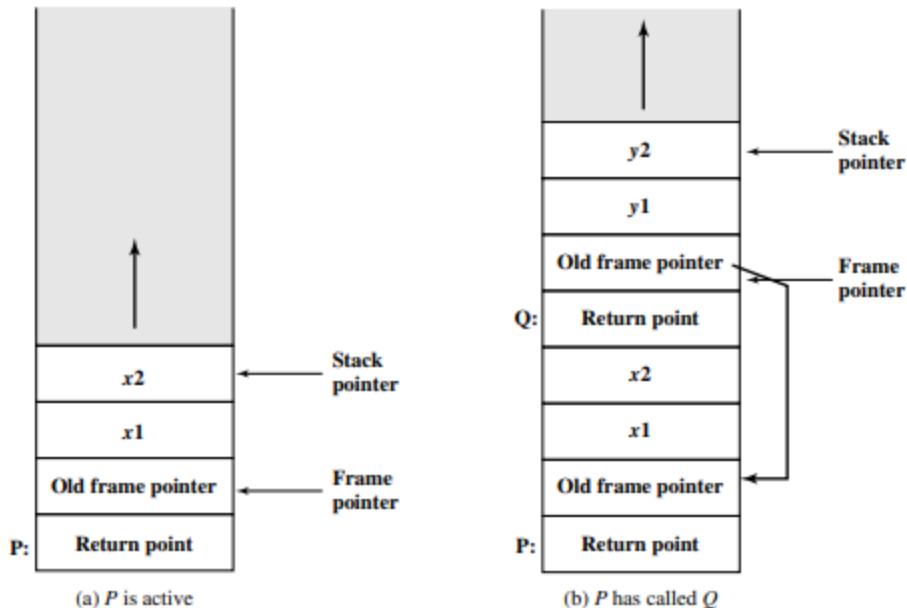


Sumber: (Stallings, 2016)

Gambar 8.6. Penggunaan Stack pada implementasi Nestet Subroutine dari Gambar 8.5

Pendekatan yang lebih umum adalah dengan menggunakan *stack*. Ketika prosesor mengeksekusi panggilan, itu menempatkan alamat pengirim di *stack*. Ketika mengeksekusi pengembalian, ia menggunakan alamat pada *stack*. Gambar 8.6 mengilustrasikan penggunaan *stack*. Selain memberikan alamat kembali, sering juga diperlukan untuk melewatkkan parameter dengan panggilan prosedur. Ini dapat diteruskan dalam register. Kemungkinan lain adalah menyimpan parameter dalam memori tepat setelah instruksi CALL. Dalam hal ini, pengembalian harus ke lokasi mengikuti parameter. Lagi pula, kedua pendekatan ini memiliki kelemahan. Jika register digunakan, program yang dipanggil dan program panggilan harus ditulis untuk memastikan bahwa register digunakan dengan benar. Penyimpanan parameter dalam memori membuatnya sulit untuk bertukar sejumlah variabel parameter. Kedua pendekatan mencegah penggunaan prosedur *reentrant*.

Pendekatan yang lebih fleksibel untuk melewati parameter adalah *stack*. Ketika prosesor mengeksekusi panggilan, itu tidak hanya menumpuk *return address*, itu menumpuk parameter untuk diteruskan ke prosedur yang dipanggil. Prosedur yang di-CALL dapat mengakses parameter dari *stack*. Setelah RETURN, *return parameter* juga dapat ditempatkan di *stack*. Seluruh rangkaian parameter, termasuk *return address*, yang disimpan untuk permohonan prosedur disebut sebagai *stack frame*.



Sumber: (Stallings, 2016)

Gambar 8.7. Penambahan isi Stack dengan prosedur sederhana P dan Q

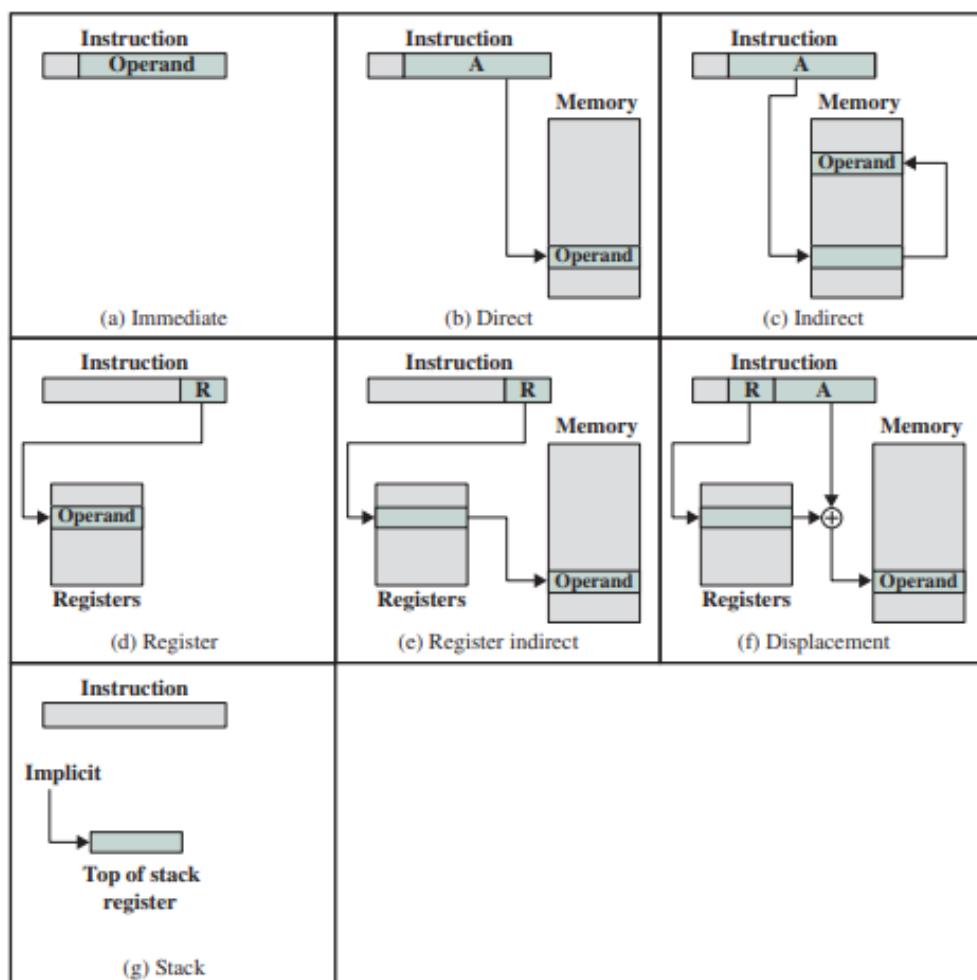
Contoh disediakan pada Gambar 8.7. Contohnya merujuk pada prosedur P di mana variabel lokal x1 dan x2 dideklarasikan, dan prosedur Q, yang dapat dipanggil P dan di mana variabel lokal y1 dan y2 dideklarasikan. Dalam gambar ini, titik balik untuk setiap prosedur adalah item pertama yang disimpan

dalam bingkai *stack* yang sesuai. Selanjutnya disimpan pointer ke awal *frame* sebelumnya. Ini diperlukan jika jumlah atau panjang parameter yang akan ditumpuk adalah variabel.

8.4. Mode Pengalamatan

Pada umumnya dalam format instruksi, bidang alamat berukuran relatif kecil. Dengan ukuran bidang alamat yang relatif tersebut, prosesor hanya dapat menjangkau sedikit lokasi memori. User selalu ingin dapat menjangkau seluruh lokasi di memori utama atau memori virtual, untuk mencapai tujuan ini, berbagai mode pengalamatan memori telah digunakan. Mode pengalamatan yang paling umum adalah: Immediate, Direct, Indirect, Register, Register indirect, Displacement, dan Stack. Gambar 8.8 mengilustrasikan mode-mode ini, dengan notasi yang digunakan adalah:

- 1) A = isi bidang alamat dalam instruksi
- 2) R = isi bidang alamat dalam instruksi yang merujuk pada register
- 3) EA (*Effective Address*) = alamat yang sebenarnya lokasi memori yang berisi *operand* yang direferensikan
- 4) (X) = isi lokasi memori X atau register X



Sumber: (Stallings, 2016)
Gambar 8.8. Mode Pengalamatan

Tabel 8.9 menunjukkan perhitungan alamat yang dilakukan untuk setiap mode pengalamatan. Hampir semua arsitektur komputer menyediakan lebih dari satu mode pengalamatan ini. Muncul pertanyaan tentang bagaimana prosesor dapat menentukan mode alamat mana yang digunakan dalam instruksi tertentu. Beberapa pendekatan telah diambil. Seringkali, opcode yang berbeda akan menggunakan mode pengalamatan yang berbeda. Juga, satu atau lebih bit dalam format instruksi dapat digunakan sebagai bidang mode. Nilai bidang mode menentukan mode pengalamatan mana yang akan digunakan.

Dalam sistem tanpa memori virtual, alamat efektif (*effective address*) dapat berupa alamat memori utama atau register. Dalam sistem memori virtual, alamat efektif adalah alamat virtual atau register. Pemetaan aktual ke alamat fisik adalah fungsi dari *Memory Management Unit* (MMU) dan tidak terlihat oleh programmer.

Tabel 8.9. Mode Pengalamatan Dasar

Mode	Algoritma	Kelebihan	Kekurangan
Immediate	$Operand=A$	Tidak ada referensi memori	Besaran <i>operand</i> terbatas
Direct	$EA=A$	Simple	Ruang alamat terbatas
Indirect	$EA=(A)$	Space alamat besar	Multiple memory references
Register	$EA=R$	Tidak ada referensi memori	Ruang alamat terbatas
Register indirect	$EA=(R)$	Space alamat besar	Extra memory references
Displacement	$EA=A+(R)$	Fleksibel	Komples
Stack	$EA=top\ of\ stack$	Tidak ada referensi memori	Aplikasi terbatas

Ket: EA = Efektive Address; A= Address

Sumber: (Stallings, 2016)

A. Immediate Addressing

Bentuk paling sederhana dari pengalamatan adalah pengalamatan segera (*immediate addressing*), di mana nilai *operand* terdapat dalam instruksi

$$\text{Operand} = A$$

Mode ini dapat digunakan untuk mendefinisikan dan menggunakan konstanta atau mengatur nilai awal variabel. Biasanya, nomor tersebut akan disimpan dalam dua bentuk komplemen; bit paling kiri dari bidang *operand* digunakan sebagai bit tanda. Ketika *operand* dimuat ke dalam register data, bit tanda diperpanjang ke kiri ke ukuran *word* data penuh. Dalam beberapa kasus, nilai biner langsung ditafsirkan sebagai bilangan bulat non-negatif yang tidak ditandatangani.

Kelebihan dari *Immediate Addressing* adalah bahwa tidak ada referensi memori selain pengambilan instruksi yang diperlukan untuk mendapatkan *operand*, sehingga menghemat satu memori atau siklus *cache* dalam siklus instruksi. Kekurangannya adalah besaran bilangan yang terbatas pada ukuran field alamat, yang pada kebanyakan set instruksi, lebih kecil dibandingkan dengan panjang *word*.

B. Direct Addressing

Bentuk pengalamatan yang sangat sederhana adalah pengalamatan langsung (*direct addressing*), di mana bidang alamat berisi alamat efektif (*Effective Addressing EA*) dari *operand*:

$$EA = A$$

Teknik ini umum di komputer generasi sebelumnya tetapi tidak umum pada arsitektur kontemporer. Ini hanya membutuhkan satu referensi memori dan tidak ada perhitungan khusus. Keterbatasan yang jelas adalah bahwa ia hanya menyediakan ruang alamat yang terbatas.

C. Indirect Addressing

Dengan pengalaman langsung, panjang bidang alamat biasanya kurang dari panjang *word*, sehingga membatasi rentang alamat. Salah satu solusinya adalah membuat bidang alamat merujuk ke alamat *word* dalam memori, yang pada gilirannya berisi alamat lengkap *operand*. Ini dikenal sebagai pengalaman tidak langsung (*indirect addressing*):

$$EA = (A)$$

Seperti yang didefinisikan sebelumnya, tanda kurung harus ditafsirkan sebagai makna isi. Keuntungan nyata dari pendekatan ini adalah bahwa untuk panjang *word* N , ruang alamat 2^N sekarang tersedia. Kerugiannya adalah bahwa eksekusi instruksi memerlukan dua referensi memori untuk mengambil *operand*: satu untuk mendapatkan alamatnya dan yang kedua untuk mendapatkan nilainya.

Meskipun jumlah *word* yang dapat dialamatkan sekarang sama dengan 2^N , jumlah alamat efektif yang berbeda yang dapat dirujuk pada satu waktu terbatas pada 2^K , di mana K adalah panjang bidang alamat. Biasanya, ini bukan pembatasan yang memberatkan, dan ini bisa menjadi aset. Dalam lingkungan memori virtual, semua lokasi alamat efektif dapat dibatasi pada halaman 0 dari proses apa pun. Karena bidang alamat instruksi kecil, maka secara alami akan menghasilkan alamat langsung bermotor rendah, yang akan muncul di halaman 0. (Satu-satunya batasan adalah bahwa ukuran halaman harus lebih besar dari atau sama dengan 2^K .) Ketika suatu proses adalah aktif, akan ada referensi berulang ke halaman 0, menyebabkannya tetap dalam memori nyata. Dengan demikian, referensi memori tidak langsung akan melibatkan, paling banyak, satu kesalahan halaman daripada dua.

D. Register Addressing

Pengalaman register (*register addressing*) mirip dengan pengalaman langsung. Satu-satunya perbedaan adalah bahwa bidang alamat merujuk pada register daripada alamat memori utama:

$$EA = R$$

Untuk memperjelas, jika isi bidang alamat register dalam instruksi adalah 5, maka register R5 adalah alamat yang dimaksud, dan nilai *operand* terkandung dalam R5. Biasanya, bidang alamat yang memiliki referensi register akan dari 3 hingga 5 bit, sehingga total dari 8 hingga 32 register tujuan umum dapat dirujuk.

Kelebihan dari pengalaman register adalah bahwa (1) hanya bidang alamat kecil yang diperlukan dalam instruksi, dan (2) tidak diperlukan referensi memori yang menghabiskan waktu. Waktu akses memori untuk register internal ke prosesor jauh lebih sedikit dari pada untuk alamat memori utama. Kerugian dari pengalaman register adalah ruang alamat sangat terbatas.

Jika pengalaman register banyak digunakan dalam set instruksi, ini menyiratkan bahwa register prosesor akan banyak digunakan. Karena jumlah register yang sangat terbatas (dibandingkan dengan lokasi memori utama), penggunaannya dengan cara ini masuk akal hanya jika digunakan secara efisien. Jika setiap *operand* dimasukkan ke register dari memori utama, dioperasikan sekali, dan kemudian dikembalikan ke memori utama, maka langkah perantara yang sia-sia telah ditambahkan. Sebaliknya, jika *operand* dalam register tetap digunakan untuk beberapa operasi, maka penghematan nyata tercapai. Contohnya adalah hasil antara dalam perhitungan. Secara khusus, anggaplah bahwa algoritma untuk perkalian pelengkap dua pasangan harus diimplementasikan dalam perangkat lunak. Lokasi berlabel A dalam bagan alur (Gambar 10.12) dirujuk berkali-kali dan harus diimplementasikan dalam register daripada lokasi memori utama. Terserah programmer atau kompiler untuk memutuskan nilai mana yang harus tetap dalam register dan mana yang harus disimpan dalam memori utama. Sebagian besar prosesor modern menggunakan beberapa register tujuan umum, menempatkan beban untuk eksekusi yang efisien pada programmer bahasa assembly (misalnya: penulis kompiler).

E. Register Indirect Addressing

Sama seperti pengalaman register analog dengan pengalaman langsung, register pengalaman tidak langsung analog dengan pengalaman tidak langsung. Dalam kedua kasus, satu-satunya perbedaan adalah apakah bidang alamat merujuk ke lokasi memori atau register. Jadi, untuk register alamat tidak langsung,

$$EA = (R)$$

Kelebihan dan keterbatasan pengalaman register tidak langsung pada dasarnya sama dengan pengalaman tidak langsung. Dalam kedua kasus, batasan ruang alamat (kisaran terbatas alamat) bidang alamat diatasi dengan meminta bidang itu merujuk ke lokasi panjang *word* yang berisi alamat. Selain itu, register pengalaman tidak langsung menggunakan referensi memori yang lebih sedikit daripada pengalaman tidak langsung.

F. Displacement Addressing

Mode pengalaman yang sangat kuat menggabungkan kemampuan *direct addressing* dan *indirect addressing*. Ini dikenal dengan berbagai nama tergantung pada konteks penggunaannya, tetapi mekanisme dasarnya sama.

$$EA = A + (R)$$

Displacement Addressing mengharuskan instruksi memiliki dua bidang alamat, setidaknya satu di antaranya eksplisit. Nilai yang terkandung dalam satu bidang alamat (nilai = A) digunakan secara langsung. Bidang alamat lain, atau referensi tersirat berdasarkan opcode, merujuk ke register yang isinya ditambahkan ke A untuk menghasilkan alamat yang efektif.

tiga penggunaan yang paling umum dari *Displacement Addressing* adalah:

- 1) Pengalaman relatif (**Relative Addressing**)
- 2) Pengalaman register dasar (**Base-Register Addressing**)
- 3) Pengindeksan (**Indexing**)

Relative Addressing

Untuk pengalaman relatif, juga disebut pengalaman relatif PC, register yang dirujuk secara implisit adalah *Program Counter* (PC). Yaitu, alamat instruksi berikutnya ditambahkan ke bidang alamat untuk menghasilkan EA. Biasanya, bidang alamat diperlakukan sebagai nomor pelengkap dua untuk operasi ini. Dengan demikian, alamat efektif adalah perpindahan relatif terhadap alamat instruksi.

Pengalaman relatif mengeksplorasi konsep lokalitas. Jika sebagian besar referensi memori relatif dekat dengan instruksi yang dieksekusi, maka penggunaan pengalaman relatif menyimpan bit alamat dalam instruksi.

Base-Register Addressing

Untuk pengalaman register-dasar, interpretasinya adalah sebagai berikut: Register yang direferensikan berisi alamat memori utama, dan bidang alamat berisi perpindahan (biasanya representasi integer yang tidak ditandatangani) dari alamat itu. Referensi register mungkin eksplisit atau implisit.

Pengalaman register-basis juga mengeksplorasi lokalitas referensi memori. Ini adalah cara yang mudah untuk mengimplementasikan segmentasi. Dalam beberapa implementasi, register segment tunggal digunakan dan digunakan secara implisit. Di tempat lain, programmer dapat memilih register untuk menyimpan alamat dasar suatu segmen, dan instruksi harus merujuknya secara eksplisit. Dalam yang terakhir ini

kasus, jika panjang bidang alamat adalah K dan jumlah register yang mungkin adalah N, maka satu instruksi dapat merujuk salah satu dari area N dari word 2K.

Indexing

Untuk pengindeksan, interpretasinya biasanya sebagai berikut: Bidang alamat referensi alamat memori utama, dan register yang direferensikan berisi perpindahan positif dari alamat itu. Perhatikan bahwa penggunaan ini hanya kebalikan dari interpretasi untuk pengalaman register-dasar. Tentu saja, ini lebih dari sekadar masalah interpretasi pengguna. Karena bidang alamat dianggap sebagai alamat memori dalam pengindeksan, umumnya berisi lebih banyak bit daripada bidang alamat dalam instruksi register-base yang sebanding. Juga ada beberapa penyempurnaan untuk pengindeksan yang tidak akan berguna dalam konteks basis-register.

Namun demikian, metode penghitungan EA adalah sama untuk pengalaman register basis dan pengindeksan, dan dalam kedua kasus referensi register kadang-kadang eksplisit dan kadang-kadang tersirat (untuk jenis prosesor yang berbeda).

Penggunaan pengindeksan yang penting adalah untuk menyediakan mekanisme yang efisien untuk melakukan operasi berulang. Pertimbangkan, misalkan, nomor register yang disimpan mulai dari lokasi A. Misalkan ingin menambahkan 1 untuk setiap elemen dalam register. Diperlukan untuk mengambil setiap nilai, menambahkan 1, dan menyimpannya kembali. Urutan alamat efektif yang dibutuhkan adalah A, A + 1, A + 2, . . . , hingga lokasi terakhir dalam register.

Dengan pengindeksan, ini mudah dilakukan. Nilai A disimpan di bidang alamat instruksi, dan register yang dipilih, disebut register indeks, diinisialisasi ke 0. Setelah setiap operasi, register indeks bertambah 1.

Karena register indeks biasanya digunakan untuk tugas-tugas berulang seperti itu, itu adalah khas bahwa ada kebutuhan untuk menambah atau mengurangi register indeks setelah setiap referensi untuk itu. Karena ini adalah operasi yang umum, beberapa sistem akan secara otomatis melakukan ini sebagai bagian dari siklus instruksi yang sama. Ini dikenal sebagai autoindexing. Jika register tertentu dikhususkan untuk pengindeksan, maka autoindexing dapat dipanggil secara implisit dan otomatis. Jika register tujuan umum digunakan, operasi autoindex mungkin perlu ditandai sedikit dalam instruksi. Autoindexing menggunakan increment dapat digambarkan sebagai berikut.

$$\begin{aligned} EA &= A + (R) \\ (R) &\leftarrow (R) + 1 \end{aligned}$$

Di beberapa mesin, pengalaman dan pengindeksan tidak langsung disediakan, dan dimungkinkan untuk menggunakan keduanya dalam instruksi yang sama. Ada dua kemungkinan: pengindeksan dilakukan sebelum atau sesudah tipuan.

Jika pengindeksan dilakukan setelah tipuan, itu disebut postindexing:

$$EA = (A) + (R)$$

Pertama, isi bidang alamat digunakan untuk mengakses lokasi memori yang berisi alamat langsung. Alamat ini kemudian diindeks oleh nilai register. Teknik ini berguna untuk mengakses salah satu dari sejumlah blok data dalam format tetap. Misalkan: sistem operasi perlu menggunakan blok kontrol proses untuk setiap proses. Operasi yang dilakukan adalah sama terlepas dari blok mana yang sedang dimanipulasi. Dengan demikian, alamat dalam instruksi yang mereferensikan blok dapat menunjuk ke lokasi (nilai = A) yang berisi pointer variabel ke awal blok kontrol proses. Register indeks berisi perpindahan di dalam blok.

Dengan preindexing, pengindeksan dilakukan sebelum tipuan:

$$EA = (A + (R))$$

Alamat dihitung dengan pengindeksan sederhana. Namun dalam kasus ini, alamat yang dihitung tidak berisi *operand*, tetapi alamat *operand*. Contoh penggunaan teknik ini adalah untuk membangun tabel cabang *multiway*. Pada titik tertentu dalam suatu program, mungkin ada cabang ke salah satu dari sejumlah lokasi tergantung pada kondisi. Daftar alamat dapat diatur mulai dari lokasi A. Dengan mengindeks ke dalam tabel ini, lokasi yang diperlukan dapat ditemukan. Biasanya, satu set instruksi tidak akan mencakup *preindexing* dan *postindexing*.

G. Stack Addressing

Mode pengalamatan terakhir yang dibahas adalah stack addressing. *Stack* adalah lokasi array linier. Kadang-kadang disebut sebagai register *pushdown* atau *first-in-last-out*. *Stack* adalah blok lokasi yang dicadangkan. Item ditambahkan ke bagian atas *stack* sehingga, pada waktu tertentu, blok sebagian terisi. Terkait dengan *stack* adalah pointer yang nilainya adalah alamat bagian atas *stack*. Sebagai alternatif, dua elemen teratas *stack* mungkin ada di register prosesor, dalam hal ini penunjuk *stack* merujuk elemen ketiga dari *stack*. Penunjuk *stack* dipertahankan dalam register. Dengan demikian, referensi untuk menumpuk lokasi dalam memori sebenarnya register alamat tidak langsung.

Mode pengalamatan *stack* adalah bentuk pengalamatan tersirat. Instruksi mesin tidak perlu menyertakan referensi memori tetapi beroperasi secara implisit di atas *stack*.

8.5. Format Instruksi.

Format instruksi mendefinisikan tata letak bit instruksi, dalam hal bidang konstituenya. Format instruksi harus menyertakan opcode dan, secara implisit atau eksplisit, nol atau lebih *operand*. Setiap *operand* eksplisit direferensikan menggunakan salah satu mode pengalamatan yang dijelaskan sebelumnya. Format instruksi secara implisit atau eksplisit harus menunjukkan mode pengalamatan untuk setiap *operand*. Untuk sebagian besar set instruksi, lebih dari satu format instruksi digunakan.

A. Panjang instruksi

Masalah desain paling mendasar yang harus dihadapi adalah panjang format instruksi. Keputusan ini memengaruhi, dan dipengaruhi oleh, ukuran memori, organisasi memori, struktur bus, kompleksitas prosesor, dan kecepatan prosesor. Keputusan ini menentukan kekayaan dan fleksibilitas mesin seperti yang terlihat oleh programmer bahasa assembly.

Pertukaran yang paling jelas di sini adalah antara keinginan untuk repertoar instruksi yang kuat dan kebutuhan untuk menghemat ruang. Pemrogram menginginkan lebih banyak opcode, lebih banyak *operand*, lebih banyak mode pengalamatan, dan jangkauan alamat yang lebih besar. Lebih banyak opcode dan *operand* membuat hidup lebih mudah bagi programmer, karena program yang lebih pendek dapat ditulis untuk menyelesaikan tugas yang diberikan. Demikian pula, lebih banyak mode pengalamatan memberikan fleksibilitas yang lebih besar kepada programmer dalam mengimplementasikan fungsi-fungsi tertentu, seperti manipulasi tabel dan percabangan multi-arah. Dan, tentu saja, dengan peningkatan ukuran memori utama dan meningkatnya penggunaan memori virtual, programmer ingin dapat mengatasi rentang memori yang lebih besar. Semua hal ini (opcode, *operand*, mode pengalamatan, kisaran alamat) membutuhkan bit dan mendorong ke arah panjang instruksi yang lebih panjang. Tetapi panjang instruksi yang lebih panjang mungkin sia-sia. Instruksi 64-bit menempati dua kali luas instruksi 32-bit tetapi mungkin kurang dari dua kali lebih berguna.

Pertimbangan terkait adalah kecepatan transfer memori. Tingkat ini tidak mengikuti peningkatan kecepatan prosesor. Dengan demikian, memori dapat menjadi hambatan jika prosesor dapat menjalankan instruksi lebih cepat daripada yang bisa mereka ambil. Salah satu solusi untuk masalah ini adalah dengan menggunakan memori *cache*; yang lain adalah menggunakan instruksi yang lebih

pendek. Dengan demikian, instruksi 16-bit dapat diambil dua kali lipat kecepatan instruksi 32-bit tetapi mungkin dapat dieksekusi kurang dari dua kali lebih cepat.

Fitur yang tampaknya biasa tetapi tetap penting adalah bahwa panjang instruksi harus kelipatan dari panjang karakter, yang biasanya 8 bit, dan dari panjang angka *fix point*. Dalam *word* tertentu, panjang memori adalah unit organisasi yang “alami”. Ukuran *fix point* biasanya menentukan ukuran angka *fix point* (biasanya keduanya sama). Ukuran *word* juga biasanya sama dengan, atau paling tidak terkait secara integral dengan, ukuran transfer memori. Karena bentuk umum dari data adalah data karakter, maka sebuah *word* idealnya digunakan untuk menyimpan bit dari karakter yang tidak terpisahkan. Jika tidak, ada bit yang terbuang di setiap *word* saat menyimpan beberapa karakter, atau karakter harus menganggungi batas *word*.

B. Alokasi Bit

Untuk panjang instruksi yang diberikan, jelas ada *trade-off* antara jumlah *opcode* dan kemampuan pengalamatan. Lebih banyak *opcode*, maka lebih banyak bit dalam bidang *opcode*. Untuk format instruksi dengan panjang tertentu, ini mengurangi jumlah bit yang tersedia untuk dialamatkan. Ada satu penyempurnaan yang menarik untuk *trade-off* ini, dan itu adalah penggunaan kode variabel - panjang. Dalam pendekatan ini, ada panjang *opcode* minimum tetapi, untuk beberapa *opcode*, operasi tambahan dapat ditentukan dengan menggunakan bit tambahan dalam instruksi. Untuk instruksi dengan panjang tetap, jumlah bit ini lebih sedikit untuk dialamatkan. Dengan demikian, fitur ini digunakan untuk instruksi yang membutuhkan *operand* lebih sedikit dan/atau pengalamatan yang kurang kuat.

Faktor-faktor yang saling terkait berikut masuk ke dalam menentukan penggunaan bit pengalamatan.

- 1) Jumlah mode pengalamatan: Kadang-kadang mode pengalamatan dapat ditunjukkan secara implisit. Misalnya, *opcode* tertentu mungkin selalu meminta pengindeksan. Dalam kasus lain, mode pengalamatan harus eksplisit, dan satu atau lebih bit mode akan diperlukan.
- 2) Jumlah *operand*: Semakin sedikit alamat yang bisa dibuat untuk program yang lebih lama dan lebih canggung. Format instruksi umum pada mesin saat ini termasuk dua *operand*. Setiap alamat *operand* dalam instruksi mungkin memerlukan indikator mode sendiri, atau penggunaan indikator mode dapat dibatasi hanya pada salah satu bidang alamat.
- 3) Registras versus memori: Mesin harus memiliki register sehingga data dapat dibawa ke prosesor untuk diproses. Dengan register yang terlihat oleh pengguna tunggal (biasanya disebut akumulator), satu alamat *operand* implisit dan tidak menggunakan bit instruksi. Namun, pemrograman register tunggal adalah canggung dan membutuhkan banyak instruksi. Bahkan dengan banyak register, hanya beberapa bit yang diperlukan untuk menentukan register. Semakin banyak register dapat digunakan untuk referensi *operand*, semakin sedikit bit yang dibutuhkan. Sejumlah penelitian menunjukkan bahwa total 8 hingga 32 register yang terlihat oleh pengguna diinginkan. Sebagian besar arsitektur kontemporer memiliki setidaknya 32 register.
- 4) Jumlah set register: Sebagian besar mesin kontemporer memiliki satu set register tujuan umum, dengan biasanya 32 register atau lebih dalam set. Register ini dapat digunakan untuk menyimpan data dan dapat digunakan untuk menyimpan alamat untuk pengalamatan perpindahan. Beberapa arsitektur, termasuk x86, memiliki koleksi dua atau lebih set khusus (seperti data dan perpindahan). Satu keuntungan dari pendekatan yang terakhir ini adalah, untuk sejumlah register yang tetap, pemisahan fungsional
- 5) Membutuhkan lebih sedikit bit untuk digunakan dalam instruksi. Misalnya, dengan dua set delapan register, hanya 3 bit yang diperlukan untuk mengidentifikasi register; *opcode* atau mode register akan menentukan set register mana yang dirujuk.
- 6) Rentang alamat: Untuk alamat yang merujuk memori, kisaran alamat yang dapat direferensikan terkait dengan jumlah bit alamat. Karena ini memberlakukan batasan yang parah, pengalamatan langsung jarang digunakan. Dengan pengalamatan perpindahan, rentang dibuka hingga panjang register alamat. Meski begitu, masih nyaman untuk memungkinkan

perpindahan yang agak besar dari alamat register, yang membutuhkan jumlah bit alamat yang relatif besar dalam instruksi.

- 7) *Granularity of addressing:* Untuk alamat yang merujuk memori daripada register, faktor lain adalah *granularity of addressing*. Dalam suatu sistem dengan 16 atau 32-bit *word*, suatu alamat dapat merujuk suatu *word* atau *byte* pada pilihan perancang. Pengalaman *byte* nyaman untuk manipulasi karakter tetapi membutuhkan, untuk memori ukuran tetap, lebih banyak bit alamat.

PDP-8.

Salah satu desain instruksi paling sederhana untuk komputer serba guna adalah untuk PDP-8. PDP-8 menggunakan instruksi 12-bit dan beroperasi dengan *words* 12-bit. Ada register tujuan umum tunggal, akumulator.

Terlepas dari keterbatasan desain ini, pengalamatannya cukup fleksibel. Setiap referensi memori terdiri dari 7 bit plus dua pengubah 1-bit. Memori dibagi menjadi halaman dengan panjang tetap masing-masing $2^7 = 128$ *word*. Penghitungan alamat didasarkan pada referensi ke halaman 0 atau halaman saat ini (halaman yang berisi instruksi ini) sebagaimana ditentukan oleh bit halaman. Bit modifier kedua menunjukkan apakah pengalaman langsung atau tidak langsung akan digunakan. Kedua mode ini dapat digunakan dalam kombinasi, sehingga alamat tidak langsung adalah alamat 12-bit yang terkandung dalam *word* halaman 0 atau halaman saat ini. Selain itu, 8 *word* khusus pada halaman 0 adalah "register." Autoindex Ketika referensi tidak langsung dibuat ke salah satu lokasi ini, preindexing terjadi.

Gambar 8.9 menunjukkan format instruksi PDP-8. Ada opcode 3-bit dan tiga jenis instruksi. Untuk opcodes 0 hingga 5, formatnya adalah instruksi referensi memori alamat tunggal termasuk bit halaman dan bit tidak langsung. Dengan demikian, hanya ada enam operasi dasar. Untuk memperbesar grup operasi, opcode 7 mendefinisikan referensi register atau microinstruction. Dalam format ini, bit yang tersisa digunakan untuk menyandikan operasi tambahan. Secara umum, setiap bit mendefinisikan operasi tertentu, dan bit ini dapat digabungkan dalam satu instruksi. Opcode 6 adalah operasi I/O; 6 bit digunakan untuk memilih satu dari 64 perangkat, dan 3 bit menentukan perintah I/O tertentu.

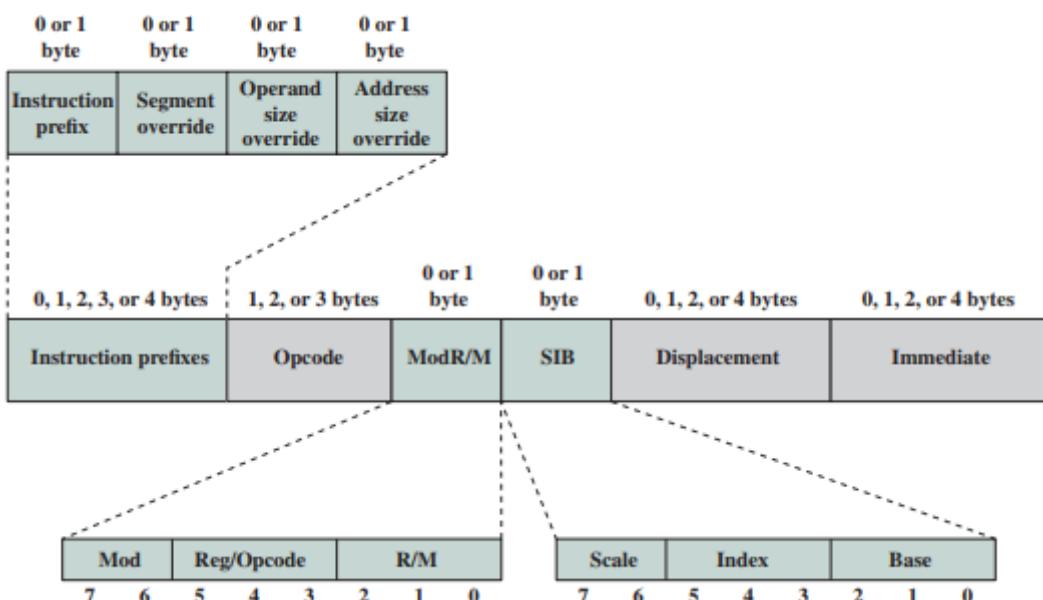
Format instruksi PDP-8 sangat efisien. Ini mendukung pengalaman tidak langsung, pengalaman perpindahan, dan pengindeksan. Dengan menggunakan ekstensi opcode, ia mendukung total sekitar 35 instruksi. Mengingat kendala dari panjang instruksi 12-bit, para desainer hampir tidak bisa berbuat lebih baik.

Memory reference instructions																	
Opcode		D/I	Z/C	Displacement													
0 2 3 4 5 11																	
Input/output instructions																	
1 1 0			Device							Opcode 11							
0 2 3 8 9 11																	
Register reference instructions																	
Group 1 microinstructions																	
1 1 1 0 CLA CLL CMA CML RAR RAL BSW IAC	0 1 2 3 4 5 6 7 8 9 10 11																
Group 2 microinstructions																	
1 1 1 0 CLA SMA SZA SNL RSS OSR HLT 0	0 1 2 3 4 5 6 7 8 9 10 11																
Group 3 microinstructions																	
1 1 1 0 CLA MQA 0 MQL 0 0 0 1	0 1 2 3 4 5 6 7 8 9 10 11																
D/I = Direct/Indirect address Z/C = Page 0 or Current page CLA = Clear Accumulator CLL = Clear Link CMA = CoMplement Accumulator CML = CoMplement Link RAR = Rotate Accumulator Right RAL = Rotate Accumulator Left BSW = Byte SWap	IAC = Increment ACcumulator SMA = Skip on Minus Accumulator SZA = Skip on Zero Accumulator SNL = Skip on Nonzero Link RSS = Reverse Skip Sense OSR = Or with Switch Register HLT = HaLT MQA = Multiplier Quotient into Accumulator MQL = Multiplier Quotient Load																

Sumber: (Stallings, 2016)
Gambar 8.9. Format Instruksi PDP-8

Format Instruksi x86

x86 dilengkapi dengan berbagai format instruksi. Dari elemen-elemen yang dijelaskan dalam ayat ini, hanya bidang opcode yang selalu ada. Gambar 8.10 menggambarkan format instruksi umum. Instruksi terdiri dari nol hingga empat awalan instruksi opsional, opcode 1 byte atau 2 byte, penentu alamat opsional (yang terdiri dari byte **ModR/M** dan byte **Base Indeks Base**) perpindahan opsional, dan opsional langsung bidang.



Sumber: (Stallings, 2016)
Gambar 8.10. Format Instruksi x86

Pertama-tama mari pertimbangkan byte awalan:

- 1) **Instruction prefixes:** Instruction prefixes, jika ada, terdiri dari awalan **LOCK** atau salah satu dari awalan berulang. Awalan **LOCK** digunakan untuk memastikan penggunaan memori bersama secara eksklusif di lingkungan multi-prosesor. Awalan berulang menentukan operasi berulang string, yang memungkinkan x86 untuk memproses string lebih cepat daripada dengan *loop* perangkat lunak biasa. Ada lima awalan yang berbeda: **REP**, **REPE**, **REPZ**, **REPNE**, dan **REPNZ**. Ketika awalan REP absolut hadir, operasi yang ditentukan dalam instruksi dieksekusi berulang kali pada elemen berturut-turut dari string; jumlah pengulangan ditentukan dalam register **CX**. Awalan REP bersyarat menyebabkan instruksi untuk mengulang sampai hitungan dalam **CX** menjadi nol atau sampai kondisi terpenuhi.
- 2) **Segment Override:** Menentukan secara eksplisit segmen mana yang harus digunakan oleh suatu instruksi, mengesampingkan pemilihan segmen-register standar yang dihasilkan oleh x86 untuk instruksi tersebut.
- 3) **Operand:** Suatu instruksi memiliki ukuran *operand* standar 16 atau 32 bit, dan awalan *operand* beralih antara *operand* 32 bit dan 16 bit.
- 4) **Ukuran alamat:** Prosesor dapat menangani memori menggunakan alamat 16 atau 32 bit. Ukuran alamat menentukan ukuran perpindahan dalam instruksi dan ukuran offset alamat yang dihasilkan selama perhitungan alamat yang efektif. Salah satu ukuran ini ditetapkan sebagai default, dan awalan ukuran alamat beralih antara generasi alamat 32-bit dan 16-bit.

Instruksi itu sendiri mencakup bidang-bidang berikut:

- 5) **Opcode:** Bidang opcode panjangnya 1, 2, atau 3 byte. Opcode juga dapat menyertakan bit yang menentukan apakah data berukuran byte atau penuh (16 atau 32 bit tergantung pada konteks), arah operasi data (ke atau dari memori), dan apakah bidang data langsung harus diperpanjang tanda.
- 6) **ModR/M:** Byte ini, dan selanjutnya, memberikan informasi pengalamatan. Byte **ModR/M** menentukan apakah *operand* ada dalam register atau dalam memori; jika ada di memori, maka bidang dalam byte menentukan mode pengalamatan yang akan digunakan. Byte **ModR/M** terdiri dari tiga bidang: Bidang **Mod** (2 bit) bergabung dengan bidang **R/M** untuk membentuk 32 nilai yang mungkin: 8 register dan 24 mode pengindeksan; bidang Reg/Opcode (3 bit) menentukan nomor register atau tiga bit informasi opcode lainnya; bidang **R/M** (3 bit) dapat menentukan register sebagai lokasi *operand*, atau dapat membentuk bagian dari pengkodean metode pengalamatan yang dikombinasikan dengan bidang **Mod**.
- 7) **SIB:** Pengkodean tertentu dari byte **ModR/M** menentukan penyertaan byte **SIB** untuk menentukan sepenuhnya mode pengalamatan. **SIB** byte terdiri dari tiga bidang: Bidang Skala (2 bit) menentukan faktor skala untuk pengindeksan berskala; bidang Indeks (3 bit) menentukan register indeks; bidang Base (3 bit) menentukan register dasar.
- 8) **Displacement:** Ketika specifier mode pengalamatan menunjukkan bahwa perpindahan digunakan, bidang pemindahan bilangan bulat bertanda tangan 8-, 16-, atau 32-bit ditambahkan.
- 9) **Immediate:** Memberikan nilai *operand* 8, 16, atau 32 bit.

9. Assembler Language

Bahasa assembly (*assembly language*) adalah bahasa pemrograman yang berjarak satu langkah dari bahasa mesin. Biasanya, setiap instruksi bahasa rakitan diterjemahkan ke dalam satu instruksi mesin oleh assembler. Bahasa rakitan bergantung pada perangkat keras, dengan bahasa rakitan berbeda untuk setiap jenis prosesor. Secara khusus, instruksi bahasa rakitan dapat membuat referensi ke register spesifik dalam prosesor, termasuk semua opcode prosesor, dan mencerminkan panjang bit dari berbagai register prosesor dan *operand* dari bahasa mesin. Oleh karena itu, seorang programmer bahasa assembly harus memahami arsitektur komputer.

Berikut adalah istilah yang perlu dipahami sebelum membahas lebih dalam mengenai *assembler language*:

- **Assembler:** Program yang menerjemahkan bahasa rakitan menjadi kode mesin.
- **Assembly Language:** Representasi simbolis dari bahasa mesin prosesor tertentu, ditambah dengan jenis pernyataan tambahan yang memfasilitasi penulisan program dan yang memberikan instruksi kepada assembler.
- **Compiler:** Program yang mengonversi program lain dari source language (atau bahasa pemrograman) ke bahasa mesin (kode objek). Beberapa compiler mengeluarkan bahasa rakitan yang kemudian dikonversi ke bahasa mesin oleh assembler terpisah. Sebuah kompiler dibedakan dari assembler oleh fakta bahwa setiap pernyataan *input* tidak, secara umum, sesuai dengan instruksi mesin tunggal atau urutan instruksi tetap. Kompiler dapat mendukung fitur-fitur seperti alokasi variabel otomatis, ekspresi aritmatika arbitrary, struktur kontrol seperti *loop FOR* dan *WHILE*, ruang lingkup variabel, operasi *input/output*, fungsi tingkat tinggi dan portabilitas source code.
- **Executable Code:** Kode mesin dihasilkan oleh prosesor bahasa kode sumber seperti assembler atau compiler. Ini adalah perangkat lunak dalam bentuk yang dapat dijalankan di komputer.
- **Set Instruksi:** Pengumpulan semua instruksi yang mungkin untuk komputer tertentu; yaitu, kumpulan instruksi bahasa mesin yang dimengerti oleh prosesor tertentu.
- **Linker:** Program utilitas yang menggabungkan satu atau lebih file yang berisi kode objek dari modul program yang dikompilasi secara terpisah menjadi satu file yang berisi kode yang dapat dimuat atau dapat dieksekusi.
- **Loader:** Program rutin yang menyalin program yang dapat dieksekusi ke dalam memori untuk dieksekusi.
- **Bahasa Mesin, atau Kode Mesin:** Representasi biner dari program komputer yang sebenarnya dibaca dan ditafsirkan oleh komputer. Suatu program dalam kode mesin terdiri dari urutan instruksi mesin (mungkin diselingi dengan data). Petunjuk adalah string biner yang mungkin berukuran sama (misalnya: *word* berukuran 32 bit untuk banyak mikroprosesor RISC modern) atau dengan ukuran berbeda.
- **Object Code:** Representasi bahasa mesin dari kode sumber pemrograman. Kode objek dibuat oleh kompiler atau assembler dan kemudian diubah menjadi kode yang dapat dieksekusi oleh *linker*.

Programer jarang menggunakan bahasa assembly untuk aplikasi atau bahkan program sistem. *High Level Language* (HLL) memberikan kekuatan ekspresif dan keringkasan yang sangat memudahkan tugas programmer. Kekurangan menggunakan bahasa assembly daripada HLL adalah sebagai berikut:

1. Waktu pengembangan. Menulis kode dalam bahasa assembly membutuhkan waktu lebih lama daripada menulis dalam bahasa tingkat tinggi.
2. Keandalan dan keamanan. Sangat mudah untuk membuat kesalahan dalam kode assembly. Assembler tidak memeriksa apakah konvensi pemanggilan dan konvensi register save dipatuhi. Tidak ada yang memeriksa Anda jika jumlah instruksi PUSH dan POP sama di semua cabang dan jalur yang memungkinkan. Ada begitu banyak kemungkinan kesalahan tersembunyi dalam kode rakitan sehingga memengaruhi keandalan dan keamanan proyek kecuali Anda memiliki pendekatan yang sangat sistematis untuk menguji dan memverifikasi
7. Kode sistem dapat menggunakan fungsi intrinsik alih-alih perakitan. Kompiler C ++ modern terbaik memiliki fungsi intrinsik untuk mengakses register kontrol sistem dan instruksi sistem lainnya. Kode assembly tidak lagi diperlukan untuk driver perangkat dan kode sistem lainnya ketika fungsi intrinsik tersedia.
8. Kode aplikasi dapat menggunakan fungsi intrinsik atau kelas vektor alih-alih perakitan. Kompiler C ++ modern terbaik memiliki fungsi intrinsik untuk operasi vektor dan instruksi khusus lainnya yang sebelumnya memerlukan pemrograman perakitan.
9. Compiler telah banyak ditingkatkan dalam beberapa tahun terakhir. Kompiler terbaik sekarang cukup bagus. Dibutuhkan banyak keahlian dan pengalaman untuk mengoptimalkan lebih baik daripada kompiler C ++ terbaik.

Namun masih ada beberapa keuntungan untuk penggunaan bahasa assembly sesekali, termasuk yang berikut:

1. Debugging dan verifikasi. Melihat kode rakitan yang dibuat oleh kompiler atau jendela pembongkaran dalam debugger berguna untuk menemukan kesalahan dan untuk memeriksa seberapa baik kompiler mengoptimalkan bagian tertentu dari kode.
2. Membuat kompiler. Memahami teknik pengkodean perakitan diperlukan untuk membuat kompiler, debugger, dan alat pengembangan lainnya.
3. Sistem tertanam. Sistem tertanam kecil memiliki sumber daya lebih sedikit daripada PC dan mainframe. Pemrograman perakitan dapat diperlukan untuk mengoptimalkan kode untuk kecepatan atau ukuran dalam sistem tertanam kecil.
4. Driver perangkat keras dan kode sistem. Mengakses perangkat keras, register kontrol sistem, dan sebagainya kadang-kadang mungkin sulit atau tidak mungkin dengan kode tingkat tinggi.
5. Mengakses instruksi yang tidak dapat diakses dari bahasa tingkat tinggi. Instruksi perakitan tertentu tidak memiliki padanan bahasa tingkat tinggi.
6. Kode modifikasi diri. Kode *self-modifying* umumnya tidak menguntungkan karena mengganggu *caching kode* yang efisien. Namun, mungkin menguntungkan, misalnya, untuk memasukkan kompiler kecil dalam program matematika di mana fungsi yang ditentukan pengguna harus dihitung berkali-kali.
7. Mengoptimalkan kode untuk ukuran. Ruang penyimpanan dan memori sangat murah saat ini sehingga tidak sebanding dengan upaya untuk menggunakan bahasa rakitan untuk mengurangi ukuran kode. Namun, ukuran *cache* masih merupakan sumber daya kritis sehingga dalam beberapa kasus mungkin berguna untuk mengoptimalkan potongan kode penting untuk ukuran agar membuatnya sesuai dengan *cache* kode.
8. Mengoptimalkan kode untuk kecepatan. Kompiler C ++ modern umumnya mengoptimalkan kode dengan cukup baik dalam banyak kasus. Tetapi masih ada kasus di mana kompiler berkinerja buruk dan di mana peningkatan dramatis dalam kecepatan dapat dicapai dengan pemrograman perakitan yang cermat.
9. Fungsi perpustakaan. Manfaat total dari pengoptimalan kode lebih tinggi di fungsi perpustakaan yang digunakan oleh banyak programmer.

10. Membuat fungsi perpustakaan kompatibel dengan banyak kompiler dan sistem operasi. Dimungkinkan untuk membuat fungsi perpustakaan dengan banyak entri yang kompatibel dengan kompiler yang berbeda dan sistem operasi yang berbeda. Ini membutuhkan pemrograman perakitan.

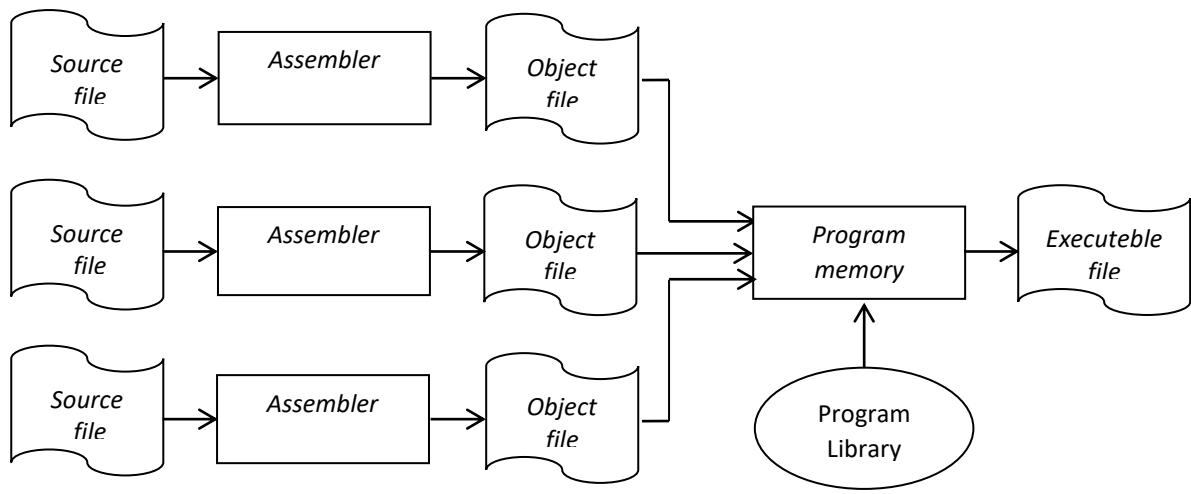
Berikut adalah beberapa istilah yang harus dipahami ketika mempelajari assembler language.

1. **Assembler:** Suatu program yang menerjemahkan bahasa assembly ke dalam kode mesin.
2. **Assembly Language:** Representasi simbolis dari bahasa mesin prosesor tertentu, ditambah dengan jenis pernyataan tambahan yang memfasilitasi penulisan program dan yang memberikan instruksi kepada assembler.
3. **Compiler:** Program yang mengubah program lain dari beberapa bahasa sumber (atau bahasa pemrograman) menjadi bahasa mesin (kode objek). Beberapa kompiler mengeluarkan bahasa rakitan yang kemudian dikonversi ke bahasa mesin oleh assembler terpisah. Komiler dibedakan dari assembler oleh fakta bahwa setiap pernyataan *input* tidak, secara umum, sesuai dengan instruksi mesin tunggal atau urutan instruksi yang tetap. Komiler dapat mendukung fitur-fitur seperti alokasi variabel otomatis, ekspresi aritmetika sewenang-wenang, struktur kontrol seperti *loop FOR* dan *WHILE*, ruang lingkup variabel, operasi *input/output*, fungsi urutan lebih tinggi, dan portabilitas kode sumber.
4. **Executable Code:** Kode mesin yang dihasilkan oleh prosesor bahasa kode sumber seperti assembler atau compiler. Ini adalah perangkat lunak dalam bentuk yang dapat dijalankan di komputer.
5. **Instruction Set:** Kumpulan semua instruksi yang memungkinkan untuk komputer tertentu; yaitu, kumpulan instruksi bahasa mesin yang dimengerti oleh prosesor tertentu.
6. **Linker:** Program utilitas yang menggabungkan satu atau lebih file yang berisi kode objek dari modul program yang dikompilasi secara terpisah menjadi satu file yang berisi kode yang dapat dimuat atau dapat dieksekusi.
7. **Object Code:** Representasi bahasa mesin dari kode sumber pemrograman. Kode objek dibuat oleh kompiler atau assembler dan kemudian diubah menjadi kode yang dapat dieksekusi oleh linker.
8. **Loader:** Program rutin yang menyalin program yang dapat dieksekusi ke dalam memori untuk dieksekusi.
9. **Machine Language** atau **Machine Code:** Representasi biner dari program komputer yang sebenarnya dibaca dan ditafsirkan oleh komputer. Suatu program dalam kode mesin terdiri dari urutan instruksi mesin (mungkin diselingi dengan data). Petunjuk adalah string biner yang mungkin berukuran sama (misalnya *word* berukuran 32-bit untuk banyak mikroprosesor RISC modern) atau dengan ukuran berbeda.

Istilah bahasa rakitan dan bahasa mesin kadang-kadang, secara keliru, digunakan secara sinonim. Bahasa mesin terdiri dari instruksi yang langsung dieksekusi oleh prosesor. Setiap instruksi bahasa mesin adalah string biner yang berisi opcode, referensi *operand*, dan mungkin bit lain yang terkait dengan eksekusi, seperti flag bit. Untuk kenyamanan, alih-alih menulis instruksi sebagai string bit, ini dapat ditulis secara simbolis (dengan nama untuk opcode dan register). Penggunaan nama simbolik pada bahasa rakitan, termasuk menetapkan nama untuk lokasi memori utama tertentu dan lokasi instruksi tertentu, menjadi sangat memudahkan pemrogram. Bahasa assembly juga mencakup pernyataan yang tidak dapat dieksekusi secara langsung tetapi berfungsi sebagai instruksi kepada assembler yang menghasilkan kode mesin dari program bahasa assembly.

Bahasa rakitan adalah representasi simbolis dari pengkodean biner komputer, sering disebut bahasa mesin. Bahasa assembly lebih mudah dibaca daripada bahasa mesin, karena menggunakan simbol daripada bit. Simbol-simbol dalam nama bahasa assembly biasanya muncul dalam pola bit, seperti opcode dan spesi fi k register, sehingga orang dapat membaca dan mengingatnya. Selain itu, bahasa

assembly memungkinkan pemrogram menggunakan label untuk mengidentifikasi dan memberi nama *words* memori tertentu yang menyimpan instruksi atau data.



Gambar 9.1. Proses Pembuatan Executeble File

Assembler memberikan representasi yang lebih bersahabat daripada 0 dan 1 komputer, yang menyederhanakan program menulis dan membaca. Nama simbolis untuk operasi dan lokasi adalah salah satu aspek dari representasi ini. Aspek lainnya adalah fasilitas pemrograman yang meningkatkan kejelasan program. Misalnya, makro, memungkinkan pemrogram untuk memperluas bahasa assembly dengan mendefinisikan operasi baru.

Assembler membaca file sumber bahasa assembly tunggal dan menghasilkan file objek yang berisi instruksi mesin dan informasi pembukuan yang membantu menggabungkan beberapa file objek ke dalam program. Gambar 9.1 mengilustrasikan bagaimana program dibangun. Kebanyakan program terdiri dari beberapa *file* (juga disebut modul) yang ditulis, dikompilasi, dan dirakit secara independen. Sebuah program juga dapat menggunakan rutinitas yang telah ditulis sebelumnya yang disediakan di *Program library*. Sebuah modul biasanya berisi referensi ke subrutin dan data yang didefinisikan dalam modul lain dan di *library*. *Code* dalam modul tidak dapat dieksekusi jika berisi referensi yang belum terselesaikan ke label di *file* atau pustaka objek lain. Alat lain, yang disebut *linker*, menggabungkan kumpulan file objek dan perpustakaan menjadi file yang dapat dijalankan langsung oleh komputer. Secara umum tugas linker adalah:

1. Mencari program library untuk menemukan rutinitas library yang digunakan oleh program
2. Menentukan lokasi memori yang akan ditempati oleh code dari setiap modul dan merelokasi instruksi dengan menyesuaikan referensi absolut
3. Memutuskan referensi di antara file

Tugas pertama linker adalah memastikan bahwa program tidak berisi label yang tidak ditentukan. Linker cocok dengan simbol eksternal dan referensi yang belum terselesaikan dari file program. Simbol eksternal dalam satu file menyelesaikan referensi dari file lain jika keduanya merujuk ke label dengan nama yang sama. Referensi yang tidak cocok berarti simbol digunakan tetapi tidak didefinisikan dimanapun dalam program.

Referensi yang belum terselesaikan pada tahap ini dalam proses linker tidak selalu berarti programmer melakukan kesalahan. Program bisa saja mereferensikan rutinitas pustaka yang kodennya tidak ada dalam file objek yang diteruskan ke linker. Setelah mencocokkan simbol dalam program, linker mencari perpustakaan program sistem untuk menemukan subrutin dan struktur data yang telah ditentukan sebelumnya yang direferensikan oleh program. Pustaka dasar berisi rutinitas yang

membaca dan menulis data, mengalokasikan dan membatalkan alokasi memori, dan melakukan operasi numerik. Perpustakaan lain berisi rutinitas untuk mengakses database atau memanipulasi jendela terminal. Program yang mereferensikan simbol yang belum terselesaikan yang tidak ada di perpustakaan mana pun adalah salah dan tidak dapat ditautkan.

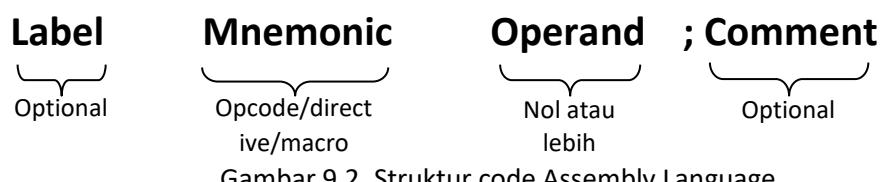
Ketika program menggunakan rutin perpustakaan, linker mengekstrak kode rutin dari perpustakaan dan menggabungkannya ke dalam segmen teks program. Rutinitas baru ini, pada gilirannya, mungkin bergantung pada rutinitas pustaka lainnya, sehingga linker terus mengambil rutinitas pustaka lain hingga tidak ada referensi eksternal yang tidak terselesaikan atau rutin tidak dapat ditemukan.

Jika semua referensi eksternal diselesaikan, linker selanjutnya menentukan lokasi memori yang akan ditempati setiap modul. Karena file dirakit secara terpisah, assembler tidak dapat mengetahui di mana instruksi atau data modul akan ditempatkan relatif terhadap modul lain. Saat linker menempatkan modul dalam memori, semua referensi absolut harus dipindahkan untuk mencerminkan lokasinya yang sebenarnya. Karena linker memiliki informasi lokasi yang mengidentifikasi semua referensi yang dapat direlokasi, maka linker dapat secara efisien menemukan dan melakukan backpatch referensi ini.

Linker menghasilkan file yang dapat dijalankan yang dapat dijalankan di komputer. Biasanya, file ini memiliki format yang sama sebagai file objek, kecuali bahwa file ini tidak berisi referensi atau informasi lokasi yang belum terselesaikan.

9.1. Elemen Assembler Language

Pernyataan dalam *assembler language* pada umumnya memiliki bentuk yang ditunjukkan pada Gambar 9.2. Format ini terdiri dari empat elemen: label, mnemonik, *operand*, dan komentar. Assembler mendefinisikan label setara dengan alamat di mana byte pertama dari kode objek yang dihasilkan untuk instruksi itu akan dimuat. Programer selanjutnya dapat menggunakan label sebagai alamat atau sebagai data di bidang alamat instruksi lain. Assembler menggantikan label dengan nilai yang diberikan ketika membuat program objek. Label paling sering digunakan dalam instruksi cabang.



Gambar 9.2. Struktur code Assembly Language

Berikut adalah contoh penggalan program dalam assembly language:

```
L2:    SUB EAX, EDX ; kurangi isi register EDX dari  
           ; isi EAX dan hasil toko di EAX  
    JG L2          ; lompat ke L2 jika hasil pengurangannya adalah positif
```

Program akan melanjutkan untuk kembali ke lokasi L2 sampai hasilnya nol atau negatif. Jadi, ketika instruksi JG dieksekusi, jika hasilnya positif, maka prosesor akan menempatkan alamat yang setara dengan label L2 pada register Program Counter.

A. Label

Label adalah penamaan yang diberikan untuk lokasi memori instruksi pada baris dimana label tersebut berada. Bagian Label ditulis mulai huruf pertama dari baris, jika baris bersangkutan tidak mengandung

Label maka label tersebut dapat digantikan dengan spasi atau TAB, yakni sebagai tanda pemisah antara bidang Label dan bidang mnemonik. Penamaan Label harus diawali dengan huruf dan bersifat case sensitive, serta tidak boleh mengandung spasi, tidak boleh sama dengan mnemonic atau nama register. Kelebihan menggunakan label adalah sebagai berikut:

1. Label membuat lokasi program lebih mudah ditemukan dan diingat.
2. Label dapat dengan mudah dipindahkan untuk memperbaiki program. Assembler akan secara otomatis mengubah alamat dalam semua instruksi yang menggunakan label ketika program dipasang kembali.
3. Programmer tidak perlu menghitung alamat memori relatif atau absolut, tetapi cukup menggunakan label yang diperlukan.

B. Operand

Pernyataan bahasa rakitan mencakup nol atau lebih *operand*. Setiap *operand* mengidentifikasi nilai langsung, nilai register, atau lokasi memori. Biasanya, bahasa rakitan menyediakan konvensi untuk membedakan antara tiga jenis referensi *operand*, serta konvensi untuk menunjukkan mode pengalamatan.

Untuk arsitektur x86, pernyataan bahasa assembly dapat merujuk ke *operand* register dengan nama. Gambar 9.3 menggambarkan register tujuan umum x86, dengan nama simbolik dan enkode bitnya. Assembler akan menerjemahkan nama simbolik ke dalam pengidentifikasi biner untuk register. Untuk pengalamatan register, nama register digunakan dalam instruksi. Sebagai contoh, MOV ECX, EBX menyalin isi register EBX ke register ECX. Pengalamatan segera (*immediate addressing*) menunjukkan bahwa nilai dikodekan dalam instruksi. Sebagai contoh, MOV EAX, 100H; menyalin nilai heksadesimal 100 ke dalam register EAX. Nilai langsung dapat dinyatakan sebagai angka biner dengan sufiks B atau angka desimal tanpa sufiks. Dengan demikian, pernyataan yang setara dengan yang sebelumnya adalah MOV EAX, 10000000B dan MOV EAX, 256.

General-purpose registers		0	16-bit	32-bit
	AH	AL	AX	EAX (000)
	BH	BL	BX	EBX (011)
	CH	CL	CX	ECX (001)
	DH	DL	DX	EDX (010)
				ESI (110)
				EDI (111)
				EBP (101)
				ESP (100)

Segment registers		0
		CS
		DS
		SS
		ES
		FS
		GS

Sumber: (Brey, 2009)

Gambar 9.3. Penamaan Register pada Intel X86

Pengalamatan langsung mengacu pada lokasi memori dan dinyatakan sebagai perpindahan dari register segmen DS. Ini paling baik dijelaskan dengan contoh. Asumsikan bahwa register segmen data 16-bit berisi nilai 1000H. Kemudian urutan berikut terjadi:

```
MOV AX, 1234H  
MOV [3518H], AX
```

Pertama register 16-bit AX diinisialisasi ke 1234H. Kemudian, di baris dua, isi AX dipindahkan ke alamat logika DS:3518H (pengalamatan segment:offset). Misalkan DS berisi 1000H, alamat logika ini dibentuk dengan menggeser isi DS ke kiri sebanyak 4 bit dan menambahkan 3518H untuk membentuk alamat logika 32-bit, 13518H.

C. Comment

Semua bahasa assembly memungkinkan penempatan komentar dalam program. Sebuah komentar dapat muncul di ujung kanan sebuah pernyataan baris teks, atau dapat menempati seluruh baris teks. Dalam kedua kasus tersebut, komentar dimulai dengan karakter khusus yang memberi sinyal kepada assembler bahwa sisa baris adalah komentar dan harus diabaikan oleh assembler. Biasanya, bahasa rakitan untuk arsitektur x86 menggunakan tanda titik koma (;) untuk karakter khusus.

9.2. Jenis Pernyataan Bahasa Rakitan

Pernyataan bahasa rakitan adalah salah satu dari empat jenis: instruksi, directive, definisi makro, dan komentar.

A. Mnemonic

Sebagian besar pernyataan noncomment dalam program bahasa assembly adalah representasi simbolik dari instruksi bahasa mesin. Hampir selalu, ada hubungan satu-ke-satu antara instruksi bahasa assembly dan instruksi mesin. Assembler menyelesaikan referensi simbolis dan menerjemahkan instruksi bahasa assembly ke dalam string biner yang terdiri dari instruksi mesin.

B. Directive

Directive, juga disebut instruksi semu (*pseudo instructions*), adalah pernyataan bahasa rakitan yang tidak langsung diterjemahkan ke dalam instruksi bahasa mesin. Alih-alih, directive adalah instruksi kepada assembler untuk melakukan tindakan tertentu yang melakukan proses perakitan. Contohnya termasuk yang berikut ini:

1. Tetapkan konstanta
2. Tentukan area memori untuk penyimpanan data
3. Inisialisasi area memori
4. Tempatkan tabel atau data tetap lainnya dalam memori
5. Izinkan referensi ke program lain

Tabel 9.1 mencantumkan beberapa *directive* NASM. Sebagai contoh, perhatikan urutan pernyataan berikut:

L2	DB "A"	; byte diinisialisasi ke kode ASCII untuk A (65)
	MOV AL, [L1]	; salin byte pada L1 ke dalam AL
	MOV EAX, L1	; menyimpan alamat byte pada L1 di EAX
	MOV [L1], AH	; salin konten AH ke byte di L1

Jika label digunakan, itu ditafsirkan sebagai alamat (*offset*) data. Jika label ditempatkan di dalam tanda kurung siku, itu ditafsirkan sebagai data di alamat.

Tabel 9.1. Directive pada NASM Assembly-Language

Nama Directive	Deksripsi	Contoh
DD, DW, DQ, DT (D=double, W=word, Q=quad word, T=ten word)	Inisialisasi ukuran suatu data atau label	L6 DD 1A92H; menginisialisasi ukuran L6 sebesar double word (32 bit)
RESB, RESW, RESD, RESQ, REST	Mencadangkan lokasi yang tidak diinisialisasi	BUFFER RESB 64: mencadangkan 64 bytes pada BUFFER
INCBIN	Menyertakan file biner dalam <i>output</i>	INCHBIN file.dat ; menyertakan file.dat pada <i>output</i>
EQU	Mendefinisikan simbol untuk sebuah nilai konstan	NILAI_1 EQU 25; mendefinisikan NILAI_1 berisi nilai konstan sebesar 25 desimal
TIMES	Ulangin instruksi beberapa kali	ZERROBUF TIMES 64 DB 0 ; inisialisasi BUFFER 64-byte untuk semua zero

Sumber: (Kaushik, 2010)

C. Definisi Makro

Definisi makro mirip dengan subrutin dalam beberapa cara. Subrutin adalah bagian dari program yang ditulis sekali dan dapat digunakan beberapa kali dengan memanggil subrutin dari setiap titik dalam program. Ketika suatu program dikompilasi atau dirakit, subrutin hanya dimuat satu kali. Panggilan ke subrutin mentransfer kontrol ke subrutin dan instruksi pengembalian dalam subrutin mengembalikan kontrol ke titik panggilan. Demikian pula, definisi makro adalah bagian kode yang ditulis oleh programmer sekali dan kemudian dapat digunakan berkali-kali. Perbedaan utama adalah bahwa ketika assembler menemukan panggilan makro, itu menggantikan panggilan makro dengan makro itu sendiri. Proses ini disebut ekspansi makro. Jadi, jika makro didefinisikan dalam program bahasa assembly dan dipanggil 10 kali, maka 10 instance dari makro akan

muncul dalam kode rakitan. Intinya, subrutin ditangani oleh perangkat keras pada saat run time, sedangkan makro ditangani oleh assembler pada waktu perakitan. Makro memberikan keuntungan yang sama dengan subrutin dalam hal pemrograman modular, tetapi tanpa *overhead* runtime dari panggilan subrutin dan kembali..

Dalam NASM dan banyak assembler lainnya, perbedaan dibuat antara makro baris tunggal dan makro multi-baris. Di NASM, makro baris tunggal didefinisikan menggunakan arahan %DEFINE. Berikut adalah contoh di mana beberapa makro baris tunggal diperluas. Pertama, didefinisikan dua makro sebagai berikut:

```
%DEFINE B(X) = 2*X
%DEFINE A(X) = 1 + B(X)
```

Pada titik tertentu dalam program bahasa assembly, pernyataan berikut muncul:

```
MOV AX, A(8)
```

Assembler memperluas pernyataan ini ke:

```
MOV AX, 1+2*8
```

yang berkumpul ke instruksi mesin untuk memindahkan nilai langsung 17 untuk register AX.

Makro multiline didefinisikan menggunakan mnemonic % MACRO. Berikut adalah contoh definisi makro multiline:

```
% PROGRAM MAKRO 1
```

```
PUSH EBP      ; simpan isi register EBP ke stack yang dituntuk oleh ESP dan  
              ; mengurangi isi ESP dengan 4  
MOV EBP, ESP ; menyalin isi ESP ke EBP  
SUB ESP, %1   ; kurangi nilai parameter pertama dari ESP
```

Angka 1 setelah nama makro pada baris %MACRO mendefinisikan jumlah parameter yang makro harapkan untuk diterima. Penggunaan %1 di dalam definisi makro mengacu pada parameter pertama ke panggilan makro.

Contoh makro call

```
MYFUNC: PROLOG 12
```

memperluas ke baris kode berikut:

```
MYFUNC: PUSH EBP  
MOV EBP, ESP  
SUB ESP, 12
```

10. Reduce Instruction Set Computer

Sejak pengembangan komputer program tersimpan sekitar tahun 1950, ada beberapa inovasi nyata di bidang organisasi dan arsitektur komputer. Berikut ini adalah beberapa kemajuan besar sejak kelahiran komputer :

- Konsep keluarga: Diperkenalkan oleh IBM dengan System/360 pada tahun 1964, diikuti segera setelahnya oleh DEC, dengan PDP-8. Konsep keluarga memisahkan arsitektur mesin dari implementasinya. Seperangkat komputer ditawarkan, dengan karakteristik harga/ kinerja yang berbeda, yang menghadirkan arsitektur yang sama kepada pengguna. Perbedaan harga dan kinerja disebabkan oleh implementasi yang berbeda dari arsitektur yang sama.
- *Control Unit* yang diprogram dengan mikroponik: Disarankan oleh Wilkes pada tahun 1951 dan diperkenalkan oleh IBM pada jalur S/360 pada tahun 1964. Pemrograman mikro memudahkan tugas merancang dan mengimplementasikan unit kendali dan memberikan dukungan untuk konsep keluarga.
- Memori *cache*: Pertama kali diperkenalkan secara komersial pada IBM S/360 Model 85 pada tahun 1968. Penyisipan elemen ini ke dalam hierarki memori secara dramatis meningkatkan kinerja.
- *Pipelining*: Suatu cara untuk memperkenalkan paralelisme ke dalam sifat yang secara esensial berurutan dari program instruksi mesin. Contohnya adalah instruksi *pipeline* dan pemrosesan vektor.
- Berbagai prosesor: Kategori ini mencakup sejumlah organisasi dan tujuan yang berbeda.
- Arsitektur *Reduce Instruction Set Computer* (RISC).

Ketika muncul, arsitektur RISC adalah perubahan dramatis dari tren historis dalam arsitektur prosesor. Analisis arsitektur RISC memusatkan perhatian pada banyak masalah penting dalam organisasi dan arsitektur komputer. RISC dirancang secara arsitektural untuk mengakomodasi implementasi *pipelined*. RISC dirancang dengan implementasi *pipeline* sederhana, yang memungkinkan perangkat keras dibuat sesederhana dan secepat mungkin sambil menyerahkan operasi yang kompleks (atau apa pun yang tidak dapat dilakukan dengan cepat di perangkat keras) ke perangkat lunak.

Karakteristik pembeda utama dari arsitektur RISC yang khas dan implementasinya dapat diringkas sebagai berikut:

- Panjang instruksi tetap (*fixed-length instructions*) digunakan untuk menyederhanakan pengambilan instruksi.
- Mesin hanya memiliki beberapa format instruksi untuk menyederhanakan dekode instruksi.
- Instruksi yang berhubungan dengan memori hanya LOAD dan STORE, instruksi lain dilakukan dalam register internal prosesor. Cara ini menyederhanakan mode pengalamatan (*addressing mode*) karena mode pengalamatan yang kompleks dapat memperlambat mesin dan jarang digunakan oleh kompiler
- Instruksi memiliki fungsi sederhana, yang membantu menjaga desain unit kontrol tetap sederhana.

- Arsitekturnya dirancang untuk implementasi *pipelined*, untuk mengoptimalkan kecepatan eksekusi.
- Prosesor RISC memerlukan waktu kompilasi yang lebih lama daripada prosesor CISC. Karena sedikitnya pilihan instruksi dan mode pengalamatan yang dimiliki prosesor RISC, maka diperlukan optimalisasi perancangan kompilator agar mampu menyusun urutan instruksi-instruksi sederhana secara efisien dan sesuai dengan bahasa pemrograman yang dipilih. Optimasi fungsi oleh kompilator ditekankan karena arsitekturnya dirancang untuk mendukung bahasa tingkat tinggi daripada pemrograman perakitan.
- Kompleksitas ada di kompilator (yang hanya memengaruhi kinerja kompilator), bukan di perangkat keras (yang akan memengaruhi kinerja setiap program yang berjalan di mesin).

Karakteristik tambahan dan sekunder yang lazim di mesin RISC meliputi:

- Instruksi tiga *operand* memudahkan kompilator untuk mengoptimalkan kode.
- Satu set register yang besar (biasanya 32 register atau lebih) dimungkinkan karena mesin memiliki unit kontrol yang kecil dan terprogram dan diinginkan karena kebutuhan kompilator untuk mengoptimalkan kode untuk arsitektur penyimpanan-beban.
- Instruksi dijalankan dalam satu siklus *clock* (atau setidaknya sebagian besar dari mereka tampaknya, karena implementasi *pipelined*).
- Instruksi transfer kontrol yang tertunda digunakan untuk meminimalkan gangguan pada pipa.
- Slot penundaan di belakang beban dan penyimpanan membantu menutupi latensi akses memori.
- Arsitektur Harvard yang dimodifikasi digunakan untuk menjaga akses memori untuk data agar tidak mengganggu pengambilan instruksi dan dengan demikian menjaga *pipeline* tetap penuh.
- *Cache on-chip* dimungkinkan karena unit kontrol kecil yang tertanam dan diperlukan untuk mempercepat pengambilan instruksi dan menjaga latensi beban dan penyimpanan seminimal mungkin.

Tidak semua arsitektur RISC menampilkan semua fitur di atas, dan tidak setiap arsitektur CISC menghindari semuanya. Namun, fitur tersebut berfungsi sebagai titik referensi yang dapat membantu untuk memahami dan mengklasifikasikan arsitektur baru atau asing. Apa yang membedakan arsitektur RISC dari CISC bukan pada daftar fitur, melainkan asumsi yang mendasari desainnya. RISC bukanlah mesin khusus, tetapi filosofi desain mesin yang menekankan pada menjaga perangkat keras sesederhana dan secepat mungkin sambil memindahkan kompleksitas ke perangkat lunak. (Perhatikan bahwa "set instruksi yang dikurangi" tidak berarti dikurangi menjadi mutlak, minimal yang diperlukan untuk komputasi. Ini berarti dikurangi menjadi hanya fitur-fitur yang berkontribusi pada peningkatan kinerja. Fitur apa pun yang memperlambat mesin dalam beberapa hal tanpa lebih dari mengimbangi itu dengan meningkatkan kinerja dalam beberapa cara dihilangkan. Mengingat serangkaian operasi mesin yang sesuai dan terbatas, filosofi RISC secara efektif menyatakan bahwa yang dapat menjalankan sebagian besar operasi dalam waktu singkat akan menang. Terserah pada kompiler untuk menemukan pengkodean terbaik dari algoritma tingkat tinggi ke dalam instruksi bahasa mesin yang dijalankan oleh prosesor yang disalurkan cepat ini, dan dengan demikian memaksimalkan kinerja sistem.

Meskipun akronim RISC diciptakan sekitar tahun 1980 oleh David Patterson dari University of California di Berkeley (dan membutuhkan beberapa tahun setelah itu untuk masuk ke bahasa umum), banyak konsep yang terkandung dalam filosofi RISC sudah ada sebelumnya. Arsitektur RISC lain yang ada sebelum nama itu sendiri muncul adalah IBM 801. Proyek ini dimulai pada tahun 1975, beberapa tahun sebelum karya Patterson di Berkeley dan John Hennessy di Stanford. IBM 801 pada awalnya dirancang sebagai komputer mini untuk digunakan sebagai prosesor kontrol untuk sistem pertukaran telepon, sebuah proyek yang akhirnya dibatalkan dan tidak pernah berhasil masuk ke pasar komersial sebagai sistem yang berdiri sendiri.

Hanya beberapa saat setelah proyek IBM 801 berlangsung, Patterson mulai mengembangkan mikroprosesor RISC I di Berkeley. Sementara, kurang lebih secara bersamaan, Hennessy memulai proyek MIPS di Stanford. (Kedua proyek tersebut didanai oleh *Defense Advanced Research Projects Agency* [DARPA].) Kedua arsitektur ini disempurnakan selama beberapa tahun pada awal 1980-an; turunan dari keduanya akhirnya dikomersialkan. Desain kedua Patterson, RISC II diadaptasi oleh Sun Microsystems (sekarang Oracle) untuk membuat arsitektur SPARC, dan Hennessy menciptakan sebuah perusahaan bernama MIPS Computer Systems (kemudian menjadi MIPS Technologies, diakuisisi pada 2013 oleh *Imagination Technologies*). Desain Patterson sangat inovatif dalam penggunaan skema pemetaan ulang register yang dikenal sebagai *overlapping register windows* untuk mengurangi biaya kinerja yang terkait dengan panggilan prosedur. Tabel 10.1 membandingkan beberapa sistem RISC dan non-RISC.

Tabel 10.1. Beberapa Karakteristik Prosesor CISC, RISC, dan Superscalar

	Complex Instruction Set Computer (CISC)			Reduced Instruction Set Computer (RISC)		Superscalar		
Characteristic	IBM 370/168	VAX 11/780	Intel 80486	SPARC	MIPS R4000	Power PC	Ultra SPARC	MIPS R10000
Year developed	1973	1978	1989	1987	1991	1993	1996	1996
Number of instructions	208	303	235	69	94	225		
Instruction size (bytes)	2–6	2–57	1–11	4	4	4	4	4
Addressing modes	4	22	11	1	1	2	1	1
Number of general-purpose registers	16	16	8	40–520	32	32	40–520	32
Control memory size (kbits)	420	480	246	—	—	—	—	—
Cache size (kB)	64	64	8	32	128	16–32	32	64

Sumber: (Patterson & Hennessy, 2014))

Meskipun arsitektur RISC telah didefinisikan dan dirancang dalam berbagai cara oleh berbagai kelompok, elemen kunci yang dibagikan oleh sebagian besar desain adalah:

- 1) Sejumlah besar register tujuan umum, dan/atau penggunaan teknologi kompiler untuk mengoptimalkan penggunaan register.
- 2) Set instruksi terbatas dan sederhana.
- 3) Penekanan pada optimalisasi jalur instruksi.

10.1. Karakteristik Eksekusi Instruksi

Salah satu bentuk evolusi yang paling terlihat terkait dengan komputer adalah bahasa pemrograman. Ketika biaya perangkat keras turun, biaya relatif perangkat lunak telah meningkat. Dengan demikian, biaya utama dalam siklus hidup suatu sistem adalah perangkat lunak, bukan perangkat keras. Menambah biaya, dan ketidaknyamanan, adalah elemen tidak dapat diandalkan: itu umum untuk program, baik sistem dan aplikasi, untuk terus menunjukkan bug baru setelah bertahun-tahun beroperasi.

Tanggapan dari para peneliti dan industri adalah mengembangkan bahasa pemrograman tingkat tinggi yang lebih kuat dan kompleks. *High Level Language* (HLL) ini memiliki kelebihan sebagai berikut:

- Memungkinkan programmer untuk mengekspresikan algoritma lebih ringkas;
- Memungkinkan kompiler untuk mengurus detail yang tidak penting dalam ekspresi algoritma programmer;
- Aering mendukung secara alami penggunaan pemrograman terstruktur dan/atau desain berorientasi objek. Sayangnya, solusi ini memunculkan masalah yang dirasakan, yang dikenal sebagai kesenjangan semantik, perbedaan antara operasi yang disediakan dalam HLL dan yang disediakan dalam arsitektur komputer. Gejala dari celah ini diduga termasuk ketidakefisienan eksekusi, ukuran program alat berat yang berlebihan, dan kompleksitas penyusun. Desainer merespons dengan arsitektur yang dimaksudkan untuk menutup celah ini. Fitur utama termasuk set instruksi besar, puluhan mode pengalamatan, dan berbagai pernyataan HLL diimplementasikan dalam perangkat keras. Contoh yang terakhir adalah instruksi mesin KASUS pada VAX. Set instruksi kompleks seperti itu dimaksudkan untuk:
 - Kemudahan tugas penulis kompiler.
 - Meningkatkan efisiensi eksekusi, karena urutan operasi yang kompleks dapat diimplementasikan dalam mikrokode.
 - Berikan dukungan untuk HLL yang lebih kompleks dan canggih.

Sementara itu, sejumlah penelitian telah dilakukan selama bertahun-tahun untuk menentukan karakteristik dan pola pelaksanaan instruksi mesin yang dihasilkan dari program HLL. Hasil penelitian ini menginspirasi beberapa peneliti untuk mencari pendekatan yang berbeda: yaitu, membuat arsitektur yang mendukung HLL lebih sederhana, daripada lebih kompleks.

Untuk memahami garis penalaran para pendukung RISC, dimulai dengan tinjauan singkat mengenai karakteristik pelaksanaan instruksi, sebagai berikut:

- Operasi yang dilakukan: Ini menentukan fungsi yang harus dilakukan oleh prosesor dan interaksinya dengan memori.
- *Operand* yang digunakan: Jenis *operand* dan frekuensi penggunaannya menentukan organisasi memori untuk menyimpannya dan mode pengalamatan untuk mengaksesnya.
- Urutan eksekusi: Ini menentukan organisasi kontrol dan saluran pipa.

Operasi

Kolom kedua dan ketiga dalam Tabel 10.2 menunjukkan frekuensi relatif terjadinya berbagai pernyataan HLL dalam berbagai program; data diperoleh dengan mengamati kejadian dalam menjalankan program bukan hanya berapa kali pernyataan itu muncul dalam kode sumber. Karenanya metrik ini menangkap perilaku dinamis. Untuk mendapatkan data dalam kolom empat dan lima (bobot instruksi mesin), setiap nilai pada kolom kedua dan ketiga dikalikan dengan jumlah instruksi mesin yang dihasilkan oleh kompiler. Hasil ini kemudian dinormalisasi sehingga kolom empat dan lima menunjukkan frekuensi relatif dari kejadian, ditimbang dengan jumlah instruksi mesin per pernyataan HLL. Demikian pula, kolom keenam dan ketujuh diperoleh dengan mengalikan frekuensi kemunculan setiap jenis pernyataan dengan jumlah relatif dari referensi memori yang disebabkan oleh setiap pernyataan. Data dalam kolom empat sampai tujuh memberikan ukuran pengganti waktu aktual yang dihabiskan untuk mengeksekusi berbagai jenis pernyataan. Hasil menunjukkan bahwa panggilan prosedur/pengembalian adalah operasi yang paling memakan waktu dalam program HLL khas.

Tabel 10.2. *Weighted Relative Dynamic Frequency* dari Karakteristik Operasi HLL

	Dynamic Occurrence		Machine-Instruction Weighted		Memory-Reference Weighted	
	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%

Sumber: (Patterson & Hennessy, 2014)

Tabel 10.2 menunjukkan dampak kinerja relatif dari berbagai jenis pernyataan dalam HLL, ketika HLL dikompilasi untuk arsitektur kumpulan instruksi kontemporer yang khas. Beberapa arsitektur lain dapat menghasilkan hasil yang berbeda. Namun, penelitian ini menghasilkan hasil yang representatif untuk arsitektur CISC kontemporer.

Operand

Studi Patterson melihat frekuensi dinamis kemunculan kelas variabel (Tabel 10.3). Hasilnya, konsisten antara program Pascal dan C, menunjukkan bahwa sebagian besar referensi adalah variabel skalar sederhana. Lebih lanjut, lebih dari 80% skalar adalah variabel lokal (untuk prosedur). Selain itu, setiap referensi ke array atau struktur memerlukan referensi ke indeks atau pointer, yang lagi-lagi biasanya skalar lokal. Dengan demikian, ada banyak referensi untuk skalar, dan ini sangat lokal.

Tabel 10.3. Persentase Dinamik dari *Operand*

	Pascal	C	Average
Integer constant	16%	23%	20%
Scalar variable	58%	53%	55%
Array/Structure	26%	26%	25%

Sumber: (Patterson & Hennessy, 2014)

Studi Patterson meneliti perilaku dinamis dari program HLL, independen dari arsitektur yang mendasarinya. Seperti dibahas sebelumnya, perlu berurusan dengan arsitektur aktual untuk memeriksa perilaku program lebih dalam.

Panggilan Prosedur

Panggilan prosedur dan pengembalian merupakan aspek penting dari program HLL. Bukti (Tabel 10.4) menunjukkan bahwa ini adalah operasi yang paling memakan waktu dalam program HLL terkompilasi. Dengan demikian, akan menguntungkan untuk mempertimbangkan cara-cara melaksanakan operasi ini secara efisien. Dua aspek yang signifikan: jumlah parameter dan variabel yang berhubungan dengan prosedur, dan kedalaman bersarang.

Studi Tanenbaum menemukan bahwa 98% prosedur yang disebut secara dinamis dilewatkan lebih sedikit dari enam argumen dan bahwa 92% di antaranya menggunakan kurang dari enam variabel skalar lokal. Hasil serupa dilaporkan oleh tim Berkeley RISC, seperti yang ditunjukkan pada Tabel 10.4. Hasil ini menunjukkan bahwa jumlah *word* yang diperlukan per aktivasi prosedur tidak besar. Studi yang dilaporkan sebelumnya menunjukkan bahwa proporsi referensi *operand* yang tinggi adalah untuk variabel skalar lokal. Studi-studi ini menunjukkan bahwa referensi tersebut sebenarnya terbatas pada variabel yang relatif sedikit.

Tabel 10.4. *Procedure Argument* dan *Local Scalar Variabel*

Percentase Prosedur Call yang Dieksekusi	Compiler, Interpreter, dan Typesetter	Small Nonnumeric Programs
> 3 argument	0-7%	0-5%
> 3 argument	0-3%	0%
> 8 words of arguments dan local scalars	1-20%	0-6%
> 12 words of arguments dan local scalars	1-6%	0-3%

Sumber: (Patterson & Hennessy, 2014)

Kelompok Berkeley yang sama juga melihat pola pemanggilan prosedur dan pengembalian dalam program HLL. Mereka menemukan bahwa jarang memiliki urutan prosedur panggilan yang panjang tanpa Interupsi diikuti oleh urutan pengembalian yang sesuai.

Implikasi Sejumlah kelompok telah melihat hasil seperti yang baru saja dilaporkan dan telah menyimpulkan bahwa upaya untuk membuat arsitektur set instruksi dekat dengan HLL bukanlah strategi desain yang paling efektif. Sebaliknya, HLL dapat didukung dengan mengoptimalkan kinerja fitur yang paling memakan waktu dari program HLL biasa.

Generalisasi dari karya sejumlah peneliti, tiga elemen muncul yang, pada umumnya, mencirikan arsitektur RISC. Pertama, gunakan register dalam jumlah besar atau gunakan kompiler untuk mengoptimalkan penggunaan register. Ini dimaksudkan untuk mengoptimalkan referensi *operand*. Studi yang baru saja dibahas menunjukkan bahwa ada beberapa referensi per pernyataan HLL dan bahwa ada proporsi yang tinggi dari pernyataan pemindahan (penugasan). Ini, ditambah dengan lokalitas dan dominasi referensi skalar, menunjukkan bahwa kinerja dapat ditingkatkan dengan mengurangi referensi memori dengan mengorbankan lebih banyak referensi register. Karena lokasi referensi ini, set register diperluas tampaknya praktis.

Kedua, perhatian yang cermat perlu diberikan pada desain pipa instruksi (*pipelined instruction*). Karena proporsi yang tinggi dari instruksi panggilan cabang dan prosedur, pipa instruksi langsung tidak efisien. Ini memanifestasikan dirinya sebagai sebagian besar instruksi yang diambil sebelumnya tetapi tidak pernah dieksekusi.

Akhirnya, satu set instruksi yang terdiri dari primitif kinerja tinggi diindikasikan. Instruksi harus memiliki biaya yang dapat diprediksi (diukur dalam waktu eksekusi, ukuran kode, dan semakin, dalam pembuangan energi) dan konsisten dengan implementasi kinerja tinggi (yang selaras dengan biaya waktu eksekusi yang dapat diprediksi).

10.2. Penggunaan Register File yang Besar

Alasan penyimpanan register dikarenakan register adalah perangkat penyimpanan tercepat yang tersedia, lebih cepat daripada memori utama dan *cache*. File register secara fisik kecil, pada *chip* yang sama dengan ALU dan *Control Unit*, dan menggunakan alamat yang jauh lebih pendek daripada alamat untuk *cache* dan memori. Dengan demikian, diperlukan strategi yang akan memungkinkan *operand* yang paling sering diakses disimpan dalam register dan untuk meminimalkan operasi memori register.

Ada dua pendekatan dasar yang mungkin, satu didasarkan pada perangkat lunak dan yang lainnya pada perangkat keras. Pendekatan perangkat lunak adalah mengandalkan kompiler untuk memaksimalkan penggunaan register. Kompiler akan mencoba untuk menetapkan register ke variabel-variabel yang akan paling banyak digunakan dalam periode waktu tertentu. Pendekatan ini membutuhkan penggunaan algoritma analisis program yang canggih. Pendekatan perangkat keras hanya dengan menggunakan lebih banyak register sehingga lebih banyak variabel dapat disimpan dalam register untuk periode waktu yang lebih lama.

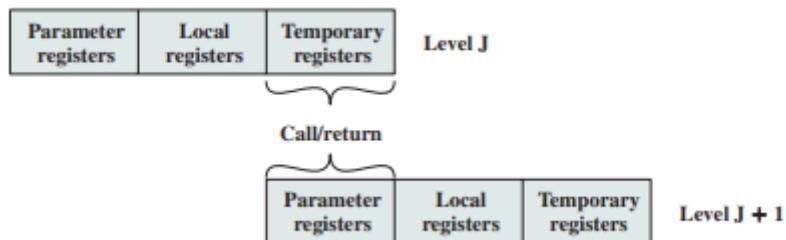
10.3. Register Windows

Karena sebagian besar referensi *operand* adalah untuk skalar lokal, pendekatan yang jelas adalah menyimpannya dalam register, dengan mungkin beberapa register dicadangkan untuk variabel global. Masalahnya adalah bahwa definisi perubahan lokal dengan setiap prosedur call dan return, adalah operasi yang sering terjadi. Pada setiap panggilan prosedur, variabel lokal harus disimpan dari register ke dalam memori, sehingga register dapat digunakan kembali oleh prosedur yang dipanggil. Selanjutnya, parameter harus dilewati. Saat kembali, variabel dari prosedur pemanggilan harus

dikembalikan (dimasukkan kembali ke register) dan hasilnya harus dikembalikan ke prosedur pemanggilan.

Solusi pertama adalah, prosedur khas hanya mempekerjakan beberapa parameter yang lulus dan variabel lokal (Tabel 10.4). Kedua, kedalaman aktivasi prosedur berfluktuasi dalam kisaran yang relatif sempit (Gambar 10.1). Untuk mengeksplorasi properti ini, beberapa set register kecil digunakan, masing-masing ditugaskan untuk prosedur yang berbeda. Panggilan prosedur secara otomatis mengalihkan prosesor untuk menggunakan jendela register berukuran tetap yang berbeda, daripada menyimpan register dalam memori. Windows untuk prosedur yang berdekatan tumpang tindih untuk memungkinkan melewati parameter.

Konsep tersebut diilustrasikan pada Gambar 10.1. Kapan saja, hanya satu jendela register yang terlihat dan dapat dialamatkan seolah-olah itu adalah satu-satunya set register (misalnya Alamat 0 hingga N-1). Jendela dibagi menjadi tiga area ukuran tetap. Register parameter menahan parameter yang diturunkan dari prosedur yang disebut prosedur saat ini dan menahan hasil untuk dilewati kembali. Register lokal digunakan untuk variabel lokal, seperti yang ditugaskan oleh kompiler. Register sementara digunakan untuk bertukar parameter dan hasil dengan level yang lebih rendah berikutnya (prosedur dipanggil oleh prosedur saat ini). Register sementara di satu tingkat secara fisik sama dengan register parameter di tingkat bawah berikutnya. Tumpang tindih ini memungkinkan parameter untuk dilewati tanpa pergerakan data yang sebenarnya. Perlu diingat bahwa, kecuali untuk tumpang tindih, register pada dua level berbeda secara fisik. Yaitu, parameter dan register lokal pada level J terpisah dari register lokal dan sementara pada level J + 1.



Sumber: (Stallings, 2016)

Gambar 10.1. Overlapping Register Windows

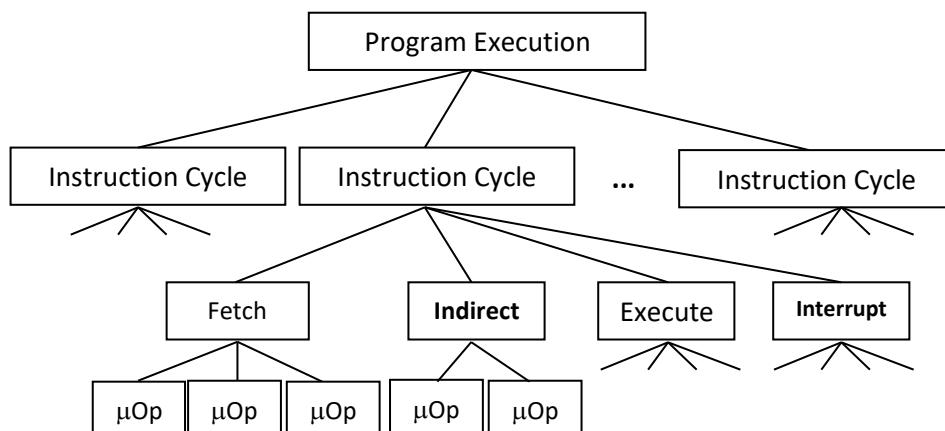
Untuk menangani kemungkinan pola panggilan dan pengembalian, jumlah jendela register harus tidak dibatasi. Sebagai gantinya, register windows dapat digunakan untuk menyimpan beberapa aktivasi prosedur terbaru. Aktivasi yang lebih lama harus disimpan dalam memori dan kemudian dipulihkan ketika kedalaman sarang menurun. Organisasi file register adalah sebagai penyangga bundar dari jendela yang tumpang tindih.

11. Control Unit

11.1. Mikro Operasi.

Dalam menjalankan suatu program, komputer melakukan urutan siklus instruksi, dengan satu instruksi mesin per siklus. Urutan siklus instruksi ini tidak harus sama dengan urutan instruksi tertulis yang membentuk program, karena keberadaan instruksi percabangan.

Setiap siklus instruksi terdiri dari sejumlah unit yang lebih kecil, dengan unit yang paling mendasar adalah *fetch*, *indirect*, *execute*, dan *interrupt*. Namun, untuk merancang *Control Unit*, perlu memecah deskripsi pada setiap unit tersebut. Bahkan, masing-masing siklus yang lebih kecil melibatkan serangkaian langkah, yang melibatkan register prosesor. Langkah-langkah ini disebut sebagai operasi mikro (*micro operation*). Awalan mikro mengacu pada fakta bahwa setiap langkah sangat sederhana dan hanya menghasilkan sedikit aksi. Gambar 11.1 menunjukkan hubungan antara berbagai konsep tersebut. Untuk meringkas, pelaksanaan suatu program terdiri dari eksekusi berurutan dari instruksi. Setiap instruksi dieksekusi selama siklus instruksi (*instruction cycle*) yang terdiri dari subsiklus yang lebih pendek (misalnya: *fetch*, *indirect*, *execute*, *interrupt*). Eksekusi setiap sub siklus melibatkan satu atau lebih operasi yang lebih pendek, yaitu operasi mikro. Operasi mikro adalah operasi fungsional, atau atom, dari suatu prosesor.



Sumber: (Stallings, 2016)

Gambar 11.1. Elemen Penyusun Eksekusi Program

A. Fetch Cycle

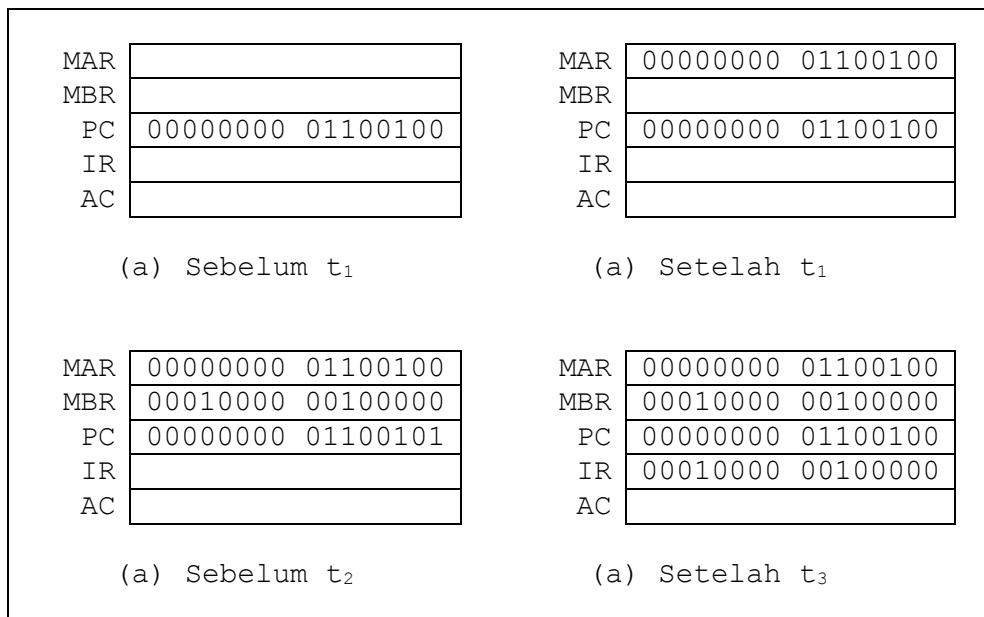
Siklus pengambilan (*fetch cycle*) adalah siklus yang terjadi pada awal setiap siklus instruksi dan menyebabkan instruksi diambil dari memori. Untuk keperluan pembahasan, diasumsikan terdapat Empat register terlibat, yaitu:

- 1) *Memory Address Register (MAR)*. Register ini terhubung ke jalur bus alamat sistem, dan menentukan alamat dalam memori untuk operasi baca atau tulis.
- 2) *Memory Buffer Register (MBR)*. Register ini terhubung ke jalur data bus sistem, dan berisi nilai yang akan disimpan dalam memori atau nilai terakhir yang dibaca dari memori
- 3) *Program Counter (PC)*. Register ini selalu menyimpan alamat instruksi selanjutnya yang akan diambil.
- 4) *Instruction Register (IR)*. Register ini menyimpan instruksi terakhir yang diambil.

Berikut adalah urutan kejadian untuk siklus pengambilan (*fetch cycle*) dari sudut pandang efeknya pada register prosesor. Contoh pada Gambar 11.2, pada awal siklus pengambilan, alamat instruksi berikutnya yang akan dieksekusi ada di register *Program Counter* (PC); dalam contoh ini, alamatnya adalah 1100100.

Langkah pertama adalah memindahkan alamat itu ke *Memory Address Register (MAR)* karena ini adalah satu-satunya register yang terhubung ke jalur alamat bus sistem.

Langkah kedua adalah membawa instruksi. Alamat yang diinginkan (dalam MAR) ditempatkan pada bus alamat, *Control Unit* mengeluarkan perintah READ pada bus kontrol, dan hasilnya muncul pada bus data dan disalin ke *Memory Buffer Register (MBR)*. Diperlukan juga untuk menambah Program Counter (PC) dengan panjang instruksi untuk merujuk ke instruksi selanjutnya. Karena dua tindakan ini (membaca *word* dari memori dan increment PC) tidak saling mengganggu, maka dapat dilakukan secara bersamaan untuk menghemat waktu.



Sumber: (Stallings, 2016)

Gambar 11.2. Urutan Kejadian Fetch cycle

Langkah ketiga adalah memindahkan konten MBR ke Instruction Register (IR). Ini membebaskan MBR untuk digunakan selama siklus tidak langsung yang memungkinkan.

Dengan demikian, siklus pengambilan (*fetch cycle*) sederhana sebenarnya terdiri dari tiga langkah dan empat operasi mikro. Setiap operasi mikro melibatkan perpindahan data ke dalam atau keluar dari

register. Selama gerakan-gerakan ini tidak saling mengganggu, untuk menghemat waktu, beberapa di antaranya dapat dilakukan secara bersamaan dalam satu langkah. Secara simbolis, urutan kejadian ini dapat ditulis sebagai berikut:

t1:	MAR \leftarrow (PC)	; memindahkan isi PC ke MAR
t2:	MBR \leftarrow Memory	; memindahkan isi memori yang alamatnya ditentukan oleh ; MAR ke dalam MBR
	PC \leftarrow (PC) + I	; tambahkan I kedalam PC ; I adalah panjang instruksi
t3:	IR \leftarrow (MBR)	; memindahkan isi MBR ke dalam IR Pengelompokan operasi mikro harus mengikuti aturan sederhana berikut:

1. Urutan kejadian yang tepat harus diikuti. Jadi (MAR \leftarrow (PC)) harus mendahului (MBR \leftarrow Memory) karena operasi membaca memori menggunakan alamat dalam MAR.
2. Konflik harus dihindari. membaca dan menulis dari register yang sama dalam satu unit waktu tidak diijinkan, karena hasilnya tidak dapat diprediksi. Misalnya, operasi mikro (MBR \leftarrow Memory) dan (IR \leftarrow MBR) tidak boleh terjadi pada unit waktu yang sama.
3. bersamaan dengan pemindahan isi memori ke MBR, maka PC harus selalu ditambah dengan panjang Instruksi yang dipindahkan ke MBR.

B. Siklus Tidak Langsung (*Indirect Cycle*).

Setelah instruksi diambil, langkah selanjutnya adalah mengambil *operand sumber* (*source operand*). Melanjutkan contoh sederhana sebelumnya, diasumsikan format instruksi adalah satu alamat, dengan pengalaman langsung (*direct addressing*) dan tidak langsung diizinkan. Jika instruksi menentukan alamat tidak langsung, maka siklus tidak langsung harus mendahului siklus eksekusi. Aliran data agak berbeda dari yang ditunjukkan pada Gambar 11.2 (Aliran Data, Siklus Tidak Langsung) dan mencakup operasi mikro berikut:

t1:	MAR \leftarrow (IR(Address))	; isi bidang Address pada IR, dipindahkan ke MAR
t2:	MBR \leftarrow Memory	; isi Memori yang alamatnya sudah dispesifikasi oleh ; MAR dipindahkan ke MBR
t3:	IR(Address) \leftarrow (MBR(Address))	; isi MBR dipindahkan ke bidang Address pada IR

C. *Interrupt Cycle*

Pada penyelesaian siklus eksekusi, test dilakukan untuk menentukan apakah ada Interupsi yang diaktifkan telah terjadi. Jika demikian, siklus interupsi terjadi. Sifat siklus ini sangat bervariasi dari satu mesin ke mesin lainnya. Berikut adalah urutan kejadian yang sangat sederhana dari siklus interrupt:

t1:	MBR \leftarrow (PC)
t2:	MAR \leftarrow Save_Address
	PC \leftarrow Routine_Address
t3:	Memori \leftarrow (MBR)

Pada langkah pertama, isi *Program Counter* (PC) ditransfer ke MBR, sehingga mereka dapat disimpan untuk kembali dari interupsi. Kemudian MAR dimuat dengan alamat di mana isi *Program Counter* (PC) akan disimpan, dan *Program Counter* (PC) dimuat dengan alamat dimulainya rutin pemrosesan-interupsi. Kedua tindakan ini masing-masing dapat menjadi operasi mikro tunggal. Namun, karena sebagian besar prosesor menyediakan beberapa tipe dan/atau level interupsi, mungkin diperlukan satu atau lebih operasi mikro tambahan untuk mendapatkan Save_Address dan Routine_Address sebelum mereka dapat ditransfer secara berurutan, masing-masing. Bagaimanapun, setelah ini dilakukan, langkah terakhir adalah menyimpan MBR, yang berisi nilai lama *Program Counter* (PC, ke dalam memori. Prosesor sekarang siap untuk memulai siklus instruksi berikutnya.

D. Exexecute Cycle

Siklus pengambilan (*fetch cycle*), tidak langsung (*indirect*), dan interupsi masing-masing melibatkan urutan kecil operasi mikro dan, dalam setiap kasus, operasi mikro yang sama diulang setiap kali. Namun pada siklus eksekusi memiliki rangkaian operasi mikro yang berbeda. Karena adanya beragam opcode, maka terdapat sejumlah rangkaian operasi mikro yang berbeda yang dapat terjadi. *Control Unit* memeriksa *opcode* dan menghasilkan urutan operasi mikro berdasarkan nilai *opcode*. Ini disebut sebagai instruksi decoding. Berikut adalah beberapa contoh:

Contoh 1:

ADD R1, X

yang menambahkan isi (konten) lokasi X ke register R1. Urutan operasi mikro berikut mungkin terjadi:

t1: MAR \leftarrow (IR (alamat))
t2: MBR \leftarrow Memori
t3: R1 \leftarrow (R1) + (MBR)

Pada langkah pertama, bidang alamat pada IR dimuat ke dalam MAR. Kemudian lokasi memori yang dirujuk dibaca. Akhirnya, isi R1 dan MBR ditambahkan oleh ALU. Operasi mikro tambahan mungkin diperlukan untuk mengekstrak referensi register dari IR dan mungkin untuk tahap *input* atau *output* ALU di beberapa register perantara.

Contoh 2:

ISZ X

Isi lokasi X bertambah 1. Jika hasilnya 0, instruksi selanjutnya dilewati. Urutan operasi mikro yang memungkinkan adalah

t1: MAR \leftarrow (IR (alamat))
t2: MBR \leftarrow Memori
t3: MBR \leftarrow (MBR) +1
t4: Memori \leftarrow (MBR)

Jika $((MBR) = 0)$ maka $(PC \leftarrow (PC) + I)$

ISZ adalah opcode yang memerlukan keputusan bersyarat. Register PC bertambah jika $(MBR) = 0$. Tes dan tindakan ini dapat diimplementasikan sebagai satu operasi mikro. Perhatikan juga bahwa operasi mikro ini dapat dilakukan selama waktu yang sama di mana nilai yang diperbarui dalam MBR disimpan kembali ke memori.

Contoh 3:

BSA X

Alamat instruksi yang mengikuti instruksi BSA disimpan di lokasi X, dan eksekusi berlanjut di lokasi X+I. Alamat yang disimpan nantinya akan digunakan untuk kembali. Ini adalah teknik sederhana untuk mendukung panggilan subrutin. Operasi mikro berikut sudah cukup:

t1: MAR \leftarrow (IR (alamat))
MBR \leftarrow (PC)
t2: PC \leftarrow (IR (alamat))
Memori d (MBR)
t3: PC \leftarrow (PC) + I

Alamat di PC pada awal instruksi adalah alamat instruksi selanjutnya secara berurutan. Ini disimpan di alamat yang ditentukan dalam IR. Alamat terakhir juga bertambah untuk memberikan alamat instruksi untuk siklus instruksi berikutnya.

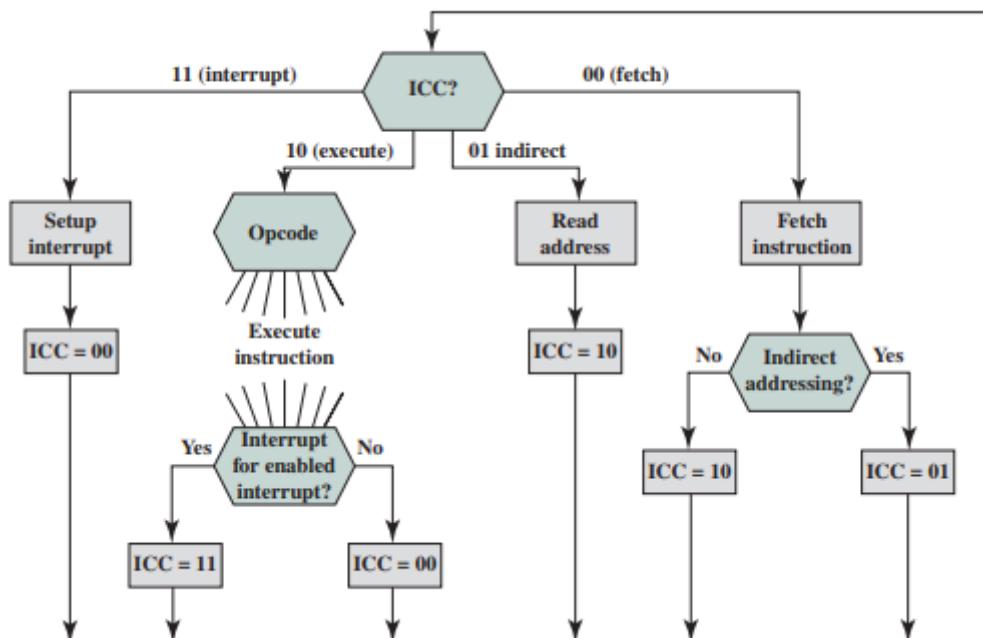
E. Instruction Cycle (Siklus Instruksi)

Setiap fase dari siklus instruksi dapat didekomposisi menjadi suatu urutan operasi mikro elementer. Dalam contoh, ada satu urutan masing-masing untuk mengambil, tidak langsung, dan mengganggu siklus, dan, untuk siklus eksekusi, ada satu urutan operasi mikro untuk setiap opcode.

Gambar 11.3 adalah *flowchart* untuk siklus eksekusi. Siklus ini, melibatkan register 2-bit baru yang disebut *Instruction Code Cycle* (ICC). ICC menentukan status prosesor dari salah satu empat kondisi berikut:

- 00: Ambil (*fetch*)
- 01: Tidak Langsung (*indirect*)
- 10: Eksekusi (*execute*)
- 11: Interupsi (*interrupt*)

Siklus tidak langsung selalu diikuti oleh siklus eksekusi. Siklus interupsi selalu diikuti oleh siklus pengambilan. Untuk siklus pengambilan dan eksekusi, siklus berikutnya tergantung pada kondisi sistem. Dengan demikian, diagram alir Gambar 11.3 mendefinisikan urutan lengkap operasi mikro, hanya bergantung pada urutan instruksi dan pola interupsi.



Sumber: (Stallings, 2016)

Gambar 11.3. *Flowchart* pada siklus insktursi

11.2.Organisasi Internal Prosesor

Elemen fungsional dasar prosesor adalah sebagai berikut:

- ALU
- Register
- Jalur data internal

- Jalur data eksternal
- *Control Unit*

ALU adalah esensi fungsional komputer. Register digunakan untuk menyimpan data internal ke prosesor. Beberapa register berisi informasi status yang diperlukan untuk mengelola urutan instruksi (misalnya: *Program Status Word*). Lainnya berisi data yang masuk atau berasal dari modul ALU, memori, dan I/O. Jalur data internal digunakan untuk memindahkan data antara register, atau antara register dan ALU. Jalur data eksternal menghubungkan register ke memori dan modul I/O, sering melalui bus sistem. *Control Unit* menyebabkan operasi terjadi di dalam prosesor. Eksekusi suatu program terdiri dari operasi yang melibatkan elemen-elemen prosesor ini.

Operasi yang dilakukan prosesor terdiri dari serangkaian operasi mikro dalam salah satu kategori berikut:

- Mentransfer data dari satu register ke register lainnya.
- Mentransfer data dari register ke antarmuka eksternal (misalnya: Bus sistem).
- Transfer data dari antarmuka eksternal ke register.
- Melakukan operasi aritmatika atau logika, menggunakan register untuk *input* dan *output*.

Semua operasi mikro yang diperlukan untuk melakukan satu siklus instruksi, termasuk dalam salah satu kategori tersebut.

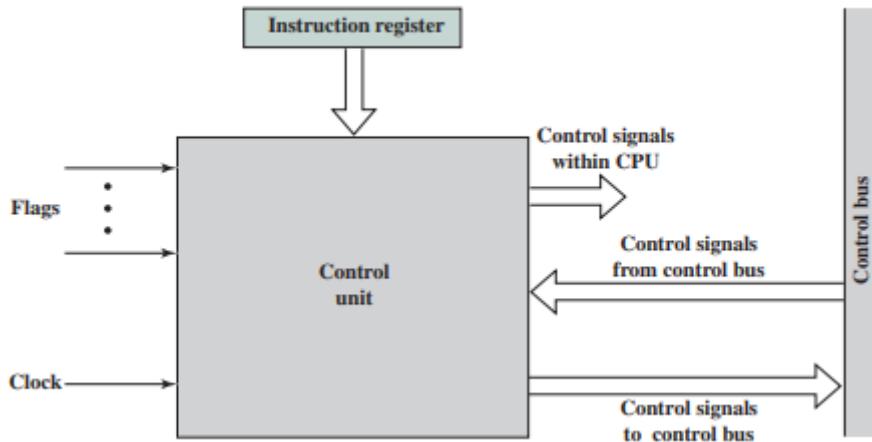
Pada dasarnya, *Control Unit* melakukan dua tugas dasar berikut:

1. *Sequencing*: *Control Unit* menyebabkan prosesor melewati serangkaian operasi mikro dalam urutan yang benar, berdasarkan pada program yang sedang dijalankan.
2. *Eksekusi*: *Control Unit* menyebabkan setiap operasi mikro dilakukan.

Agar *Control Unit* dapat menjalankan fungsinya dengan benar, maka harus memiliki *input* yang dapat menentukan status sistem, dan *output* yang dapat mengontrol perilaku sistem. Ini adalah spesifikasi eksternal dari *Control Unit*. Secara internal, *Control Unit* harus memiliki logika yang diperlukan untuk melakukan fungsi sekuensing dan eksekusi. Gambar 11.4 adalah model umum *Control Unit*, yang menunjukkan semua *input* dan *output*nya.

Input yang diperlukan oleh *Control Unit* adalah:

1. *Clock*: *Control Unit* menyebabkan satu atau serangkaian operasi mikro, yang secara simultan dilakukan untuk setiap pulsa *clock*. Ini kadang-kadang disebut sebagai waktu siklus prosesor, atau waktu siklus *clock*.
2. Register instruksi: Mode opcode dan pengalamatan dari instruksi saat ini digunakan untuk menentukan operasi mikro mana yang harus dilakukan selama siklus eksekusi.
3. Bendera (*flag*): Ini diperlukan oleh *Control Unit* untuk menentukan status prosesor dan hasil operasi ALU sebelumnya. Misalnya, untuk instruksi ISZ: *Control Unit* akan menambah PC jika bendera nol diatur.
4. Sinyal kontrol dari bus kontrol (*control signals from control bus*): Bagian bus kontrol dari bus sistem memberikan sinyal ke *Control Unit*.
5. Mengontrol sinyal dalam prosesor (*control signals within CPU*): Ini adalah dua jenis: yang menyebabkan data dipindahkan dari satu register ke register lainnya, dan yang mengaktifkan fungsi ALU tertentu.
6. Mengontrol sinyal ke bus kontrol (*control signals to control bus*): Ini juga terdiri dari dua jenis: sinyal kontrol ke memori, dan sinyal kontrol ke modul I/O.



Sumber: (Stallings, 2016)

Gambar 11.4. Blok Diagram *Control Unit*

Tiga jenis sinyal kontrol digunakan untuk: mengaktifkan fungsi ALU; mengaktifkan jalur data; dan merupakan sinyal pada bus sistem eksternal atau antarmuka eksternal lainnya. Semua sinyal ini pada akhirnya diterapkan secara langsung sebagai *input* biner ke gerbang logika individu.

Control Unit melacak di mana ia berada dalam siklus instruksi. Pada titik tertentu, ia tahu bahwa siklus pengambilan akan dilakukan selanjutnya. Langkah pertama pada siklus pengambilan adalah mentransfer konten PC ke MAR. *Control Unit* melakukan ini dengan mengaktifkan sinyal kontrol yang membuka gerbang antara bit PC dan bit MAR. Langkah selanjutnya adalah membaca *word* dari memori ke dalam MBR dan menambah PC. *Control Unit* melakukan ini dengan mengirimkan sinyal kontrol berikut secara bersamaan:

- Sinyal kontrol yang membuka gerbang, memungkinkan konten MAR ke bus alamat;
- Sinyal kontrol baca memori pada bus kontrol;
- Sinyal kontrol yang membuka gerbang, memungkinkan konten bus data disimpan di MBR;
- Kontrol sinyal ke logika yang menambahkan 1 ke isi PC dan menyimpan hasilnya kembali ke PC.

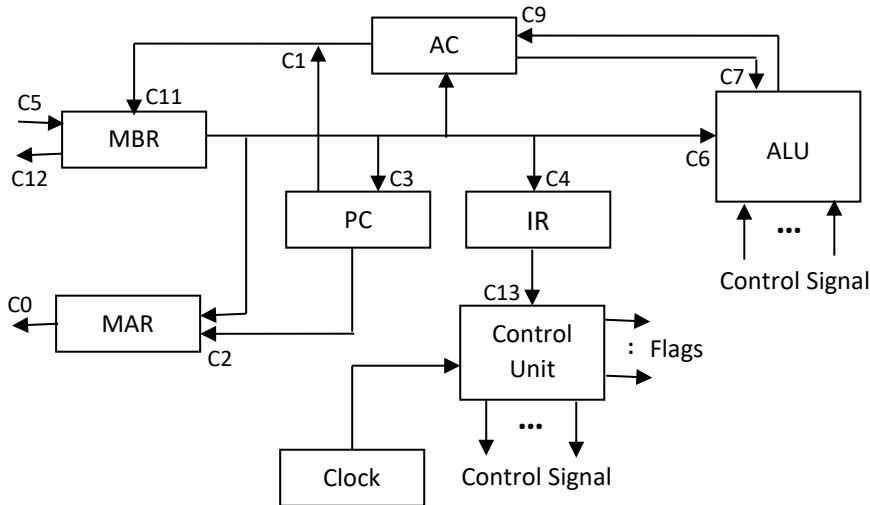
Setelah ini, *Control Unit* mengirimkan sinyal kontrol yang membuka gerbang antara MBR dan IR. Ini melengkapi siklus pengambilan kecuali untuk satu hal: *Control Unit* harus memutuskan apakah akan melakukan siklus tidak langsung atau siklus eksekusi berikutnya. Untuk memutuskan ini, ia memeriksa IR untuk melihat apakah referensi memori tidak langsung dibuat.

Siklus tidak langsung dan interupsi memiliki kejadian yang sama. Untuk siklus eksekusi, *Control Unit* dimulai dengan memeriksa opcode dan, atas dasar itu, memutuskan urutan operasi mikro yang akan dijalankan untuk siklus eksekusi.

Gambar 11.5 adalah contoh sederhana untuk mengilustrasikan fungsi unit kendali. Ini adalah prosesor sederhana dengan akumulator tunggal (AC). Jalur data antar elemen ditunjukkan. Jalur kontrol untuk sinyal yang berasal dari unit kontrol tidak ditampilkan, tetapi penghentian sinyal kontrol diberi label Ci dan ditandai dengan panah. Unit kontrol menerima input dari *clock*, IR, dan flag. Dengan setiap siklus *clock*, unit kontrol membaca semua inputnya dan memancarkan satu set sinyal kontrol. Sinyal kontrol pergi ke tiga tujuan terpisah, yaitu:

1. Jalur data: Unit kontrol mengontrol aliran data internal. Misalnya, pada pengambilan instruksi, isi register penyanga memori ditransfer ke IR. Untuk setiap jalur yang akan dikontrol, ada sakelar (ditunjukkan dengan lingkaran pada gambar). Sinyal kontrol dari unit kontrol untuk sementara membuka gerbang agar data bisa lewat.

2. ALU: Unit kendali mengontrol pengoperasian ALU dengan serangkaian sinyal kendali. Sinyal ini mengaktifkan berbagai sirkuit logika dan gerbang di dalam ALU.
3. Bus sistem: Unit kontrol mengirimkan sinyal kontrol ke jalur kontrol bus sistem (misalnya: Memori READ).



Sumber: (Stallings, 2016)

Gambar 11.5. Jalur Data dan Sinyal Kontrol

Unit kontrol harus menjaga informasi tentang posisinya dalam siklus instruksi. Dengan menggunakan informasi ini, dan dengan membaca semua inputnya, unit kontrol memancarkan urutan sinyal kontrol yang menyebabkan terjadinya operasi mikro. Ini menggunakan pulsa *clock* untuk mengatur waktu urutan kejadian, sehingga memungkinkan waktu antara kejadian untuk level sinyal menjadi stabil.

Tabel 11.1. *Micro Operation* dan *Control Signal*.

Instruction Cycle	Micro Operation	Active Control Signal
Fetch	$t_1: MAR \leftarrow (PC)$	C2
	$t_2: MBR \leftarrow \text{Memory}$ $PC \leftarrow (PC) + 1$	C5, CR
	$t_3: IR \leftarrow (MBR)$	C4
Indirect	$t_1: MAR \leftarrow (IR(\text{Address}))$	C8
	$t_2: MBR \leftarrow \text{Memory}$	C5, CR
	$t_3: IR(\text{Address}) \leftarrow (MBR(\text{Address}))$	C4
Interrupt	$t_1: MAR \leftarrow (PC)$	C1
	$t_2: MAR \leftarrow \text{Save address}$ $PC \leftarrow \text{Routing address}$	
	$t_3: \text{Memory} \leftarrow MAR$	C12, CW

CR = Read Control Signal ke bus system

CW = Write Control Signal ke bus system

Sumber: (Stallings, 2016)

Unit kontrol (control unit) adalah mesin yang menjalankan seluruh komputer. Ini dilakukan hanya berdasarkan mengetahui instruksi yang akan dieksekusi dan sifat hasil operasi aritmatika dan logika. Ia tidak pernah bisa mengetahui data yang sedang diproses atau hasil aktual yang dihasilkan. Dan unit kontrol mengendalikan semuanya dengan beberapa sinyal kontrol ke titik-titik di dalam prosesor dan beberapa sinyal kontrol ke bus sistem. Tabel 11.1 menunjukkan sinyal kontrol yang diperlukan untuk

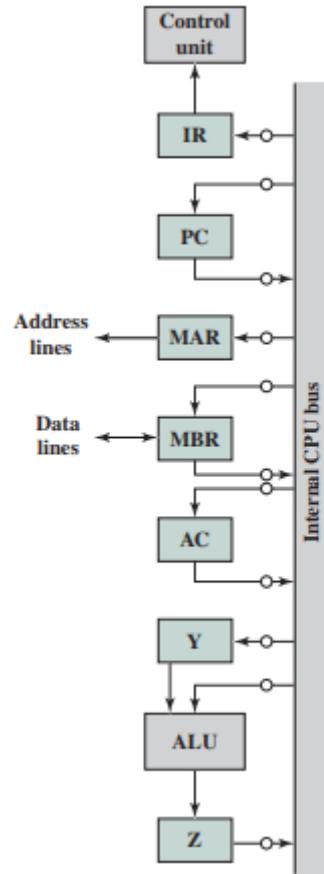
beberapa urutan operasi mikro yang dijelaskan sebelumnya. Untuk kesederhanaan, data dan jalur kontrol untuk menaikkan PC dan untuk memuat alamat tetap ke PC dan MAR tidak ditampilkan.

11.3.Organisasi Internal Prosesor

Gambar 11.6 menunjukkan penggunaan berbagai jalur data pada prosesor. Satu bus internal menghubungkan ALU dan semua register prosesor. Gerbang dan sinyal kontrol disediakan untuk perpindahan data ke dan dari bus dari setiap register. Sinyal kontrol tambahan mengontrol transfer data ke dan dari bus sistem (eksternal) dan pengoperasian ALU. Dua register baru, berlabel Y dan Z, telah ditambahkan ke organisasi. Ini diperlukan untuk operasi ALU yang melibatkan dua *operand*, satu dapat diperoleh dari bus internal, tetapi yang lain harus diperoleh dari sumber lain (register AC dapat digunakan untuk tujuan ini) tetapi ini membatasi fleksibilitas sistem dan tidak akan berfungsi dengan prosesor dengan banyak register tujuan umum. Register Y menyediakan penyimpanan sementara untuk *input* lainnya. ALU adalah sirkuit kombinatorial tanpa penyimpanan internal. Jadi, ketika sinyal kontrol mengaktifkan fungsi ALU, *input* ke ALU ditransformasikan ke *output*. Oleh karena itu, *output* dari ALU tidak dapat langsung dihubungkan ke bus, karena *output* ini akan memberi umpan balik ke *input*. Register Z menyediakan penyimpanan *output* sementara. Dengan pengaturan ini, operasi untuk menambah nilai dari memori ke AC akan memiliki langkah-langkah berikut:

- t1: MAR d (IR (alamat))
- t2: MBR d Memori
- t3: Y d (MBR)
- t4: Z d (AC) + (Y)
- t5: AC d (Z)

Organisasi lain dimungkinkan, tetapi, secara umum, beberapa jenis bus internal atau bus internal digunakan. Penggunaan jalur data umum menyederhanakan tata letak interkoneksi dan kontrol prosesor. Alasan praktis lain untuk penggunaan bus internal adalah untuk menghemat ruang.

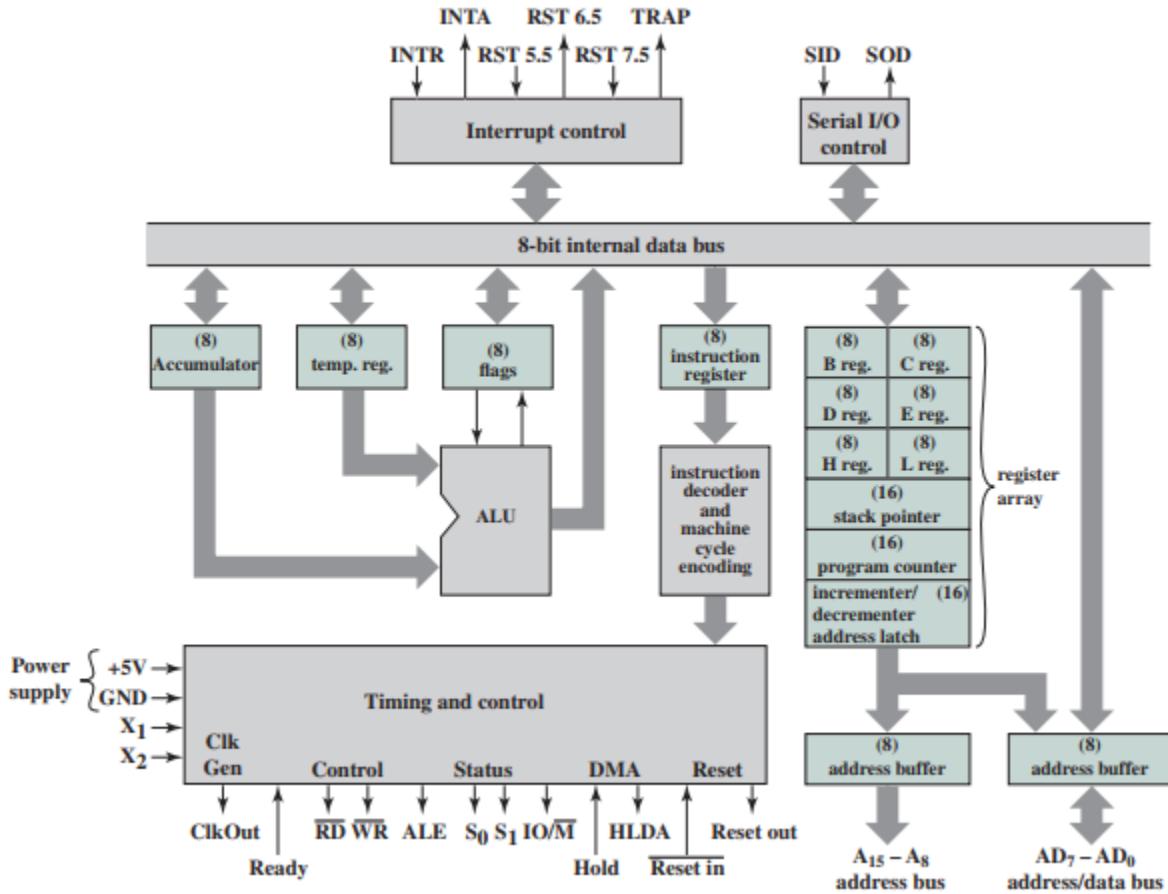


Sumber: (Stallings, 2016)

Gambar 11.6. Prosesor dengan Internal Bus

11.4. Intel 8085

Mikroprosesor Intel 8085 dapat digunakan untuk menganalisa implementasi organisasi prosesor. Intel 8085 merupakan mikroprosesor NMOS 8-bit yang dikemas dalam bentuk paket IC 40 Pin *dual in line*, beroperasi pada tegangan +5 V DC. Kecepatan *clock* yang digunakan pada mikroprosesor ini sekitar 3 MHZ, mampu menangani memori hingga 64 K byte (yaitu $2^{16} = 65536$ byte). Gambar 11.7 adalah blok diagram organisasi prosesor Intel 8085. Komponen fungsional utama mikroprosesor 8085A adalah: bagian Register, Unit Aritmatika dan Logika, Bagian Pengaturan Waktu (timing and control), Kontrol Interupsi (interrupt control), dan Serial I/O control.



Sumber: (Brey, 2009)

Gambar 11.7. Blok Diagram Prosesor Intel 8085

Mikroprosesor 8085 memiliki delapan register 8-bit yang dapat dialamatkan, yaitu: A (register Akumulator), F (Flag register), B, C, D, E, H , dan L. Dari register ini register B, C, D, E, H dan L adalah register tujuan umum 8-bit. Register-register ini dapat digunakan sebagai register tunggal atau kombinasi dari dua register sebagai pasangan register 16 bit. Pasangan register yang valid adalah pasangan register B-C, D-E atau H-L. Byte orde tinggi dari data 16 bit disimpan di register pertama (B dalam pasangan register B-C), dan byte orde rendah di register kedua (C dalam pasangan register B-C). Pasangan register H-L juga dapat digunakan untuk pengalamanan indirect register, karena pasangan register ini juga bisa berfungsi sebagai penunjuk data. Selain register tujuan umum ini, 8085 memiliki dua register 8-bit Accumulator (A) dan Flag (F) sebagai register tujuan khusus dan dua register 16 bit yaitu Program counter (PC) dan stack pointer (SP).

Acumulator (A) adalah register buffer 8 bit yang banyak digunakan dalam operasi aritmatika, logika, load, dan store serta dalam instruksi input/output. Semua operasi aritmatika dan logika dilakukan pada isi akumulator, yaitu salah satu *operand* selalu dibawa ke akumulator.

Flag (F) adalah register 8-bit yang terkait dengan pelaksanaan instruksi di mikroprosesor. Dari 8 bit register flag, 5 bit berisi informasi penting dalam hal satus flag. Lima bendera tersebut adalah:

- 1) *Sign flag* (S). Set (S = 1), jika hasil operasi instruksi negatif (MSB hasilnya 1); reset (S = 0) untuk hasil positif (MSB adalah nol).
- 2) *Zero flag* (Z). Yaitu Z = 1 jika hasilnya nol, dan Z = 0 jika hasilnya bukan nol.
- 3) *Carry flag* (CY). Bendera carry diset ke 1, jika ada *carry* (atau *borrow*) ke bit order tertinggi (posisi 9 tidak ada) sebagai hasil dari eksekusi instruksi penambahan atau pengurangan. Jika tidak ada *carry* (atau *borrow*) ke bit order yang lebih tinggi, *carry flag* di-reset ke 0.

- 4) *Parity flag* (P). Setelah operasi aritmatika dan logika, jika hasilnya memiliki bit 1 sebanyak bilangan genap, maka bit paritas diset 1. Jika di sisi lain jumlah bit 1 adalah ganjil, bendera paritas di-reset 0 (pada mode paritas ganjil).
- 5) *Auxiliary Carry flag* (AC). Bendera ini (AC) disetel ke 1, jika ada *overflow* di bit 3 akumulator. Bendera AC digunakan dalam aritmatika BCD. Ini diilustrasikan sebagai pada penjumlahan A6h+5Fh berikut

$$\begin{array}{r}
 1010 \quad 0110 \\
 0101 \quad 1111 + \\
 \hline
 1\ 0000 \ 1\ 1111 \\
 \uparrow \qquad \uparrow \\
 \text{CY} \qquad \text{AC}
 \end{array}$$

Program Counter (PC) adalah register 16 bit yang digunakan untuk mengirim alamat 16 bit pada pengambilan instruksi dari memori. Ini bertindak sebagai pointer yang menunjukkan alamat instruksi berikutnya yang akan diambil dan dijalankan. *Program Counter* diperbarui setelah instruksi diambil oleh prosesor. Jika instruksi adalah instruksi satu byte, maka *Program Counter* akan diperbarui oleh satu (yaitu $\text{PC} = \text{PC} + 1$). Demikian pula, untuk instruksi dua dan tiga byte, *Program Counter* akan diperbarui oleh dua (yaitu $\text{PC} = \text{PC} + 2$) atau tiga (yaitu $\text{PC} = \text{PC} + 3$) sesuai panjang lokasi masing-masing instruksi yang diambil.

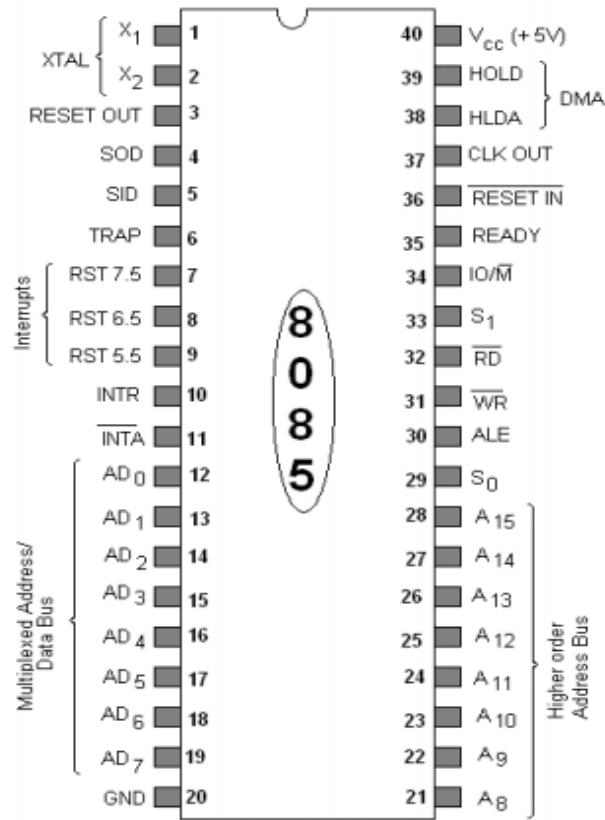
Stack adalah area memori di mana informasi sementara disimpan dengan metode *Last-In-First-Out* (LIFO). Alamat di area stack ditetapkan oleh stack pointer di mana informasi pertama disimpan sebagai entri tumpukan pertama. Ini dilakukan dengan menginisialisasi stack pointer dengan sebuah instruksi. Entri tumpukan yang lebih tinggi dibuat di alamat yang semakin menurun.

Timing and Control. Inti dari unit kontrol adalah modul pengaturan waktu dan modul kontrol (*timing and control*). Modul ini mencakup *clock* dan menerima input akibat instruksi saat ini dan beberapa sinyal kontrol eksternal. Outputnya terdiri dari sinyal kontrol ke komponen lain dari prosesor ditambah sinyal kontrol ke bus sistem eksternal.

Waktu operasi prosesor disinkronkan oleh *clock* dan dikontrol oleh *Control Unit* melalui sinyal kontrol. Setiap siklus instruksi dibagi menjadi satu hingga lima siklus mesin; setiap siklus mesin secara bergantian dibagi menjadi tiga hingga lima state. Setiap state berlangsung selama satu siklus *clock*. Selama state, prosesor melakukan satu atau satu set operasi mikro simultan sebagaimana ditentukan oleh sinyal kontrol.

Jumlah siklus mesin ditetapkan untuk instruksi yang diberikan tetapi bervariasi dari satu instruksi ke instruksi lainnya. Siklus mesin didefinisikan setara dengan akses bus. Jadi, jumlah siklus mesin untuk sebuah instruksi bergantung pada berapa kali prosesor harus berkomunikasi dengan perangkat eksternal. Misalnya, jika sebuah instruksi terdiri dari dua bagian 8-bit, maka diperlukan dua siklus mesin untuk mengambil instruksi tersebut. Jika instruksi tersebut melibatkan memori 1-byte atau operasi I/O, maka siklus mesin ketiga diperlukan untuk eksekusi.

Gambar 11.8 menunjukkan 40 pin pada IC mikroprosesor Intel 8085 secara lengkap.



Sumber (Kaushik, 2010)

Gambar 11.8. Pin pada IC Mikroprosesor Intel 8085

Tabel 11.2 menjelaskan sinyal eksternal yang masuk dan keluar dari 8085. Ini terkait dengan bus sistem eksternal. Sinyal ini adalah antarmuka antara prosesor 8085 dan sistem lainnya.

Tabel 11.2. Sinyal Eksternal pada Intel 8085

A. Address and Data Signals																																			
1	High Address (A15-A8)	8 bit tertinggi dari 16 bit alamat																																	
2	Address/Data (AD7-AD0)	Adalah sinyal yang dimultipleks, untuk 8 bit alamat terendah dari 16 bit alamat atau Data 8bit																																	
3	Serial Input Data (SID)	Satu bit input data untuk mengakomodir perangkat eksternal dengan transmisi serial (satu bit pada satu waktu)																																	
4	Serial Output Data (SOD)	Satu bit output data untuk mengakomodir perangkat eksternal dengan transmisi serial																																	
B. Timing and Control Signal																																			
1	CLK (OUT)	<i>Clock</i> sistem. Sinyal CLK menuju chip perangkat untuk sinkronisasi timing																																	
2	X1, X2	Sinyal ini datang dari Xtal untuk mengendalikan internal <i>clock</i> generator																																	
3	Address Latch Enable (ALE)	Digunakan selama <i>clock</i> state dari siklus mesin dan menyebabkan memori dan perangkat untuk dapat mengenali alamat lengkap 16 bit																																	
4	Status (S0 dan S1)	Control signal yang digunakan untuk mengindikasikan operasi read atau write. Bersama sinyal IO/M, maka akan menentukan siklus mesin.																																	
		<table border="1"> <thead> <tr> <th>Machine Cycle</th><th>IO/M</th><th>S1</th><th>S0</th></tr> </thead> <tbody> <tr> <td>fetch Cycle</td><td>0</td><td>1</td><td>1</td></tr> <tr> <td>Memory Read Cycle</td><td>0</td><td>1</td><td>0</td></tr> <tr> <td>Memory Write Cycle</td><td>0</td><td>0</td><td>1</td></tr> <tr> <td>I/O Read Cycle</td><td>1</td><td>1</td><td>0</td></tr> <tr> <td>I/O Write Cycle</td><td>1</td><td>0</td><td>1</td></tr> <tr> <td>INTR Acknowledge</td><td>1</td><td>1</td><td>1</td></tr> <tr> <td>Halt</td><td>Hi-Z</td><td>0</td><td>0</td></tr> </tbody> </table>	Machine Cycle	IO/M	S1	S0	fetch Cycle	0	1	1	Memory Read Cycle	0	1	0	Memory Write Cycle	0	0	1	I/O Read Cycle	1	1	0	I/O Write Cycle	1	0	1	INTR Acknowledge	1	1	1	Halt	Hi-Z	0	0	
Machine Cycle	IO/M	S1	S0																																
fetch Cycle	0	1	1																																
Memory Read Cycle	0	1	0																																
Memory Write Cycle	0	0	1																																
I/O Read Cycle	1	1	0																																
I/O Write Cycle	1	0	1																																
INTR Acknowledge	1	1	1																																
Halt	Hi-Z	0	0																																
5	IO/M	Digunakan untuk meng-enable Memori atau modul I/O yang akan dilakukan operasi read atau write																																	
6	Read Control (RD)	Mengindikasikan memori atau modul I/O untuk dilakukan operasi read, dan bus data tersedia untuk transfer data																																	
7	Write Control (WR)	Mengindikasikan data pada bus data yang akan digunakan diteredia untuk dituliskan pada memori atau modul I/O																																	
C. Simbol inisialisasi Memori dan I/O																																			
1	Hold	Meminta CPU untuk melepaskan kontrol dan penggunaan bus sistem eksternal. CPU akan menyelesaikan eksekusi instruksi yang ada di IR dan kemudian memasuki status hold, di mana tidak ada sinyal yang dimasukkan oleh CPU ke bus kontrol, alamat, atau data. Selama status tunggu, bus dapat digunakan untuk operasi DMA																																	
2	Hold Acknoledgement (HLDA)	Sinyal keluaran unit kontrol ini mengenali sinyal HOLD dan menunjukkan bahwa bus sekarang tersedia.																																	
3	READY	Digunakan untuk menyinkronkan CPU dengan memori yang lebih lambat atau perangkat I/O. Ketika perangkat yang dialamatkan menyatakan READY, CPU dapat melanjutkan dengan operasi input atau output. Jika tidak, CPU memasuki status WAIT hingga perangkat siap.																																	
D. Signal yang berhubungan dengan Interupsi																																			
1	TRAP	Restart Interrupts (RST 7.5, 6.5, 5.5)																																	
2	Interrupt Request (INTR)	Kelima baris ini digunakan oleh perangkat eksternal untuk menginterupsi CPU. CPU tidak akan menerima permintaan jika dalam status tahan atau jika interupsi dinonaktifkan.																																	
3	Interrupt Acknowledge (ACK)	Pengakuan permintaan interupsi																																	
4	RESET IN	Menyebabkan isi register PC disetel ke nol. CPU melanjutkan eksekusi di lokasi nol																																	

5	RESET OUT	Mengakui bahwa CPU telah disetel ulang. Sinyal dapat digunakan untuk mengatur ulang sisa sistem.
D. Voltage dan Ground		
1	VCC	+5-volt power supply
2	VSS	Electrical ground

Sumber (Kaushik, 2010)

Daftar Pustaka

- Brey, B. B. (2009). *The Intel microprocessors : 8086/8088, 80186/80188, 80286, 80386, 80486, Pentium, Pentium Pro processor, Pentium II, Pentium III, Pentium 4, and Core2 with 64-bit extensions : architecture, programming, and interfacing*. Person Prentice Hall.
- Kaushik, D. K. (2010). *An Introduction to Microprocessor 8085* (Issue January 2010). Dhanpat Rai Publishing Company (p) Ltd.
- Patterson, D. A., & Hennessy, J. L. (2014). *In Praise of Computer Organization and Design : The Hardware/Software Interface , Fifth Edition*.
- Stallings, W. (2016). *Computer Organization and Architecture Designing for Performance* (10th ed.). Pearson.

INDEX

8

8085, 236, 237

A

Accumulator, 237

arbitrary numbers, 142

Arbitrase bus, 97

Arithmatic and Logic Unit, 7

Arithmetic and Logic Unit, 3, 248

ARM, 10, 23

Arsitektur Harvard, 4

arsitektur mezzanine, 115

Arsitektur Princeton, 3

Arsitektur von Newmann, 3

ASCII, 188

Asosiatif access, 42

Assembler, 211

Assembly Language, 211

Assosiative Mapping, 53

Auxiliary Carry flag, 238

AVR, 10

B

bit paritas, 66

block compare, 67

Bus, 112

C

cache, 43

cache memory, 5

Carry flag, 237

Central Processing Unit, 3, 248

Charles Babbage, 3

Column Address Strobe, 63

Compiler, 211

constant angular velocity, 69

Control, 2, 248

control unit, 234

Control Unit, 3, 248

Core, 5

CPU Interconnection, 3, 248

D

Data movement, 1

Data processing, 1, 248

Data storage, 1, 248

Deeply Embedded Devices, 8

Direct access, 42

direct addressing, 202

Direct Mapping, 49

Direct Memory Access (DMA), 104

Disk, 68

DMA Aknowledge, 106

DMA request, 106

DMAC 8237A, 106

double precision, 159

DRAM, 63

E

EDVAC, 11, 12

EEPROM, 62

effective address, 202

embedded system, 7

ENIAC, 11

EPROM, 61

Error Correction Code, 66

Exa Byte, 41

Executable Code, 211

execute cycle, 27

Extended ASCII, 189

extended single precision, 159

F

feromagnetik, 21

fetch cycle, 27, 228

fix-point, 155

Flag, 237

flash memory, 62

floating-point, 156

G

gate, 17

Giga Byte, 41

H

hard failure, 65

head, 68

Hold Acknowledge, 106

Hold Request, 106

Human readable, 86

I

I/O, 3, 248

I/O channel, 109

IEEE 754, 158

IEEE 754-2008, 158

immediate addressing, 202

indirect addressing, 203

Instruction Code Cycle, 231

Instruction Logic, 7

Intel 4004, 22

Intel 8008, 22

Intel 80386, 22

Interconnect Component Peripheral, 124

Internet of things, 8

interrupt driven I/O, 95

isolated I/O, 91

K

Kilo Byte, 41

Komputer, 1

Komputer IAS, 12

L

latensi, 42

Least Frequently Used, 59

Least Recently Used, 59

level cache, 5

Load/Storage Logic, 7

Loader, 211

lokalitas referensi, 43

LSI, 21

M

magnetik disk, 68

Magnetoresistive Read, 69

Main Memory, 3, 248

MCS, 10

Mechine readable, 86

Mega Byte, 41

Memory Mangement Unit, 47

memory-mapped I/O, 91

micro operation, 227

Mikrokontroler, 9

Mikroprosesor, 9

mnemonic, 15

Moore, 19

Motherboard, 5

multicore, 5

multiple interrupt line, 97

Multiple Zone Recording, 70

multiplexor, 109

N

nonvolatile, 43

O

Object Code, 211

Operand, 224

P

package, 18

paritas ganjil, 189

paritas genap, 189

Parity flag, 238

PCI, 126

PCIe, 133

Peta Byte, 41

PIC, 10

PIC 82C59A, 99

PPI 8255A, 100

Program Counter, 238

Program Status Word, 96

programmable interrupt controller, 99

programmed I/O, 93

PROM, 61

R

radix, 146

RAID, 75

RAID 0, 78

RAID 1, 79

RAID 2, 80

RAID 3, 80

RAID 4, 81

RAID 5, 82

RAM, 63

Random access, 42

Rantai daisy, 97

Read Only Memory, 60

refresh, 63

Register, 3, 248

regisiter addreesing, 203

representasi bias, 156

RISC, 220

Row Address Strobe, 63

S

SCSI, 115

selector, 109

Sequential access, 42

Set-Associative Mapping, 55

shared bus, 116

sign bit, 144

Sign flag, 237

signed-magnitude, 143

single precision, 159

Smal Scale Integration, 19

Soft error, 65

software poll, 97

Solid State Drive, 83

solid-state, 16

SRAM, 64

Stack, 206, 238

stored-program concept, 11

struktur interkoneksi, 111

substrate, 68

syndrome word, 67

System Interconnection, 3, 248

T

Tag identifier, 49

Tera Byte, 41

Timing and Control, 238

Timing Seek, 73

track, 69

transfer rate, 42

Transfer Time, 74

transistor, 16

Two's Complement, 144

unsigned integer, 144

V

VLSI, 21

volatile, 43, 63

von Neumann, 3

W

wafer, 18

Write-through cache, 59

Y

Yotta Byte, 41

U

ULSI, 21

unicode, 189

Z

Zero flag, 237

Zetta Byte, 41

GLOSARIUM

<i>Central Processing Unit (CPU)</i>	Unit yang mengendalikan operasi komputer dan membentuk fungsi pemrosesan data. CPU sering disebut sebagai prosesor
Arsitektur Harvard	Arsitektur komputer yang memisahkan jalur bus antara memory dan I/O
Arsitektur von Neumann	dikenal juga sebagai Arsitektur Princeton, adalah Arsitektur komputer pertama yang menerapkan konsep stored programm
Bit	singkatan dari binary digit, merupakan unsur terkecil dari data komputer
<i>cache memory</i>	memori berlapis-lapis antara processor dan <i>main memory</i>
<i>Complex Instruction Set Computing (CISC)</i>	Rancangan komputer dengan jumlah instruksi yang sangat banyak
Core	pemrosesan individual (terdiri dari Control Unit, ALU, register, dan mungkin cache) pada <i>chip</i> prosesor
<i>embedded system</i>	mengacu pada penggunaan elektronik dan perangkat lunak dalam suatu produk tertentu
Hukum Moore	Hukum yang memprediksi bahwa jumlah transistor yang dapat dimasukkan ke dalam satu <i>chip</i> meningkat dua kali lipat setiap tahun
I/O	Unit atau modul yang befungsi memindahkan data antara komputer dengan perangkat luar
<i>Internet of things (IoT)</i>	istilah yang mengacu pada perluasan interkoneksi perangkat pintar ke internet, mulai dari <i>devices</i> hingga sensor kecil
<i>Main Memory</i>	memori kerja yang digunakan untuk menyimpan data
memori <i>non-volatile</i>	memori yang dapat menyimpan data secara permanen
memori <i>volatile</i>	memori yang bersifat tidak permanen, tergantung pada supply listrik
Mikrokontroler	Mikrokontroler adalah sebuah <i>chip</i> yang didalamnya terdapat sebuah mikroprosesor yang telah dilengkapi dengan RAM, ROM, I/O Port, Timer dan Serial COM dalam satu paket
Mikroprosesor	istilah yang digunakan untuk prosesor pertama yang dibuat oleh manusia, dengan kemampuan yang sangat terbatas dibandingkan dengan prosesor modern
<i>mnemonic</i>	Penulisan instruksi dalam bentuk simbolik bertujuan agar mudah dibaca dan diingat
<i>multicore</i>	Prosesor adalah komponen komputer yang menginterpretasikan dan menjalankan instruksi, terbuat dari sepotong fisik silikon yang mengandung satu atau lebih core. Jika sebuah prosesor mengandung banyak core, itu disebut sebagai prosesor multicore
Program Counter (PC) Register	Register yang menyimpan alamat instruksi berikutnya yang akan diambil atau dijemput (fetch)
<i>Reduced Instruction Set Computing (RISC).</i>	Rancangan komputer dengan jumlah instruksi yang sedikit

<i>Register</i>	Memori pada prosesor yang berukuran kecil dengan kecepatan sangat tinggi
Stack	adalah area memori di mana informasi sementara disimpan dengan metode Last-In-First-Out (LIFO)
<i>System Interconnection</i>	Mekanisme yang digunakan untuk komunikasi antara CPU, memori utama, dan I/O.
Personal Computer	Sistem komputer yang dirancang khusus untuk penggunaan oleh perorangan.

RANGKUMAN

Komputer adalah sebuah sistem yang sangat kompleks, yang terdiri dari hardware softare dan brainware. Pada dewasa ini, sebuah mesin komputer mengandung jutaan komponen elektronik dasar. Sehingga untuk menggambarkan sebuah sistem komputer, maka diawali dengan menganalisis sifat hirarkisnya. Sistem hirarkis adalah sekumpulan subsistem yang saling terkait, dimana masing-masing subsistem tersusun dalam struktur, mulai dari tingkatan tertinggi sampai terendah. Pada setiap tingkat, sistem terdiri dari serangkaian komponen dan keterkaitannya yang dideskripsikan dalam struktur dan fungsi. Struktur adalah bagaimana cara komponen saling terkait, sedangkan fungsi adalah bagaimana pengoperasian setiap komponen individu sebagai bagian dari struktur.

Pada dasarnya struktur dan fungsi komputer terbentuk dari empat fungsi dasar, yaitu: *Data processing, Data storage, Data movement, dan Control*. Komputer tradisional dengan prosesor tunggal, memiliki empat komponen utama pada strukturnya, yaitu: *Central Processing Unit (CPU), Main Memory, I/O, dan System Interconnection*. Dari keempat komponen utama tersebut, CPU adalah komponen yang paling kompleks. Struktur utama dari CPU adalah: *Control Unit (CU), Arithmetic and Logic Unit, Register, dan CPU Interconnection*.

Hampir semua desain komputer kontemporer didasarkan pada konsep yang dikembangkan oleh John von Neumann di Institute for Advanced Studies, Princeton, yang didasarkan pada tiga konsep utama, yaitu: data dan instruksi disimpan dalam satu memori baca-tulis, isi memori dapat dialamatkan berdasarkan lokasi, tanpa memperhatikan jenis data yang terkandung di dalamnya, dan eksekusi terjadi secara berurutan (kecuali diubah secara eksplisit) dari satu instruksi ke instruksi berikutnya.

Pada disain awal, Sistem komputer digital menjalankan program pada satu CPU (prosesor). Hal ini masih berlaku untuk banyak sistem saat ini, terutama yang mementingkan biaya daripada kinerja komputasi yang tinggi. Meskipun banyak peningkatan telah dilakukan selama bertahun-tahun untuk ide aslinya, hampir setiap prosesor yang tersedia saat ini adalah turunan dari arsitektur von Neumann. Komputer modern pada umumnya memiliki banyak prosesor. Ketika semua prosesor ini berada pada satu *chip*, maka disebut sebagai komputer *multicore*, dan setiap unit pemrosesan atau *Central Processing Unit* (terdiri dari *Control Unit, ALU, register, dan mungkin cache*) disebut *core*.

BIOGRAFI



Elly Mufida, M.Kom. Lahir di Jakarta pada tanggal 4 Oktober 1972. Menyelesaikan pendidikan S1 Program Studi Teknik Komputer pada Universitas Guna Dharma, dan menyelesaikan S2 Tahun 2011 pada Program Studi Ilmu Komputer Sekolah Tinggi Manajemen Informatika dan Komputer Nusa Mandiri. Mulai mengajar di AMIK BSI sejak tahun 1996 pada Program Studi Teknik Komputer. Mata Kuliah yang pernah diajarni selama mengajar adalah: Sistem Digital, Elektronika Dasar, Mikroprosesor, Mikrokontroler, Arsitektur Komputer, Matematika Diskrit, Aljabar Linier, dan Sistem Pakar. Mulai 2018, mengajar di Fakultas Teknik dan Informatika Universitas Bina Sarana Informatika, pada Program Studi Teknologi Komputer dan Teknik Elektronika. Tahun 2017-2018 aktif menjadi Tim Editor pada Jurnal Teknik Komputer. Sejak Tahun 2018 sampai sekarang aktif menjadi reviewer pada Jurnal Teknik Komputer dan Jurnal Infortech. Mulai tahun 2018 sampai sekarang aktif pada Badan Penjaminan Mutu Universitas Bina Sarana Informatika sebagai auditor internal dan tim pengolah data.



Henny Leidiyana, M.Kom. Lahir di Jakarta pada tanggal 12 November 1975. Ia merupakan alumnus Jurusan Teknik Informatika Fakultas Teknologi Industri Universitas Persada Indonesia YAI Jakarta. Pada tahun 1998 mengikuti Program Magister Ilmu Komputer dan lulus pada tahun 2011 dari Sekolah Tinggi Manajemen Informatika dan Komputer Nusa Mandiri Jakarta. Pada tahun 2000 diangkat menjadi Dosen Universitas Bina Sarana Informatika dan ditempatkan Program Studi Sistem Informasi Fakultas Teknik dan Informasi.



Eva Rahmawati, M. Kom, lahir di Bogor pada tanggal 15 Desember 1989. Menyelesaikan pendidikan D3 pada Program Studi Manajemen Informatika di AMIK BSI Tangerang lulus pada tahun 2010 lalu melanjutkan pendidikan S1 Program Studi Sistem Informasi di STMIK PGRI Tangerang lulus pada Tahun 2011, dan menyelesaikan S2 tahun 2015 pada Program studi Ilmu Komputer di STMIK Nusa Mandiri. Mulai tahun 2012 mengajar di Program Studi Sistem Informasi Fakultas Teknik dan Informasi Universitas Bina Sarana Informatika. Pada Tahun 2016 menjadi Dosen Tetap di Sekolah Tinggi Manajemen Informatika dan Komputer Nusa Mandiri pada Program Studi Teknik Informatika.



Hylenarti Hertyana, M.Kom, lahir di Jakarta pada tanggal 19 Juni 1985. Telah menyelesaikan pendidikan D3 program studi Manajemen Informatika di AMIK BSI pada tahun 2007. Menyelesaikan pendidikan S1 program studi Sistem Informasi pada tahun 2011 serta menyelesaikan S2 program studi Ilmu Komputer pada tahun 2015 di Sekolah Tinggi Manajemen Informatika dan Komputer Nusa Mandiri Jakarta. Mulai mengajar tahun 2011 di Universitas Bina Sarana Informatika pada program studi Sistem Informasi fakultas Teknik dan Informasi. Pada tahun 2019 menjadi Dosen tetap di Sekolah Tinggi Manajemen Informatika dan Komputer Nusa Mandiri Jakarta pada program studi Sistem Informasi .