Computer Architecture

Fundamentals of Computer Architecture

(From here session 3)

Amir Mahdi Hosseini Monazzah

Room 332,
School of Computer Engineering,
Iran University of Science and Technology,
Tehran, Iran.

monazzah@iust.ac.ir

Spring 2025

*Parts of slides are adopted from prof. David Brooks lectures, Department of Electrical Engineering and Computer Science, Harvard University

Outline

- Instruction Set Architecture (ISA)
  - Definition
  - Different computer architectures
- Basic (Mano's) computer elements
  - Instruction codes
  - Registers
  - Bus
  - Computer instruction
  - Timing and control
  - Instruction cycle
- CISC vs. RISC
  - Classification
  - CISC architecture examples
  - Comparison
  - Destiny
- A simple processor prototype
  - Patterson execution engine
  - MIPS/RISC V

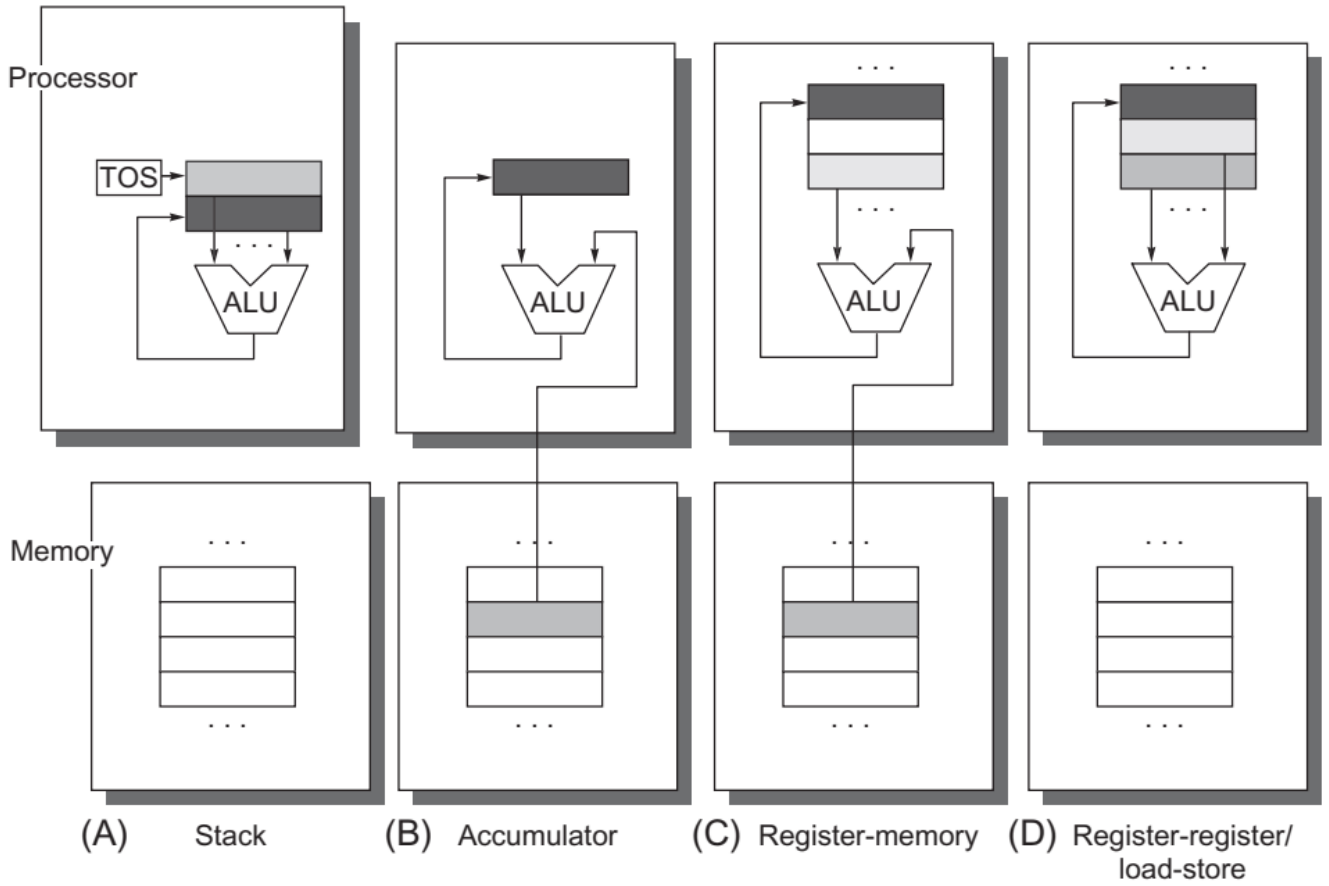
Instruction Set Architecture (ISA)

- "Instruction Set Architecture is the structure of a computer that a machine language programmer (or a compiler) must understand to write a correct (timing independent) program for that machine."
  - IBM, Introducing the IBM 360 (1964)
- The ISA defines:
  - Operations that the processor can execute
  - Data Transfer mechanisms + how to access data
  - Control Mechanisms (branch, jump, etc)
  - "Contract" between programmer/compiler + HW

Classifying ISAs

- The type of internal storage: the most basic differentiation of different ISAs in a processor
- The major choices are
  - Stack
  - An accumulator
  - A set of registers
- Operands may be named explicitly or implicitly
  - The general-purpose register architectures have only explicit operands
    - Either registers or memory locations

Classifying ISAs



Stack

- Architectures with implicit “stack”
  - Acts as source(s) and/or destination, TOS is implicit
  - Push and Pop operations have 1 explicit operand
- Example: C = A + B
  - Push A // S[++TOS] = Mem[A]
  - Push B // S[++TOS] = Mem[B]
  - Add // Term1 = S[TOS--], Term2 = S[TOS--] , S[++TOS] = Term1 + Term2
  - Pop C // Mem[C] = S[TOS--]
- x86 FP uses stack (complicates pipelining)

Accumulator

- Architectures with one implicit register
  - Acts as source and/or destination
  - One other source explicit
- Example: C = A + B
  - Load A // (Acc)umulator <= A
  - Add B // Acc <= Acc + B
  - Store C // C <= Acc
- Accumulator implicit, bottleneck?
- x86 uses accumulator concepts for integer

Register

- Most common approach
  - Fast, temporary storage (small)
  - Explicit operands (register IDs)
- Example: C = A + B
  - Register-memory

    - Load R1, A
    - Add R3, R1, B
    - Store R3, C
    -

  - load/store

    Load R1, A
    Load R2, B
    Add R3, R1, R2
    Store R3, C
- All RISC ISAs are load/store

Register-memory computers
IBM 360,
Intel x86,
Motorolla 68K

Common addressing modes for recent processors

Addressing mode	Example instruction	Meaning	When used
Register	`Add R4,R3`	`Regs[R4]←Regs[R4] +Regs[R3]`	When a value is in a register
Immediate	`Add R4,3`	`Regs[R4]←Regs[R4]+3`	For constants
Displacement	`Add R4,100(R1)`	`Regs[R4]←Regs[R4] +Mem[100+Regs[R1]]`	Accessing local variables (+ simulates register indirect, direct addressing modes)
Register indirect	`Add R4,(R1)`	`Regs[R4]←Regs[R4] +Mem[Regs[R1]]`	Accessing using a pointer or a computed address
Indexed	`Add R3,(R1+R2)`	`Regs[R3]←Regs[R3] +Mem[Regs[R1]+Regs [R2]]`	Sometimes useful in array addressing: R1 = base of array; R2 = index amount
Direct or absolute	`Add R1,(1001)`	`Regs[R1]←Regs[R1] +Mem[1001]`	Sometimes useful for accessing static data; address constant may need to be large
Memory indirect	`Add R1,@(R3)`	`Regs[R1]←Regs[R1] +Mem[Mem[Regs[R3]]]`	If R3 is the address of a pointer *p*, then mode yields **p*
Autoincrement	`Add R1,(R2)+`	`Regs[R1]←Regs[R1] +Mem[Regs[R2]]` `Regs[R2]←Regs[R2]+d`	Useful for stepping through arrays within a loop. R2 points to start of array; each reference increments R2 by size of an element, *d*
Autodecrement	`Add R1, -(R2)`	`Regs[R2]←Regs[R2] - d` `Regs[R1]←Regs[R1] +Mem[Regs[R2]]`	Same use as autoincrement. Autodecrement/-increment can also act as push/pop to implement a stack.
Scaled	`Add R1,100(R2)[R3]`	`Regs[R1]←Regs[R1] +Mem[100+Regs[R2] +Regs[R3] * d]`	Used to index arrays. May be applied to any indexed addressing mode in some computers

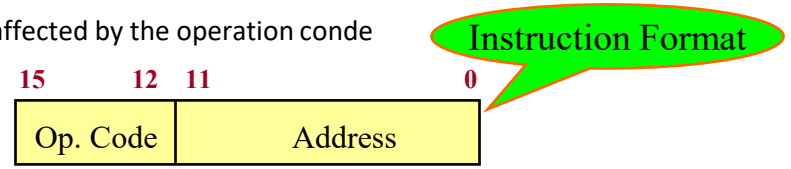
Negotiation with computers? (Up to/from here session 3/4)

- The user of a computer can control the process by means of a ***program***
- A program is a set of ***instructions*** that specify the operations, operand(s) and the sequence (control)
- An instruction is a binary code that specifies a sequence of microoperations
- Instruction codes together with data are stored in memory (= stored program concept)

Instruction Cycle

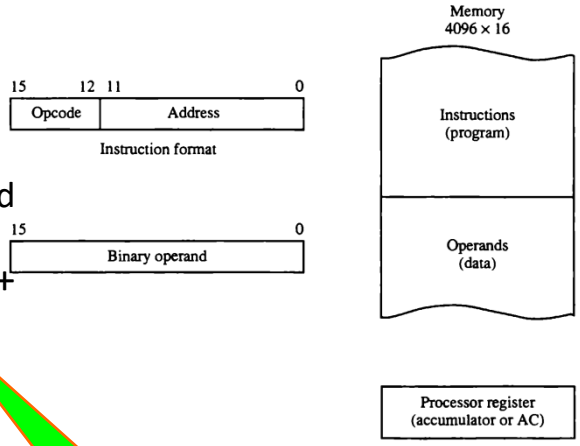
  - The computer reads each instruction from memory and ***places it in a control register***. The control then ***interprets the binary code*** of the instruction and proceeds to ***execute it*** by issuing a sequence of microoperations.

- Instruction code
  - A group of bits that instruct the computer to perform a specific operation
  - It is usually divided into parts
    - Operation code:
      - The most basic part of an instruction code
      - A group of bits that define such operations as add, subtract, multiply, shift, and complement (*bit 12-15 : 2^4 = 16 distinct operations*)
    - Address:
      - Identifies the place(es) that is (are) affected by the operation conde



Negotiation with computers?

- Stored program (accumulator) organization
  - The simplest way to organize a computer
    - One processor register: AC (Accumulator)
      - The operation is performed with the memory operand and the content of AC
    - Instruction code format with two parts: Op. Code + Address
      - Op. Code: specify 16 possible operations (4 bit)
      - Address: specify the address of an operand (12 bit)
      - If an operation in an instruction code does not need an operand from memory, the rest of the bits in the instruction (***address field***) can be used for other purpose (Therefore, use more than 16 instructions: Tab. 5-2, total of 25 instructions)
    - Memory: 12 bit = 4096 word (Instruction and Data are stored)
      - Store each instruction code (***program***) and operand (***data***) in 16-bit memory word

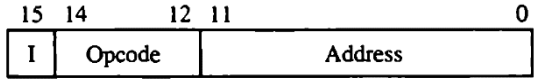


Example: Clear AC, Increment AC, Complement AC, ...

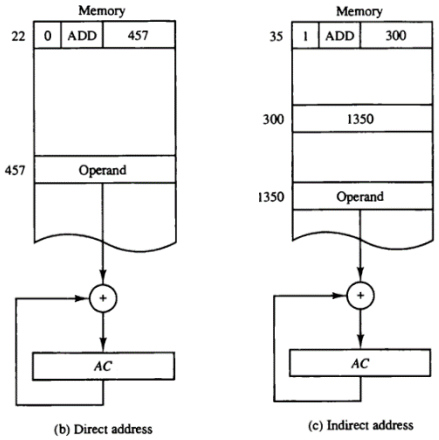
Negotiation with computers?

- Addressing modes
  - Immediate operand address:
    - The second part of an instruction code (***address field***) specifies an ***operand***
  - Direct operand address: ***Fig. 5-2(b)***
    - The second part of an instruction code specifies the ***address of an operand***
  - Indirect operand address: ***Fig. 5-2(c)***
    - The bits in the second part of the instruction designate an ***address of a memory word in which the address of the operand is found*** (used as pointer)
  - One bit of the instruction code is used to distinguish between a direct and an indirect address: ***Fig. 5-2(a)***
- Effective address
  - The ***operand address*** in *computation-type instruction* or the ***target address*** in a *branch-type instruction*

I=0 : Direct,
I=1 : Indirect



(a) Instruction format



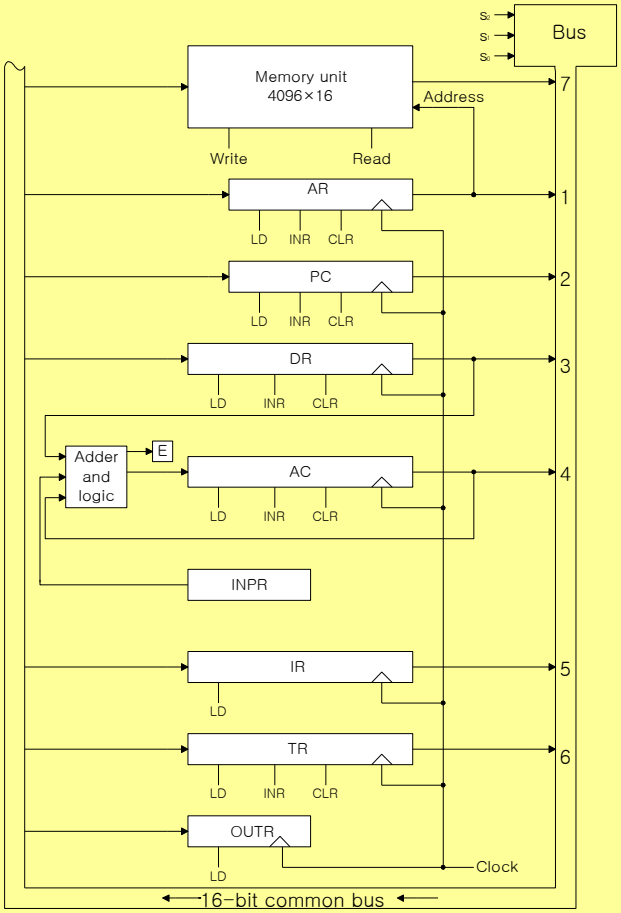
(b) Direct address (c) Indirect address

The nearest storage to the processors!

- List of registers for the basic computer: ***Tab. 5-1***
- Basic computer registers and memory: ***Fig. 5-3***
  - Data Register (***DR***): hold the operand (data) read from memory (***16 bit***)
  - Accumulator Register (***AC***): general purpose processing register (***16 bit***)
  - Instruction Register (***IR***): hold the instruction read from memory (***16 bit***)
  - Temporary Register (***TR***): hold a temporary data during processing (***16 bit***)
  - Address Register (***AR***): hold a memory address (***12 bit***)
  - Program Counter (***PC***): (***12 bit***)
    - Hold the address of the next instruction to be read from memory after the current instruction is executed.
    - Instruction words are read and executed in sequence unless a branch instruction is encountered
      - A branch instruction calls for a transfer to a nonconsecutive instruction in the program
      - The address part of a branch instruction is transferred to PC to become the address of the next instruction
    - To read instruction, memory read cycle is initiated, and PC is incremented by one (next instruction fetch)
  - Input Register (***INPR***): receive an **8-bit** character from an input device
  - Output Register (***OUTR***): hold an **8-bit** character for an output device

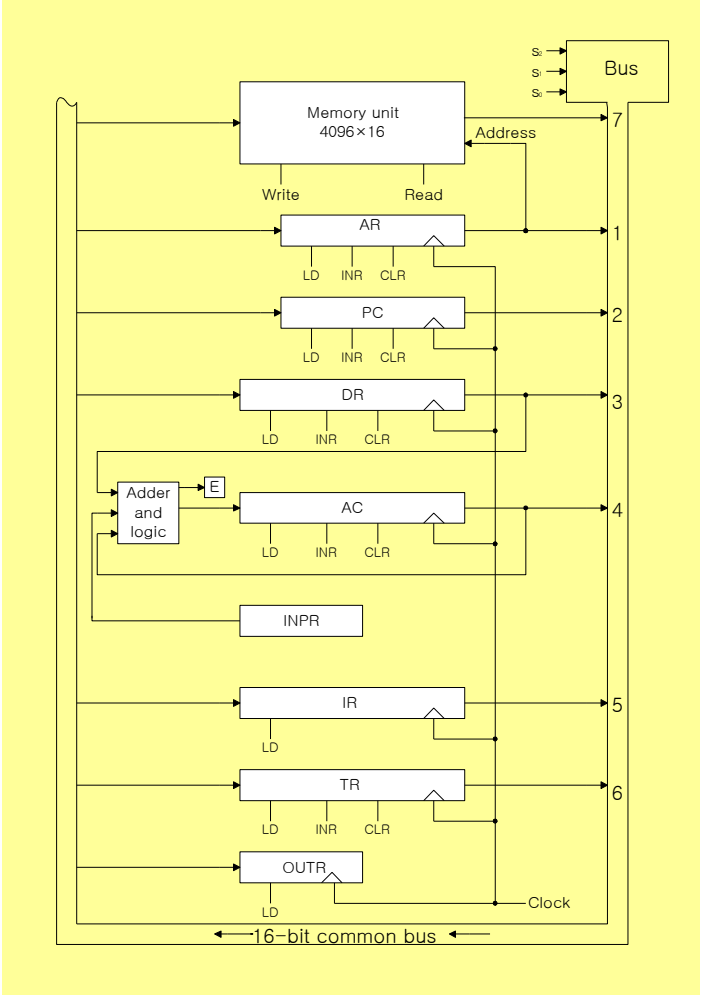
Common bus system

- A more efficient scheme for transferring information in a system with many registers is to use a common bus (*in Sec. 4-3*)
- The basic computer has eight registers, a memory unit, and a control unit (*in Sec. 5-4*)
- Paths must be provided to transfer information from one register to another and between memory and registers



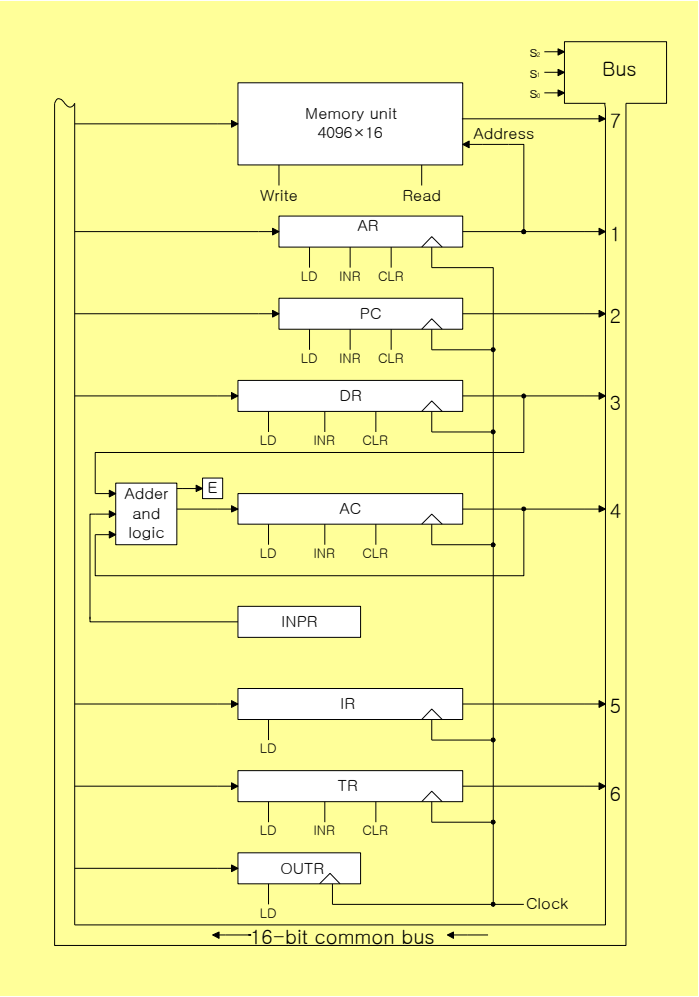
Common bus system

- The connection of the registers and memory of the basic computer to a common bus system : ***Fig. 5-4***
  - The outputs of six registers and memory are connected to the common bus
  - The specific output is selected by mux (S0, S1, S2) :
    - Memory(7), AR(1), PC(2), DR(3), AC(4), IR(5), TR(6)
    - INPR and OUTR are not selected because input/output to/from external device is possible through AC
    - When the mux is selected, data is retrieved from memory or register and placed on the bus
      - When LD (Load Input) is enable, the particular register receives the data from the bus



Common bus system

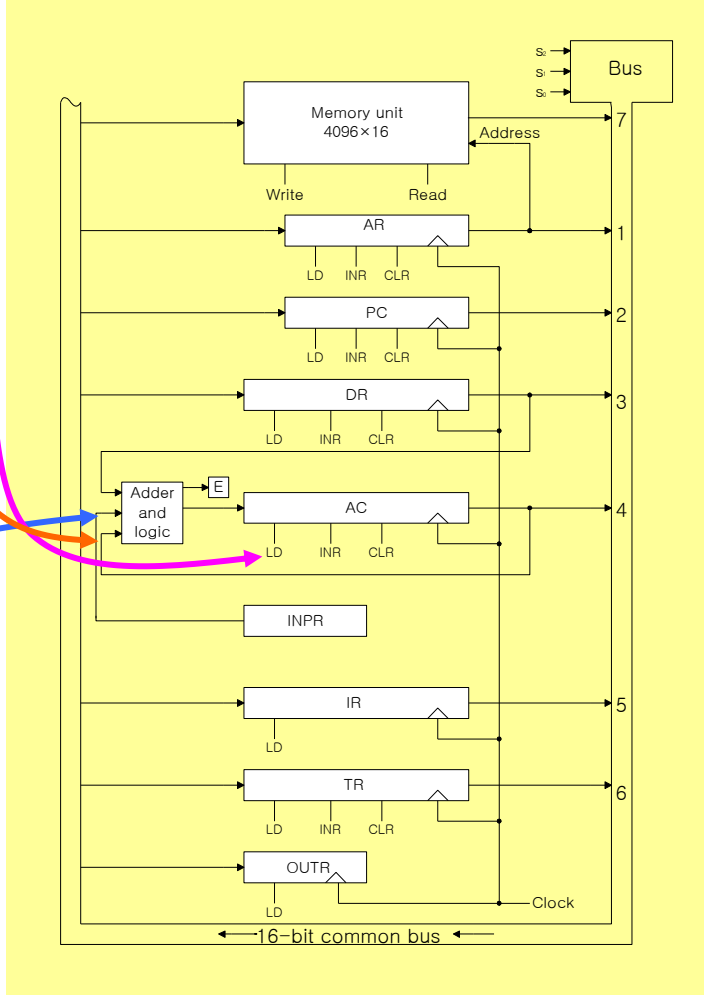
- Control Input: LD, INC, CLR, Write, Read
- Address Register: Separate address bus is not required (simultaneous processing of address and data by one bus)
  - AC can only read memory via DR (*See LDA command on p. 146*)
  - AC can directly write to memory (*See STA command on p. 147*)



Common bus system

- Accumulator (AC): 3 types of input path
  - 1) Register microoperation: clear AC, shfit AC,…
  - 2) Data Register: **add** DR **to** AC, **and** DR **to** AC (The result of the operation is stored in AC and depending on the result, End carry bit set/reset), memory READ (only via DR)
  - 3) INPR: Input data from external device (No need to go through *Adder & Logic*)
- Note) Two microoperations can be executed at the same time

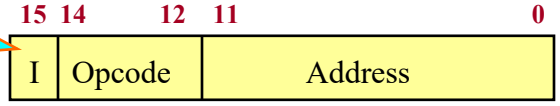
$$DR \leftarrow AC : s_2 s_1 s_0 = 100(4), DR(load)$$
$$AC \leftarrow DR : DR \rightarrow Adder\ \&\ Logic \rightarrow AC(load)$$



Types of instructions (Up to/from here session 4/5)

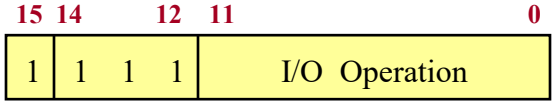
- Three instruction code formats: ***Fig. 5-5***
  - Memory-reference instruction
    - Opcode = 000 ~ 110
      - I=0 : 0xxx ~ 6xxx, I=1: 8xxx ~Exxx

I=0 : Direct,
I=1 : Indirect



  - Register-reference instruction
    - 7xxx (7800 ~ 7001) : CLA, CMA, ...

  - Input-Output instruction
    - Fxxx (F800 ~ F040) : INP, OUT, ION, SKI, ...



Symbol	Hex Code I = 0	Hex Code I = 1	Description
AND	0xxx	8xxx	And memory word to AC
ADD	1xxx	9xxx	Add memory word to AC
LDA	2xxx	Axxx	Load memory word to AC
STA	3xxx	Bxxx	Store content of AC in memory
BUN	4xxx	Cxxx	Branch unconditionally
BSA	5xxx	Dxxx	Branch and Save return address
ISZ	6xxx	Exxx	Increment and skip if zero
CLA	7800		Clear AC
CLE	7400		Clear E
CMS	7200		Complement AC
CME	7100		Complement E
CIR	7080		Circulate right AC and E
CIL	7040		Circulate left AC and E
INC	7020		Increment AC
SPA	7010		Skip next instruction if AC positive
SNA	7008		Skip next instruction if AC negative
SZA	7004		Skip next instruction if AC zero
SZE	7002		Skip next instruction if E is 0
HLT	7001		Halt computer
INP	F800		Input character to AC
OUT	F400		Output character from AC
SKI	F200		Skip on input flag
SKO	F100		Skip on output flag
ION	F080		Interrupt On
IOF	F040		Interrupt Off

What is completeness of an instruction set?

- Instruction set completeness
  - Arithmetic, logical, and shift: CMA, INC, ..
  - Moving information to and from memory and AC: STA, LDA
  - Program control: BUN, BSA, ISZ
  - Input/output: INP, OUT

If the computer includes a sufficient number of instructions in each of the above categories

Clock pulses

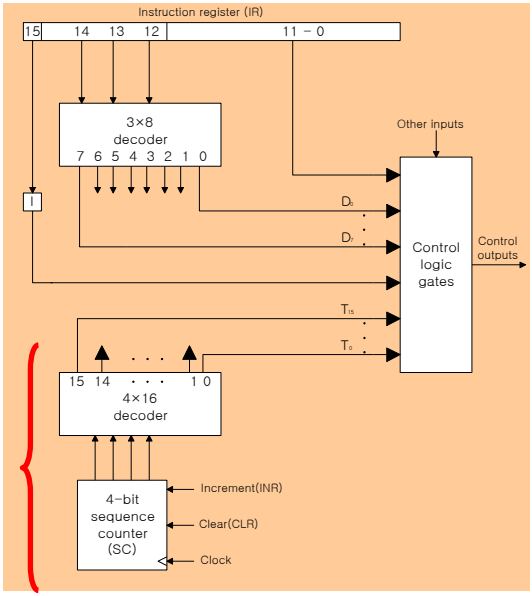
- A master clock generator controls the timing for all registers in the basic computer
- The clock pulses are applied to all FFs and registers in system
- The clock pulses do not change the state of a register unless the register is enabled by a control signal
- The control signals are generated in the control unit: ***Fig. 5-6***
  - The control signals provide control inputs for the multiplexers in the common bus, control inputs in processor registers, and microoperations for the accumulator

Control unit

- Two major types of control organization
  - Hardwired control
    - The control logic is implemented with gates, FFs, decoders, and other digital circuits
    - + Fast operation, - Wiring change (if the design has to be modified)
  - Microprogrammed control
    - The control information is stored in a control memory, and the control memory is programmed to initiate the required sequence of microoperations
    - + Any required change can be done by updating the microprogram in control memory, - Slow operation
  - Will be covered in details later!

Control unit

- Control unit = Control logic gate + 3X8 decoder + instruction register + timing signal
- Timing signal = 4X16 decoder + 4-bit sequence counter
- Example) control timing:
  - Sequence counter is cleared when $D_3T_4 = 1$: $D_3T_4 : SC \leftarrow 0$
    - Memory R/W cycle time > Clock cycle time
      - You must add a wait cycle.



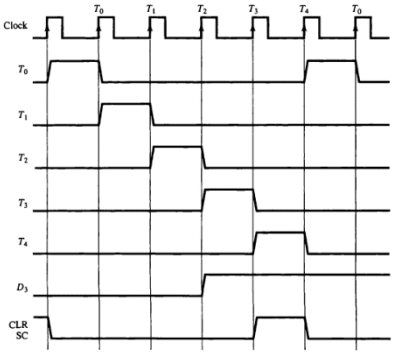


Figure 5-7 Example of control timing signals.

RTL statement (at a glance!)

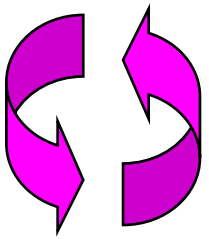
- Example) Register Transfer Language (RTL) statement
  - A transfer of the content of PC into AR if timing signal T_0 is active $T_0 : AR \leftarrow PC$
    - 1) During T_0 active, the content of PC is placed onto the bus $(S_2 S_1 S_0)$
    - 2) LD (load) input of AR is enabled, the actual transfer occurs at the next positive transition of the clock (*T_1 rising edge clock*)
    - 3) SC (sequence counter) is incremented

$0000(T_0) \rightarrow 0001(T_1)$

T_0 : Inactive
T_1 : Active

Instruction engine

- Instruction Cycle
  - 1) Instruction fetch from Memory
  - 2) Instruction decode
  - 3) Read effective address (if indirect addressing mode)
  - 4) Instruction execution
  - 5) Go to step 1) : Next Instruction [PC + 1]



Continue indefinitely unless HALT instruction is encountered

Fetch

- Instruction fetch: T0, T1
  - T0 = 1 $T_0 : AR \leftarrow PC$
    - 1) Place the content of PC onto the bus by making the bus selection inputs $S_2S_1S_0$=010
    - 2) Transfer the content of the bus to AR by enabling the LD input of AR

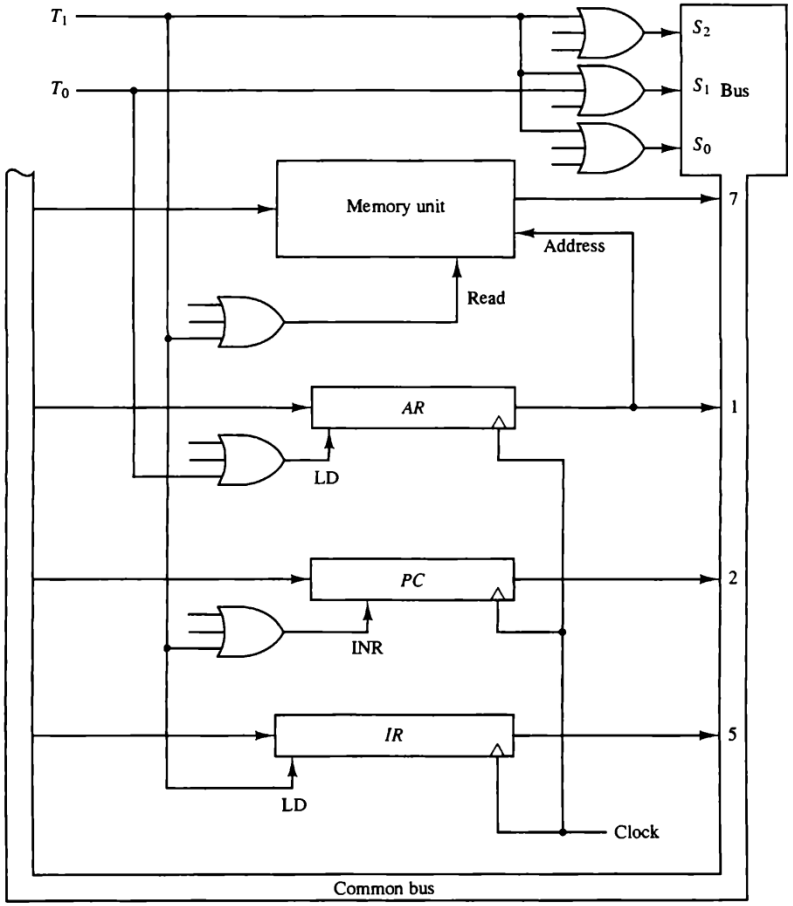


Figure 5-8 Register transfers for the fetch phase.

Fetch

- Instruction fetch: T0, T1
  - T1 = 1 $T_1 : IR \leftarrow M[AR],\ PC \leftarrow PC + 1$
    - 1) Enable the read input memory
    - 2) Place the content of memory onto the bus by making $S_2S_1S_0 = 111$
    - 3) Transfer the content of the bus to IR by enable the LD input of IR
    - 4) Increment PC by enabling the INR input of PC

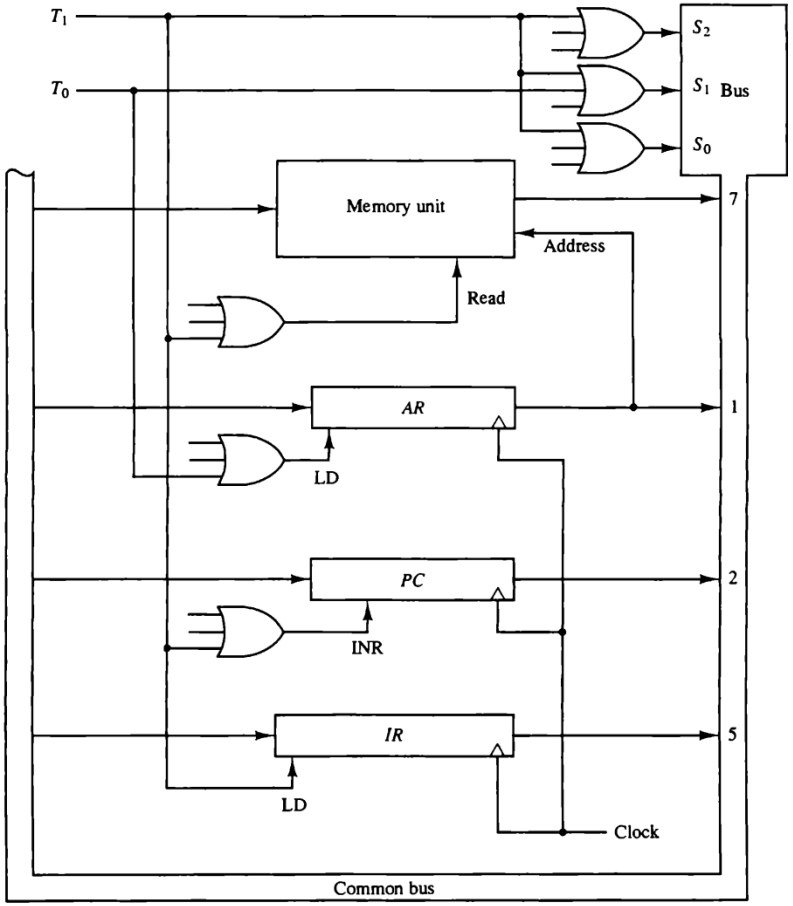


Figure 5-8 Register transfers for the fetch phase.

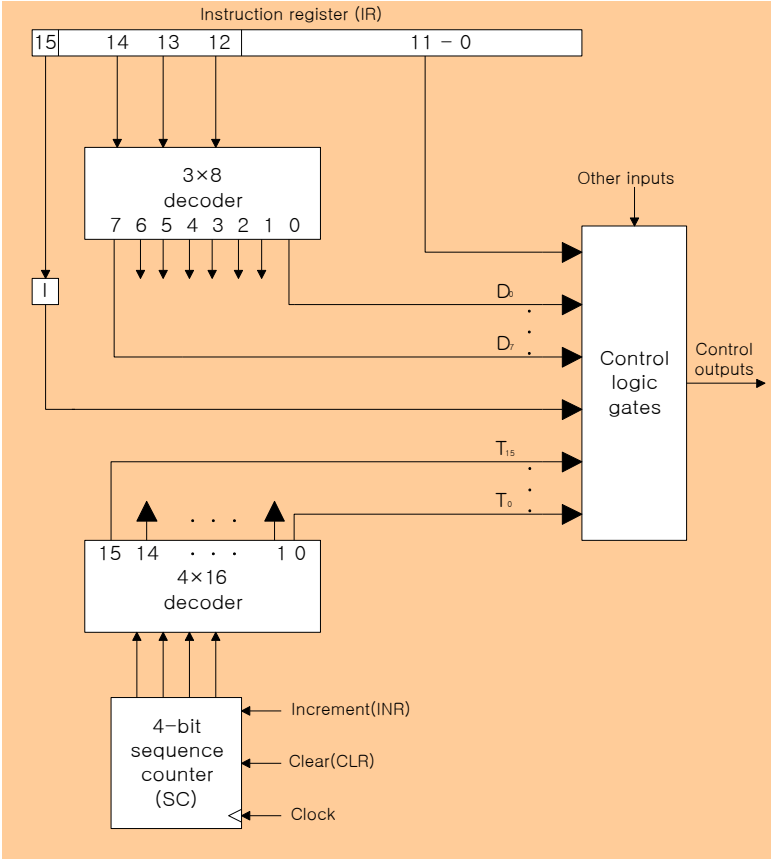
Decode

- Instruction decode: T2

$$T_2 : D_0,....,D_7 \leftarrow Decode\ IR(12-14),\ AR \leftarrow IR(0-11),\ I \leftarrow IR(15)$$

Op.code **Address** **Di/Indirect**

  - IR(12-14) controls D0 - D7 outputs!



Execute

- Instruction execution: T3, T4, T5, T6

$IR(12-14) = 111$

D_7=1: Register (I=0) → $D_7 I' T_3$ (Execute)
I/O (I=1) → $D_7 I T_3$ (Execute)

Read effective Address

D_7=0: Memory Ref. Indirect (I=1) → $D_7' I T_3$ ($AR \leftarrow M[AR]$)
Direct (I=0) → nothing in T_3

  - Register and I/O instructions are executed in T3

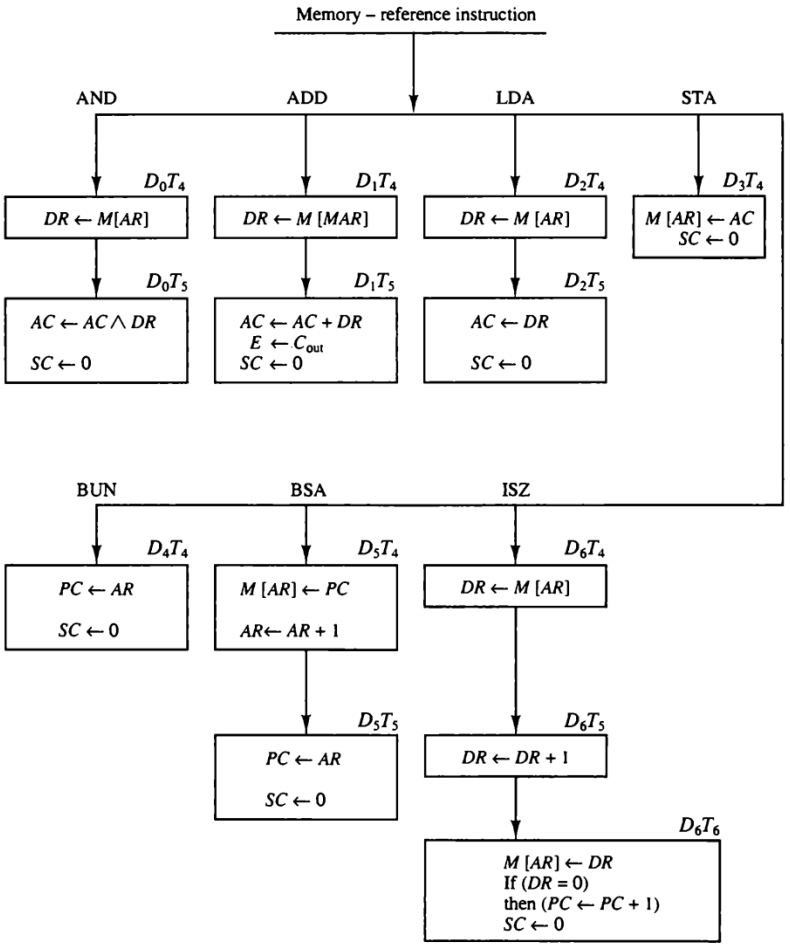


Figure 5-11 Flowchart for memory-reference instructions.

Execute

- Instruction execution: T3, T4, T5, T6
  - Memory reference
    - Effective Address is read in T3.
    - Based on the type of memory command it requires T4, T5, and T6

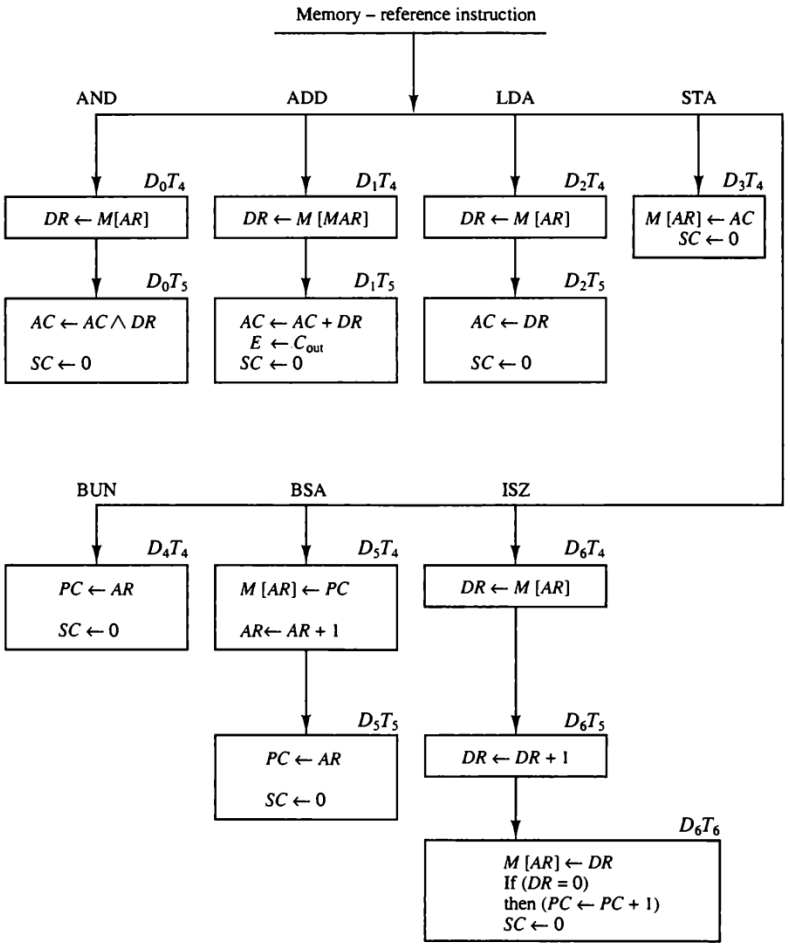


Figure 5-11 Flowchart for memory-reference instructions.

Execute

- Instruction execution: T3, T4, T5, T6
  - Flowchart for instruction cycle (initial configuration)

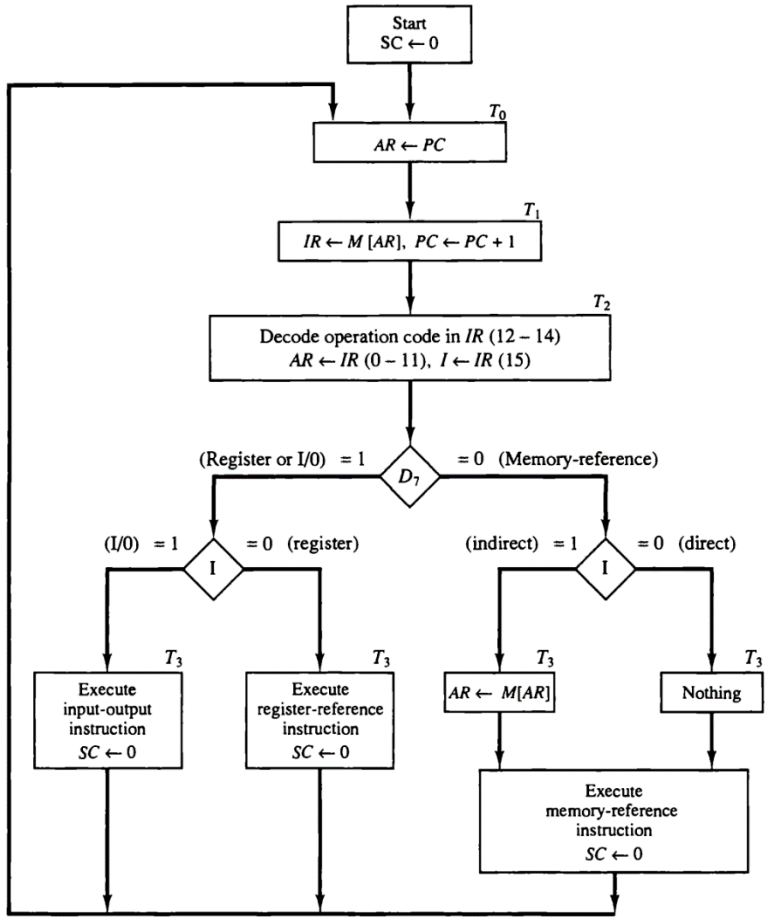


Figure 5-9 Flowchart for instruction cycle (initial configuration).

Execute (Up to/from here session 5/6)

- Instruction execution: T3, T4, T5, T6
  - Flowchart for instruction cycle (initial configuration)
  - Register ref. instruction
    - r = $D_7I'T_3$: Common for all reg. inst.
    - IR(i) = B_i ⟵ IR(0 -11)
    - ***B_0 - B_{11}*** : **12 Register Ref. Instruction**

TABLE 5-3 Execution of Register-Reference Instructions

$D_7I'T_3 = r$ (common to all register-reference instructions)
$IR(i) = B_i$ [bit in IR(0–11) that specifies the operation]

	r:	$SC \leftarrow 0$	Clear SC
CLA	rB_{11}:	$AC \leftarrow 0$	Clear AC
CLE	rB_{10}:	$E \leftarrow 0$	Clear E
CMA	rB_9:	$AC \leftarrow \overline{AC}$	Complement AC
CME	rB_8:	$E \leftarrow \overline{E}$	Complement E
CIR	rB_7:	$AC \leftarrow$ shr AC, $AC(15) \leftarrow E$, $E \leftarrow AC(0)$	Circulate right
CIL	rB_6:	$AC \leftarrow$ shl AC, $AC(0) \leftarrow E$, $E \leftarrow AC(15)$	Circulate left
INC	rB_5:	$AC \leftarrow AC + 1$	Increment AC
SPA	rB_4:	If $(AC(15) = 0)$ then $(PC \leftarrow PC + 1)$	Skip if positive
SNA	rB_3:	If $(AC(15) = 1)$ then $(PC \leftarrow PC + 1)$	Skip if negative
SZA	rB_2:	If $(AC = 0)$ then $PC \leftarrow PC + 1$	Skip if AC zero
SZE	rB_1:	If $(E = 0)$ then $(PC \leftarrow PC + 1)$	Skip if E zero
HLT	rB_0:	$S \leftarrow 0$ (S is a start–stop flip-flop)	Halt computer

By not used address field

Memory-reference instructions

3 X 8 Decoder

- D_7 : Register or I/O = 1
- D_6 - D_0 : **7 Memory ref. instruction**

IR(12,13,14) = 111

TABLE 5-4 Memory-Reference Instructions

Symbol	Operation decoder	Symbolic description
AND	D_0	$AC \leftarrow AC \wedge M[AR]$
ADD	D_1	$AC \leftarrow AC + M[AR]$, $E \leftarrow C_{out}$
LDA	D_2	$AC \leftarrow M[AR]$
STA	D_3	$M[AR] \leftarrow AC$
BUN	D_4	$PC \leftarrow AR$
BSA	D_5	$M[AR] \leftarrow PC$, $PC \leftarrow AR + 1$
ISZ	D_6	$M[AR] \leftarrow M[AR] + 1$, If $M[AR] + 1 = 0$ then $PC \leftarrow PC + 1$

- AND to AC

$$D_0T_4 : DR \leftarrow M[AR]$$
$$D_0T_5 : AC \leftarrow AC \wedge DR, SC \leftarrow 0$$

- ADD to AC

$$D_1T_4 : DR \leftarrow M[AR]$$
$$D_1T_5 : AC \leftarrow AC + DR, E \leftarrow C_{out}, SC \leftarrow 0$$

- LDA: memory read

$$D_2T_4 : DR \leftarrow M[AR]$$
$$D_2T_5 : AC \leftarrow DR, SC \leftarrow 0$$

- Memory r/w cycle time > clock cycle time
  - You must add a wait cycle

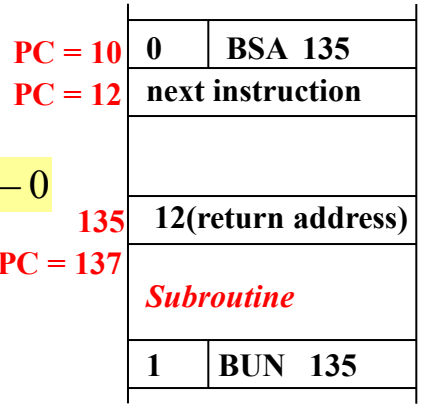
Memory-reference instructions

- STA: memory write $D_3T_4 : M[AR] \leftarrow AC, SC \leftarrow 0$
- BUN: branch unconditionally $D_4T_4 : PC \leftarrow AR, SC \leftarrow 0$
- BSA: branch and save return address

$$D_5T_4 : M[AR] \leftarrow PC, AR \leftarrow AR + 1$$
$$D_5T_5 : PC \leftarrow AR, SC \leftarrow 0$$

  - Return address: save return address (135 ← 12)
  - Subroutine call: $D_5T_4: M[135] \leftarrow 12(PC),\ 136(AR) \leftarrow 135 + 1$, $D_5T_5: 136(PC) \leftarrow 136(AR),\ SC \leftarrow 0$
- ISZ: increment and skip if zero

$$D_6T_4 : DR \leftarrow M[AR]$$
$$D_6T_5 : DR \leftarrow DR + 1$$
$$D_6T_6 : M[AR] \leftarrow DR, if\ (DR = 0)\ then\ (PC \leftarrow PC + 1), SC \leftarrow 0$$



Memory-reference instructions

- Control flowchart
  - Flowchart for the 7 memory reference instruction
    - The longest instruction: ISZ (T6)

Symbol	Hexadecimal code $I = 0$	$I = 1$	Description
AND	0xxx	8xxx	AND memory word to AC
ADD	1xxx	9xxx	Add memory word to AC
LDA	2xxx	Axxx	Load memory word to AC
STA	3xxx	Bxxx	Store content of AC in memory
BUN	4xxx	Cxxx	Branch unconditionally
BSA	5xxx	Dxxx	Branch and save return address
ISZ	6xxx	Exxx	Increment and skip if zero

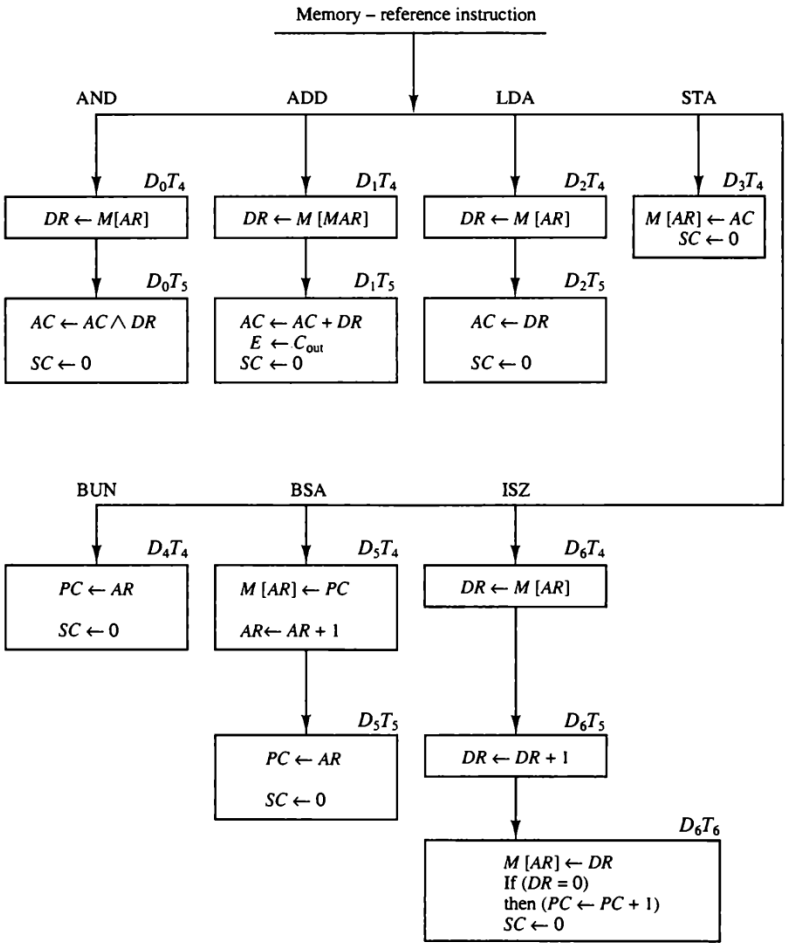


Figure 5-11 Flowchart for memory-reference instructions.

Memory-reference instructions

- Control flowchart
  - Flowchart for the 7 memory reference instruction
    - Therefore, it can be implemented as 3 bit sequence counter (currently 4 bits are prepared for expansion)

Symbol	Hexadecimal code $I = 0$	Hexadecimal code $I = 1$	Description
AND	0xxx	8xxx	AND memory word to AC
ADD	1xxx	9xxx	Add memory word to AC
LDA	2xxx	Axxx	Load memory word to AC
STA	3xxx	Bxxx	Store content of AC in memory
BUN	4xxx	Cxxx	Branch unconditionally
BSA	5xxx	Dxxx	Branch and save return address
ISZ	6xxx	Exxx	Increment and skip if zero

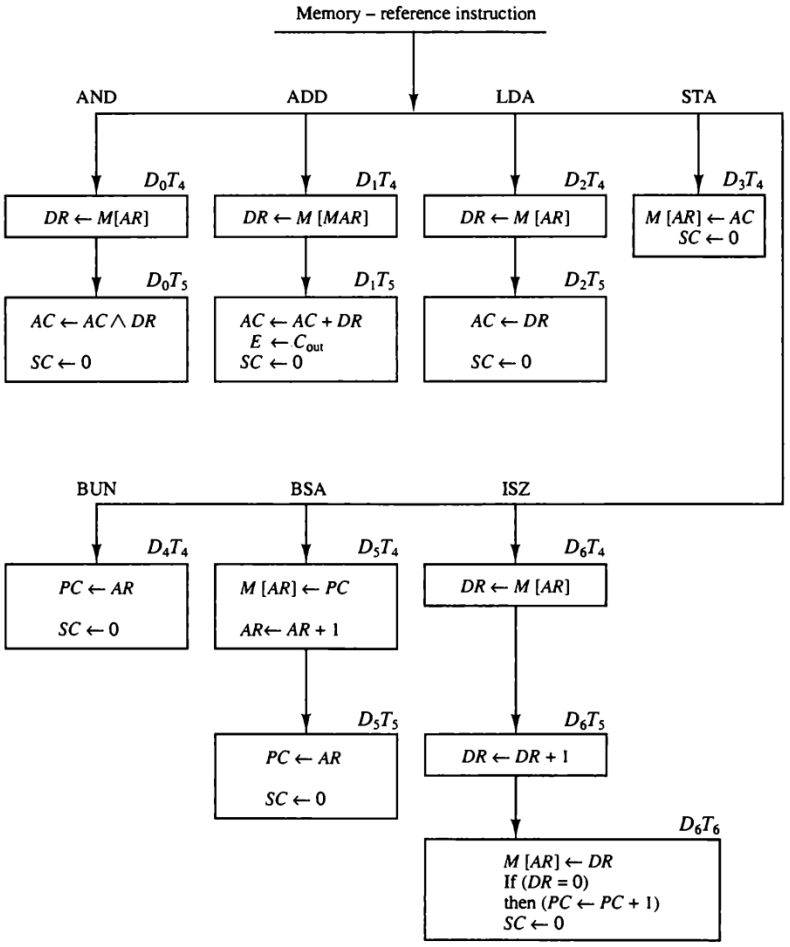


Figure 5-11 Flowchart for memory-reference instructions.

Question

- An instruction at address 021 in the basic computer has $I = 0$, an operation code of the AND instruction, and an address part equal to 083 (all numbers are in hexadecimal). The memory word at address 083 contains the operand B8F2 and the content of AC is A937. Go over the instruction cycle and determine the contents of the following registers at the end of the execute phase: PC, AR, DR, AC, and IR. Repeat the problem six more times starting with an operation code of another memory-reference instruction.

Answer

	PC	AR	DR	AC	IR
Initial	021	—	—	A937	—
AND	022	083	B8F2	A832	0083
ADD	022	083	B8F2	6229	1083
LDA	022	083	B8F2	B8F2	2083
STA	022	083	—	A937	3083
BUN	083	083	—	A937	4083
BSA	084	084	—	A937	5083
ISZ	022	083	B8F3	A937	6083

Input-output and interrupt

- Input-output configuration: ***Fig. 5-12***
  - Input register (***INPR***), output register (***OUTR***)
    - These two registers communicate with a communication interface serially and with the AC in parallel
    - Each quantity of information has eight bits of an alphanumeric code
  - Input flag (***FGI***), output flag (***FGO***)
    - FGI: ***set*** when INPR is ready, ***clear*** when INPR is empty
    - FGO: ***set*** when operation is completed, ***clear*** when output device is in the process of printing

1 : Ready
0 : Not ready

- Input-output instruction:
  - p = D_7IT_3: Common part
  - IR(i) = B_i ⟵ IR(6 -11)
  - ***B_6 - B_{11}***: **6 I/O Instruction**

Not used for address

TABLE 5-5 Input-Output Instructions

$D_7IT_3 = p$ (common to all input–output instructions)
$IR(i) = B_i$ [bit in $IR(6–11)$ that specifies the instruction]

	p:	$SC \leftarrow 0$	Clear SC
INP	pB_{11}:	$AC(0–7) \leftarrow INPR$, $FGI \leftarrow 0$	Input character
OUT	pB_{10}:	$OUTR \leftarrow AC(0–7)$, $FGO \leftarrow 0$	Output character
SKI	pB_9:	If ($FGI = 1$) then ($PC \leftarrow PC + 1$)	Skip on input flag
SKO	pB_8:	If ($FGO = 1$) then ($PC \leftarrow PC + 1$)	Skip on output flag
ION	pB_7:	$IEN \leftarrow 1$	Interrupt enable on
IOF	pB_6:	$IEN \leftarrow 0$	Interrupt enable off

Input-output and interrupt (Up to/from here session 6/7)

- CPU

```
/* Input */        /* Initially FGI = 0 */
  loop:  If FGI = 0 goto loop
         AC ← INPR,  FGI ← 0

/* Output */       /* Initially FGO = 1 */
  loop:  If FGO = 0 goto loop
         OUTR ← AC,  FGO ← 0
```

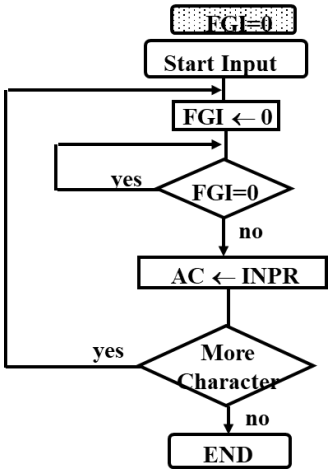
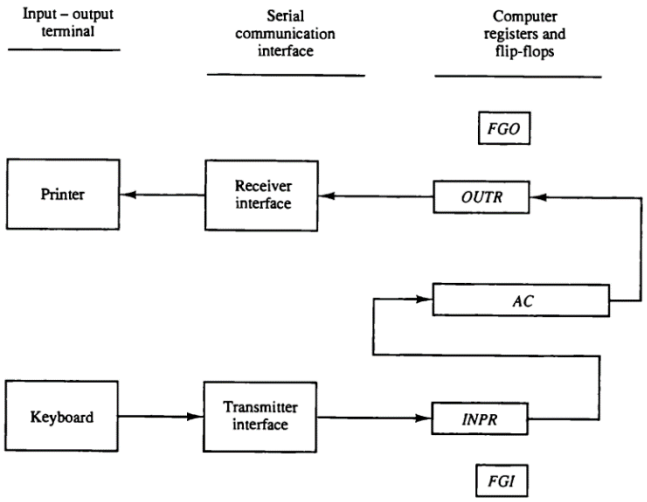


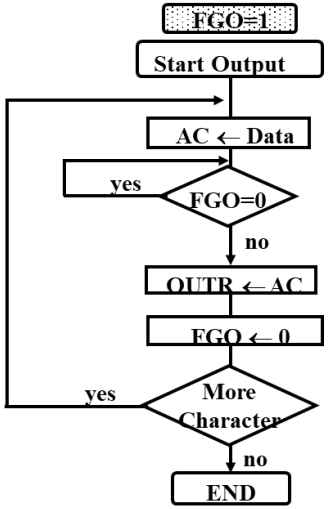
Figure 5-12 Input-output configuration.



- I/O device

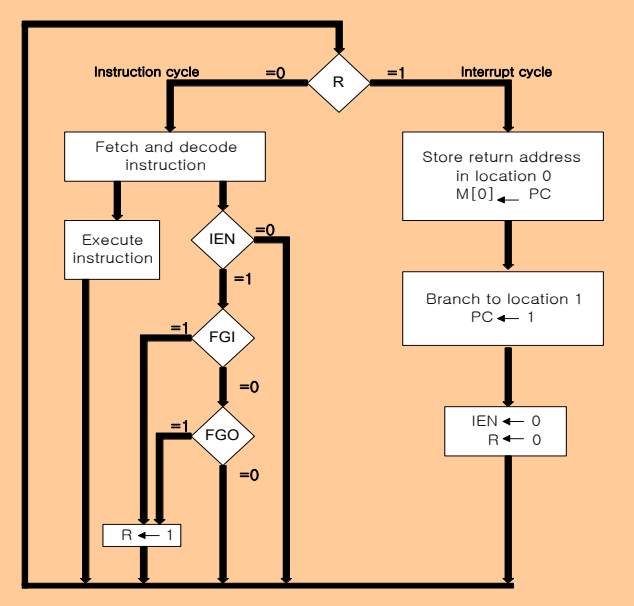
```
loop: If FGI = 1 goto loop
      INPR ← new data, FGI ← 1

loop: If FGO = 1 goto loop
      consume OUTR, FGO ← 1
```



Input-output and interrupt

- Program interrupt
  - I/O transfer modes
    - 1) Programmed I/O
    - 2) Interrupt-initiated I/O
    - 3) DMA
    - 4) IOP (IO Processor)
    - Here we consider **interrupt-initiated I/O method** is used
      - If FGI or FGO is 1: Interrupt is enabled!
      - Maskable Interrupt
        - » Interrupt mask is enabled with ION
        - » Interrupt mask is disabled with IOF



- Interrupt cycle
  - During the execute phase, IEN is checked by the control
    - IEN = 0: the programmer does not want to use the interrupt, so control continues with the next instruction cycle
    - IEN = 1: the control circuit checks the flag bit, if either flag set to 1, R F/F is set to 1
  - At the end of the execute phase, control checks the value of R
    - R = 0: enter the normal instruction cycle
    - R = 1: enter the interrupt cycle

Input-output and interrupt

- Demonstration of the interrupt cycle: ***Fig. 5-14***
  - The memory location at address 0 as the place for storing the return address
  - Whenever an interrupt occurs branch to memory location 1
  - Always set IEN=0 in interrupt cycle (*Therefore, in order to receive interrupt after running ISR, the ION instruction must be executed at the end of ISR*)
- The condition for R = 1

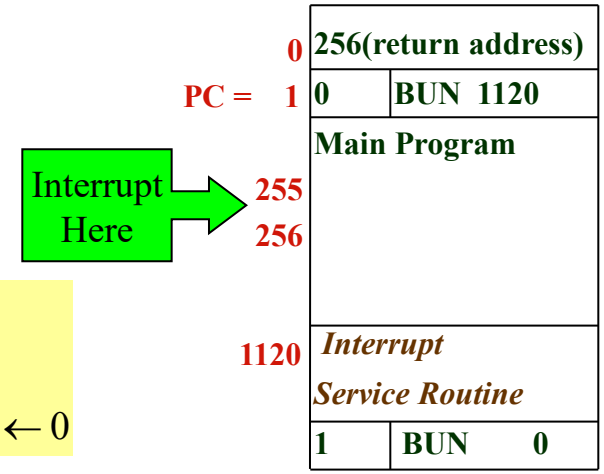
$$T_0' T_1' T_2' (IEN)(FGI + FGO) : R \leftarrow 1$$

- Modified fetch phase for interrupts
  - Modified fetch and decode Phase

Save Return Address (PC) at 0

Jump to 1(PC=1)

$$RT_0 : AR \leftarrow 0, TR \leftarrow PC$$
$$RT_1 : M[AR] \leftarrow TR, PC \leftarrow 0$$
$$RT_2 : PC \leftarrow PC + 1, IEN \leftarrow 0, R \leftarrow 0, SC \leftarrow 0$$



Complete computer description

- The final flowchart of the instruction cycle
- The control function and microoperation

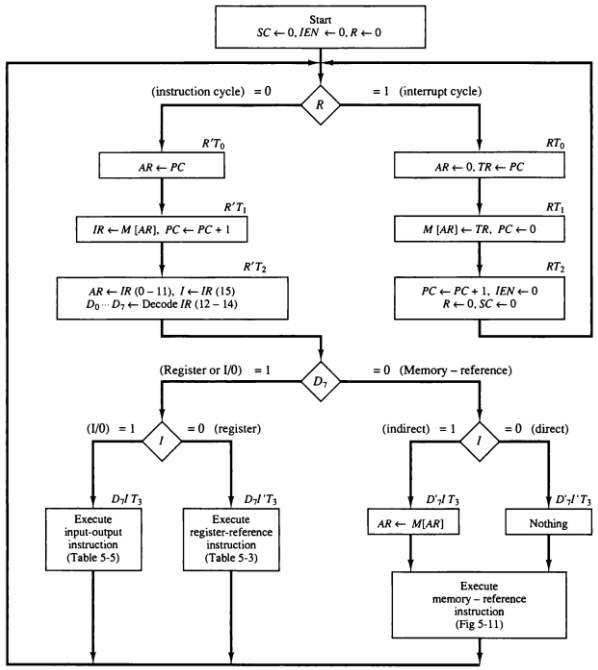


Figure 5-15 Flowchart for computer operation.

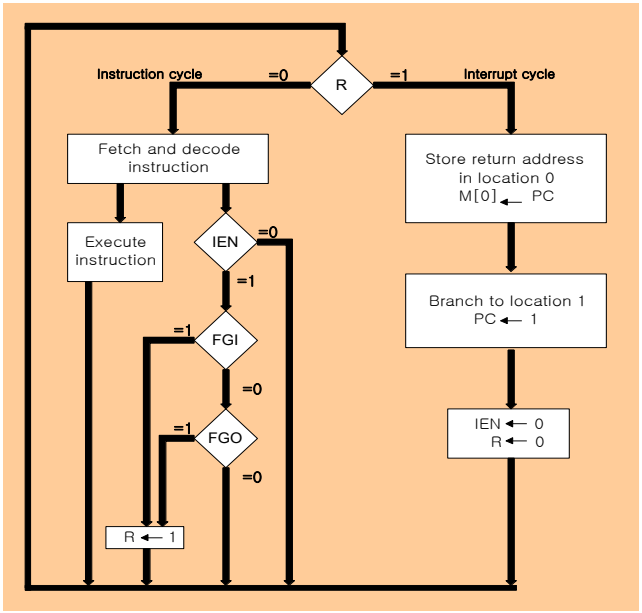


TABLE 5-6 Control Functions and Microoperations for the Basic Computer

Fetch	$R'T_0$:	$AR \leftarrow PC$
	$R'T_1$:	$IR \leftarrow M[AR]$, $PC \leftarrow PC + 1$
Decode	$R'T_2$:	$D_0, \ldots, D_7 \leftarrow$ Decode $IR(12\text{–}14)$, $AR \leftarrow IR(0\text{–}11)$, $I \leftarrow IR(15)$
Indirect	D_7IT_3:	$AR \leftarrow M[AR]$
Interrupt:		
	$T_0'T_1'T_2(IEN)(FGI + FGO)$:	$R \leftarrow 1$
	RT_0:	$AR \leftarrow 0$, $TR \leftarrow PC$
	RT_1:	$M[AR] \leftarrow TR$, $PC \leftarrow 0$
	RT_2:	$PC \leftarrow PC + 1$, $IEN \leftarrow 0$, $R \leftarrow 0$, $SC \leftarrow 0$
Memory-reference:		
AND	D_0T_4:	$DR \leftarrow M[AR]$
	D_0T_5:	$AC \leftarrow AC \wedge DR$, $SC \leftarrow 0$
ADD	D_1T_4:	$DR \leftarrow M[AR]$
	D_1T_5:	$AC \leftarrow AC + DR$, $E \leftarrow C_{out}$, $SC \leftarrow 0$
LDA	D_2T_4:	$DR \leftarrow M[AR]$
	D_2T_5:	$AC \leftarrow DR$, $SC \leftarrow 0$
STA	D_3T_4:	$M[AR] \leftarrow AC$, $SC \leftarrow 0$
BUN	D_4T_4:	$PC \leftarrow AR$, $SC \leftarrow 0$
BSA	D_5T_4:	$M[AR] \leftarrow PC$, $AR \leftarrow AR + 1$
	D_5T_5:	$PC \leftarrow AR$, $SC \leftarrow 0$
ISZ	D_6T_4:	$DR \leftarrow M[AR]$
	D_6T_5:	$DR \leftarrow DR + 1$
	D_6T_6:	$M[AR] \leftarrow DR$, if $(DR = 0)$ then $(PC \leftarrow PC + 1)$, $SC \leftarrow 0$
Register-reference:		$D_7I'T_3 = r$ (common to all register-reference instructions) $IR(i) = B_i$ $(i = 0, 1, 2, \ldots, 11)$
	r:	$SC \leftarrow 0$
CLA	rB_{11}:	$AC \leftarrow 0$
CLE	rB_{10}:	$E \leftarrow 0$
CMA	rB_9:	$AC \leftarrow \overline{AC}$
CME	rB_8:	$E \leftarrow \overline{E}$
CIR	rB_7:	$AC \leftarrow$ shr AC, $AC(15) \leftarrow E$, $E \leftarrow AC(0)$
CIL	rB_6:	$AC \leftarrow$ shl AC, $AC(0) \leftarrow E$, $E \leftarrow AC(15)$
INC	rB_5:	$AC \leftarrow AC + 1$
SPA	rB_4:	If $(AC(15) = 0)$ then $(PC \leftarrow PC + 1)$
SNA	rB_3:	If $(AC(15) = 1)$ then $(PC \leftarrow PC + 1)$
SZA	rB_2:	If $(AC = 0)$ then $PC \leftarrow PC + 1)$
SZE	rB_1:	If $(E = 0)$ then $(PC \leftarrow PC + 1)$
HLT	rB_0:	$S \leftarrow 0$
Input-output:		$D_7IT_3 = p$ (common to all input–output instructions) $IR(i) = B_i$ $(i = 6, 7, 8, 9, 10, 11)$
	p:	$SC \leftarrow 0$
INP	pB_{11}:	$AC(0\text{–}7) \leftarrow INPR$, $FGI \leftarrow 0$
OUT	pB_{10}:	$OUTR \leftarrow AC(0\text{–}7)$, $FGO \leftarrow 0$
SKI	pB_9:	If $(FGI = 1)$ then $(PC \leftarrow PC + 1)$
SKO	pB_8:	If $(FGO = 1)$ then $(PC \leftarrow PC + 1)$
ION	pB_7:	$IEN \leftarrow 1$
IOF	pB_6:	$IEN \leftarrow 0$

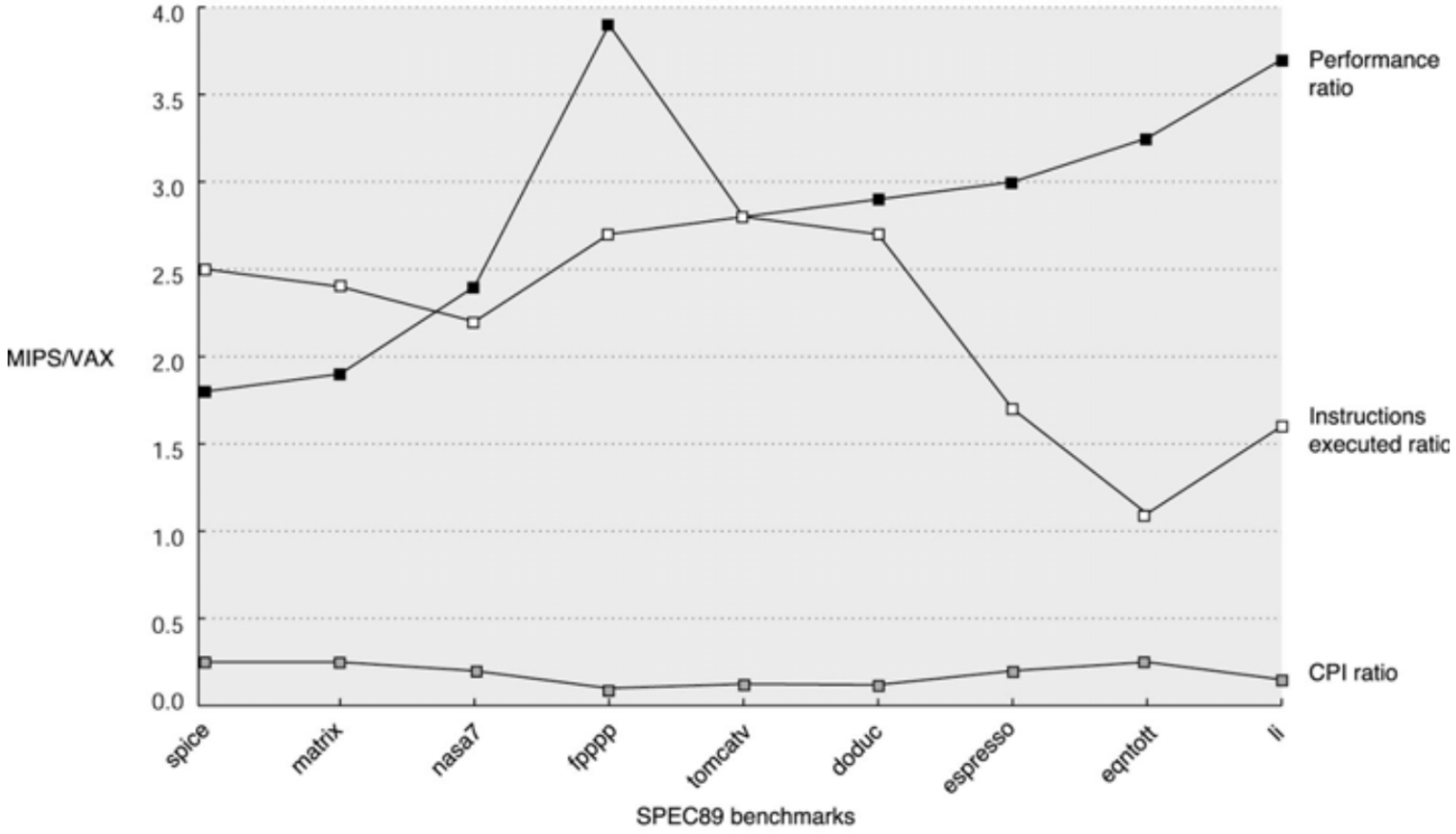
Introduction

- RISC (Reduced Instruction Set Computer)
- CISC (Complex Instruction Set Computer)
- Debate raged from early 80s through 90s
- Now it is fairly irrelevant
- Research in the late 70s/early 80s led to RISC
  - IBM 801 -- John Cocke – mid 70s
  - Berkeley RISC-1 (Patterson)
  - Stanford MIPS (Hennessy)

RISC vs. CISC arguments

- RISC
  - Simple Implementation
  - Load/store, fixed-format 32-bit instructions, efficient pipelines
  - Lower CPI
  - Compilers do a lot of the hard work
    - MIPS = Microprocessor without Interlocked Pipelined Stages
- CISC
  - Simple Compilers (assists hand-coding, many addressing modes, many instructions)
  - Code density

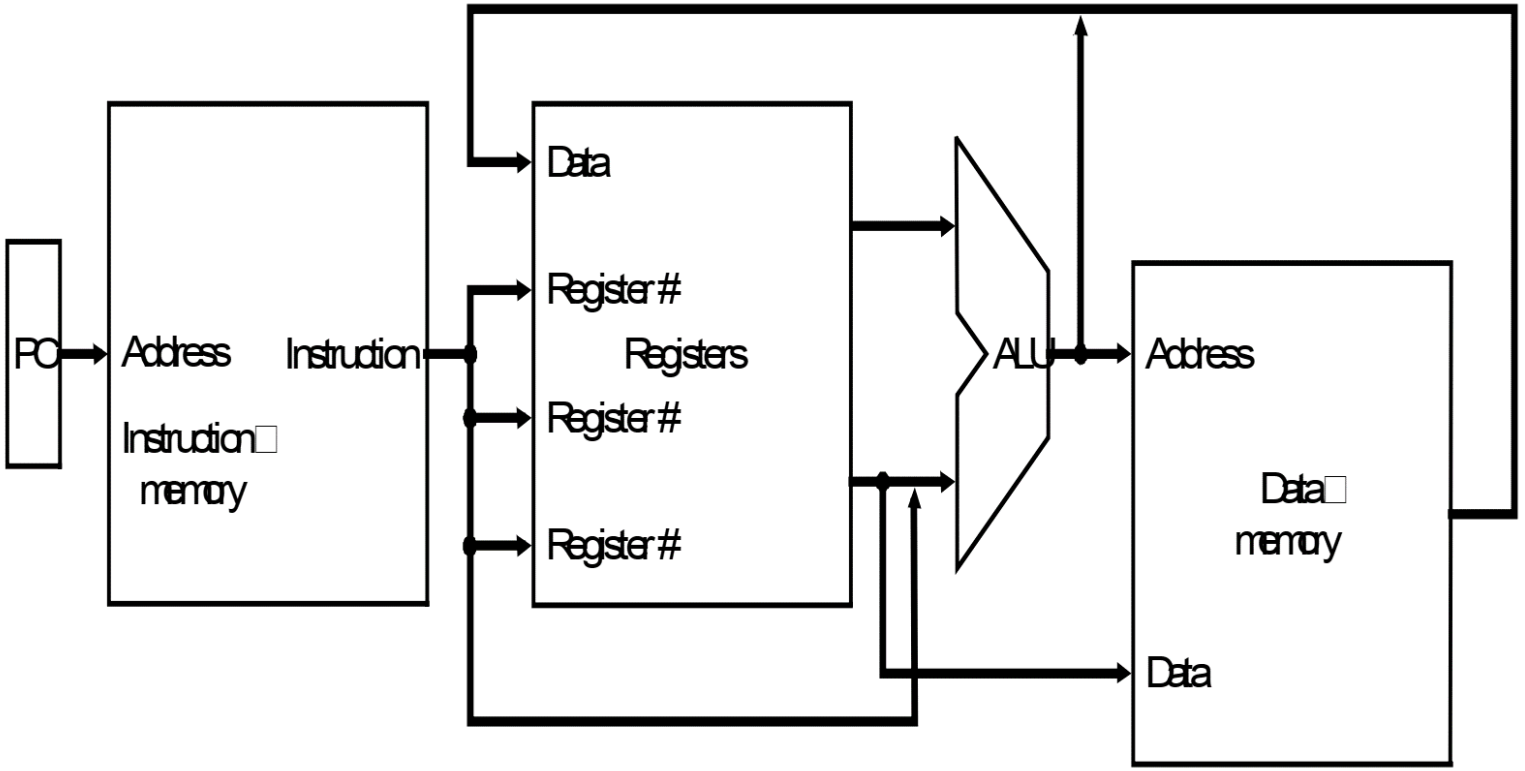
MIPS/VAX comparison



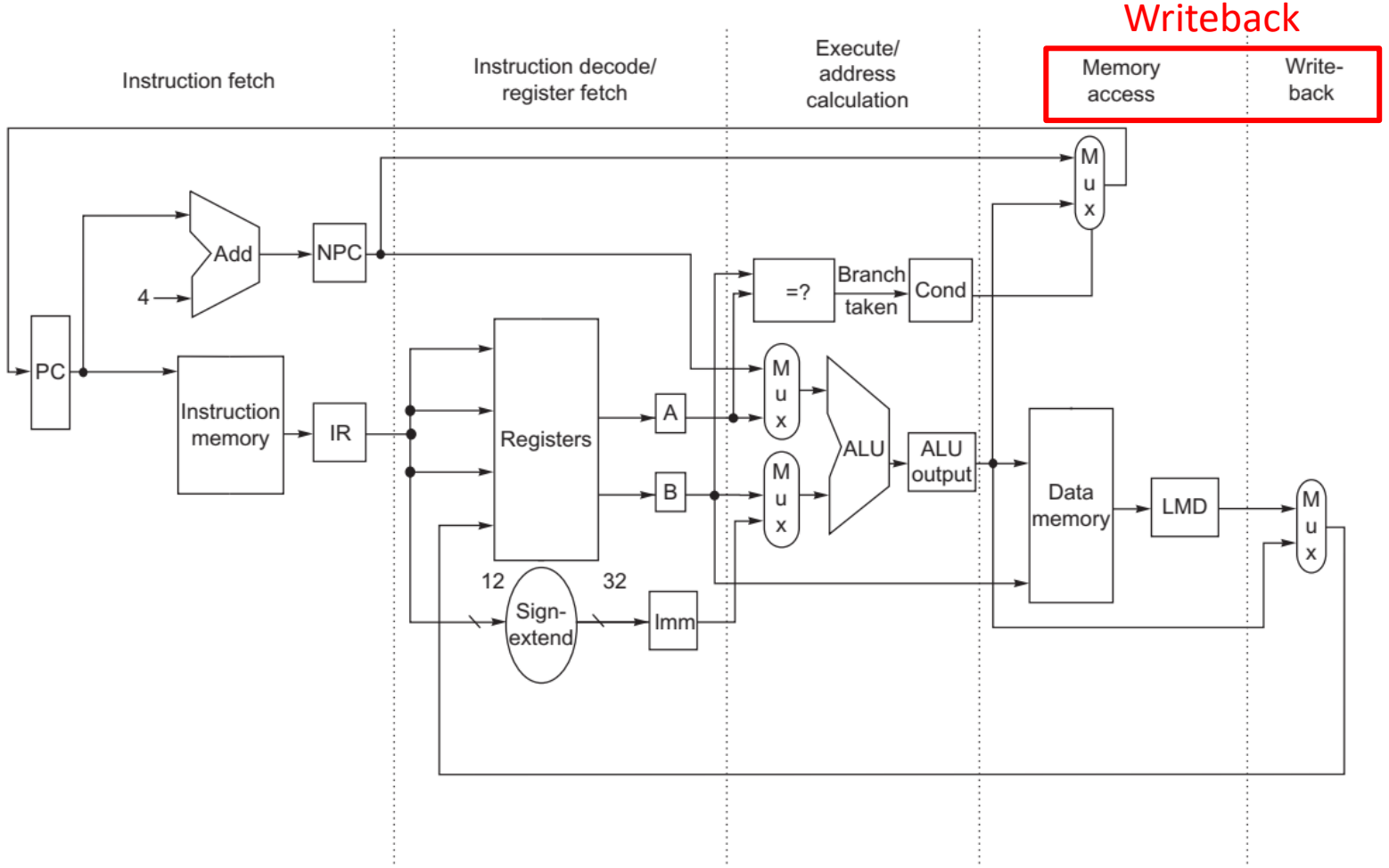
After the dust settled!

- Turns out it doesn't matter much
- Can decode CISC instructions into internal "microISA"
  - This takes a couple of extra cycles (PLA implementation) and a few hundred thousand transistors
  - In 20 stage pipelines, 55M tx processors this is minimal
  - Pentium 4 caches these micro-Ops
- Actually may have some advantages
  - External ISA for compatibility, internal ISA can be tweaked each generation (Transmeta Crusoe)

Abstract implementation



Final implementation (MIPS/RISC V data path)



(Up to here session 7)