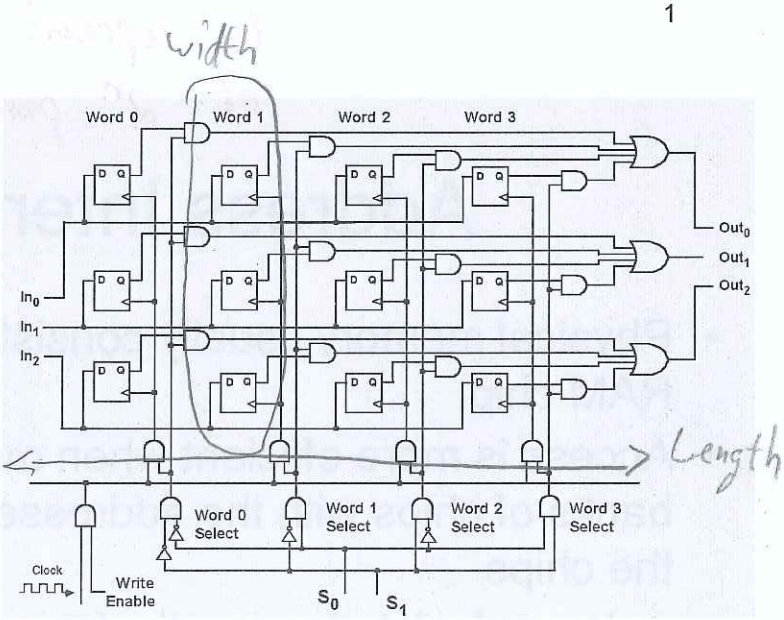
Lecture 13

Memory



Write a word to memory:

1. An address is asserted on S_0 and S_1.
2. Write enable (WE) is set to high.
3. The decoder using S_0 and S_1 enables only one AND gate, selecting a given word in memory.
4. The line selected in Step 3 combines with the clock and WE select only one word.
5. The write gate enabled in Step 4 drives the clock for the selected word.
6. When the clock pulses, the word on the input lines is loaded into the D flip-flop.

4x3 memory

Memory Organization

- Computer memory consists of a linear array of addressable storage cells that are similar to registers.
- Memory can be byte-addressable, or word-addressable, where a word typically consists of two or more bytes.
- Memory is constructed of RAM (random access memory) chips, often referred to in terms of length × width.
  - e.g., 4M × 16: 4M ($2^2 \times 2^{20} = 2^{22}$) items and each item is 16 bits width. This requires 2^{22} different addresses to uniquely identify 2^{22} items.
  - In general, unsigned numbers are used for memory addresses. 2^{22} items are indexed from 0 to $2^{22} - 1$ in memory.
  - Indexing M items requires $\lceil \log_2 M \rceil$ bits (address lines) to represent the address.

Address Interleaving

- Physical memory usually consists of more than one RAM chip.
- Access is more efficient when memory is organized into banks of chips with the addresses interleaved across the chips
  - low-order interleaving: the low order bits of the address specify which memory bank contains the address of interest.
  - High-order interleaving: the high order address bits specify the memory bank.

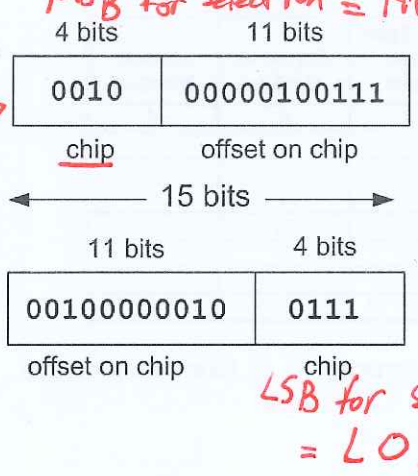
Exercise

Assume a memory consisting of 16 2K x 8 bit chips.

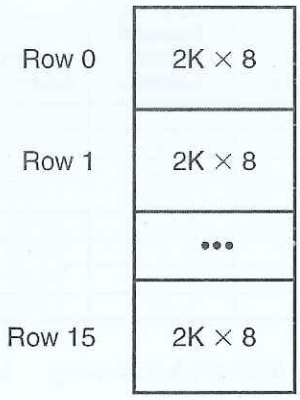
- Memory is 32K = $2^5 \times 2^{10} = 2^{15}$ = n
- 15 bits are needed for each address.
- 4 bits to select the chip, and 11 bits for the offset into the chip that selects the byte.
  - In high-order interleaving the high-order 4 bits select the chip.
  - In low-order interleaving the low-order 4 bits select the chip.

MSB for selection = HOI

16 chips = 2^4

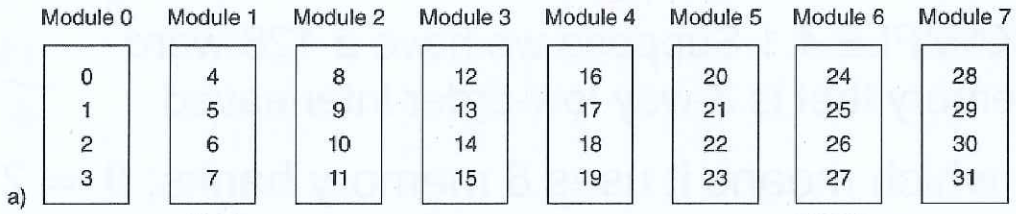


LSB for selection = LOI

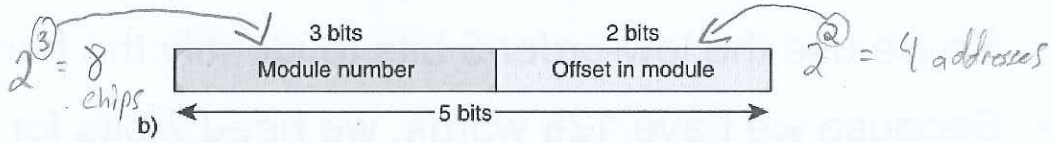


row 2 selected 0010

Memory Organization Exercise



8 chips with 4 addresses
8 x 4 = 32



$2^3 = 8$ chips

$2^2 = 4$ addresses

Module	Decimal Word Address	Binary Address	Address Split per Given Structure	Module Number	Offset in Module
Module 0	0	00000	000 00	0	0
	1	00001	000 01	0	1
	2	00010	000 10	0	2
	3	00011	000 11	0	3
Module 1	4	00100	001 00	1	0
	5	00101	001 01	1	1
→	6	00110	001 10	1	2
	7	00111	001 11	1	3

c)

1st chip 3rd column

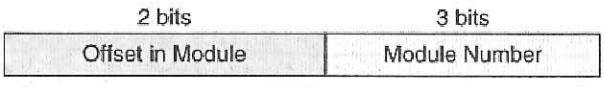
a) High-Order Memory Interleaving b) Address Structure c) First Two Modules

Memory Organization Exercise

a)

Module 0	Module 1	Module 2	Module 3	Module 4	Module 5	Module 6	Module 7
0	1	2	3	4	5	6	7
8	9	10	11	12	13	14	15
16	17	18	19	20	21	22	23
24	25	26	27	28	29	30	31

b)



c)

Module	Decimal Word Address	Binary Address	Address Split per Given Structure	Offset in Module	Module Number
Module 0	0	00000	00 000	0	0
	8	01000	01 000	1	0
	16	10000	10 000	2	0
	24	11000	11 000	3	0
Module 1	1	00001	00 001	0	1
	9	01001	01 001	1	1
	17	10001	10 001	2	1
	25	11001	11 001	3	1

a) Low-Order Memory Interleaving b) Address Structure c) First Two Modules

Exercise

Understand!

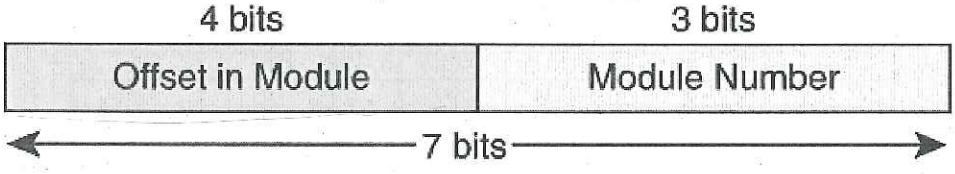
$128 = \frac{2^7}{2^3}$ $\frac{128}{8} = 2^4$

$\frac{128}{8} = 16$ 2^4

- EXAMPLE 4.1 Suppose we have a 128-word memory that is 8-way low-order interleaved
  - which means it uses 8 memory banks; $8 = 2^3$
- So we use the low-order 3 bits to identify the bank.
- Because we have 128 words, we need 7 bits for each address ($128 = 2^7$).

8 chips of 128 words need 7 bits

$2^7 = 128$



Interrupts

- The normal execution of a program is altered when an event of higher-priority occurs. The CPU is alerted to such an event through an interrupt.
- Interrupts can be triggered by
  - I/O requests, arithmetic errors (such as division by zero), or when an invalid instruction is encountered.
- Interrupt handling: Each interrupt is associated with a procedure that directs the actions of the CPU when an interrupt occurs.
- Interrupts are classified:
  - Maskable: disabled or ignored
  - Nonmaskable interrupts are high-priority interrupts that cannot be disabled and must be acknowledged.

Exercise

2^{25} from 2^{20} $2^8 \times 2^{20} = 28$ 2^{32} 32 bit words

1. Suppose a system has a byte-addressable memory size of 256MB. How many bits are required for each address? = $2^8 = 256$ $2^{20} = M \rightarrow 2^{28} = 28$ bits
2. Suppose that a 32MB system memory is built from 32 1MB RAM chips. How many address lines are needed to select one of the memory chips? $2^5 \times 2^{20} = 2^{25}$ = need the 5 bit for select
3. Suppose that a system uses 32-bit memory words and its memory is built from 16 1M × 8 RAM chips. How many address bits are required to uniquely identify each memory word? $1M/4 = 16 \times 0.25 = 4M = 2^2 \cdot 2^{20} =$ 22
4. Suppose that a system uses 16-bit memory words and its memory is built from 32 1M × 8 RAM chips. How many address bits are required to uniquely identify each memory word? 16 of 8 ∴ half addresses = $16 \times 0.5 = 8M = 2^{20} \cdot 2^3$ 8M words
5. Suppose that a system uses 32-bit memory words and its memory is built from 16 1M × 16 RAM chips. How large, in words, is the memory on this system?
6. Suppose we have a 1024-word memory that is 16-way low-order interleaved. What is the size of the memory address offset field?
7. Given a number of 2048 bytes consisting of several 64x8 RAM chips and assuming byte-addressable memory, how many bits are required for chip selection and how many bits are required for each address on chip?

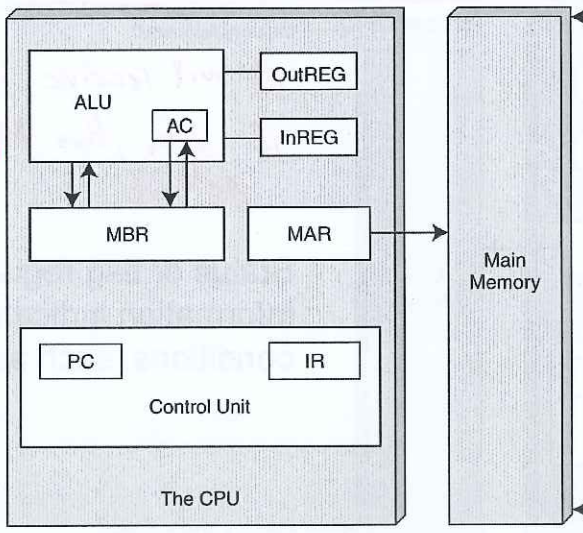
Lecture 14

- Today's topics
  - MARIE
    - Architecture
    - Registers
    - Buses
    - Instruction set Architecture
    - Register transfer notation
    - Instruction processing

MARIE Architecture

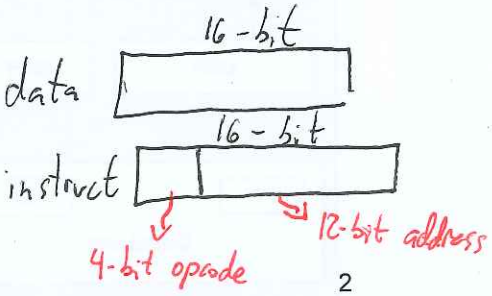
- Binary, two's complement data representation.
- Stored program, fixed word length data and instructions.
- 4K words of word-addressable main memory. → 4K length of 16-bit width
- 16-bit data words. → 16 D-flipflops per column
- 16-bit instructions, 4 for the opcode and 12 for the address.
- A 16-bit arithmetic logic unit (ALU). → process 16-bit at a time
- Seven registers for control and data movement.

- opcode of 4-bit is to detect the particular instruction controller needs to do



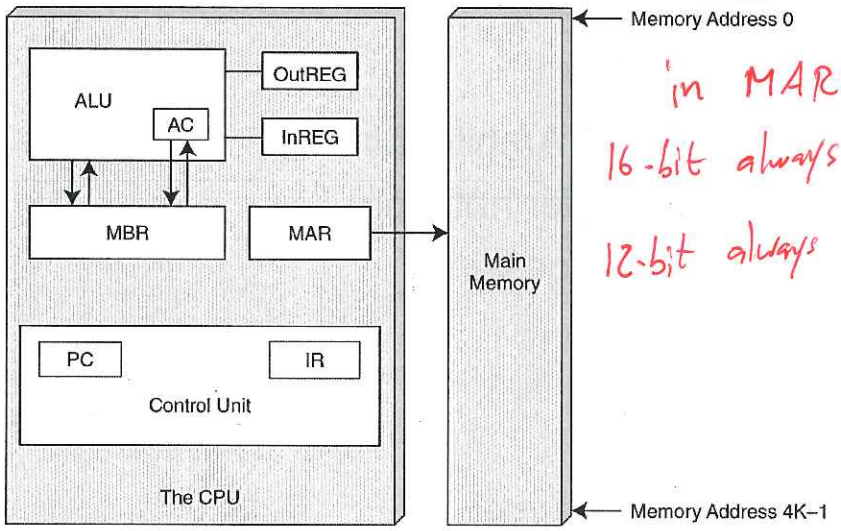
16-bit words for data

$2^4 \cdot 2^{12} = 2^{16}$ → 12 bits for address



MARIE's Registers

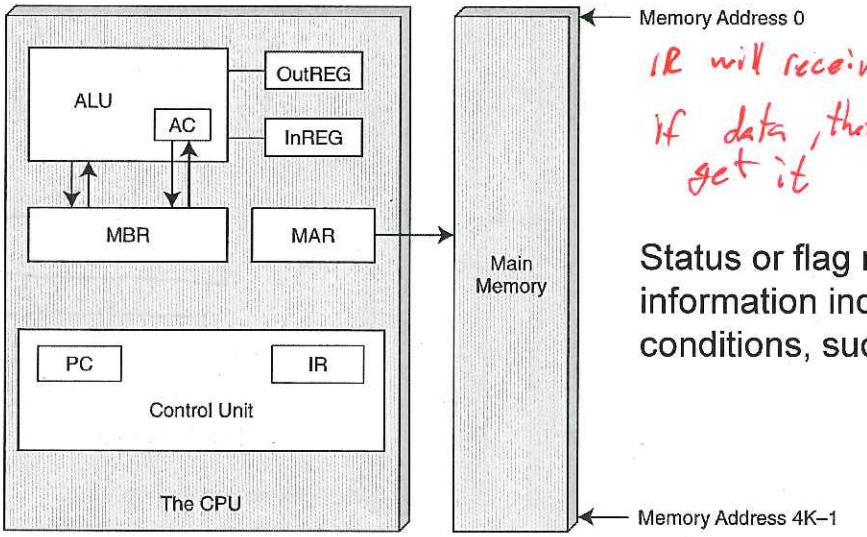
1. Accumulator, AC, a 16-bit register that holds a conditional operator (e.g., "less than") or one operand of a two-operand instruction.
2. Memory address register, MAR, a 12-bit register that holds the memory address of the data being referenced.
3. Memory buffer register, MBR, a 16-bit register that holds the data after its retrieval from, or the data ready to be written to memory.



in MARIE
16-bit always data in memory
12-bit always address

MARIE's Registers

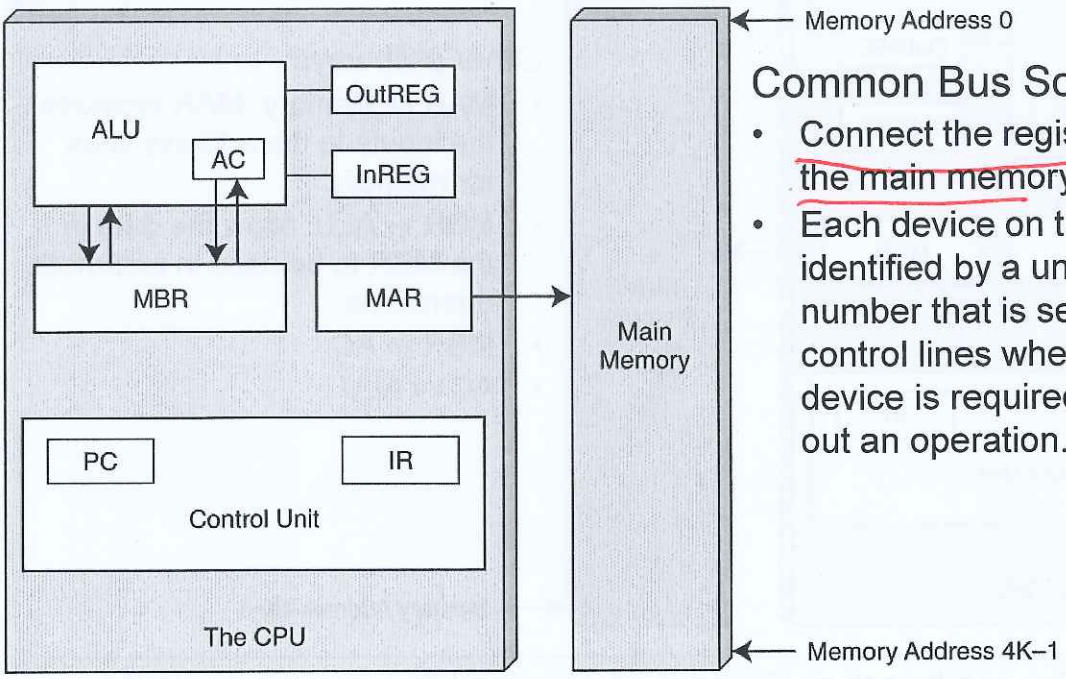
4. Program counter, PC, a 12-bit register that holds the address of the next instruction to be executed.
5. Instruction register, IR, which holds the next instruction to be executed.
6. Input register, InREG, an 8-bit register, holds data read from an input device.
7. Output register, OutREG, an 8-bit register, holds data for the output device.



IR will receive instructions
If data, then MBR will get it

Status or flag register: holds information indicating various conditions, such as overflow.

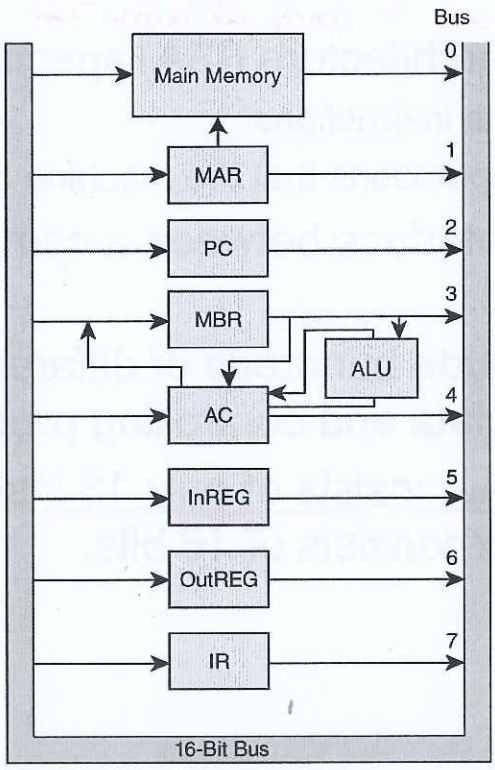
Paths in MARIE



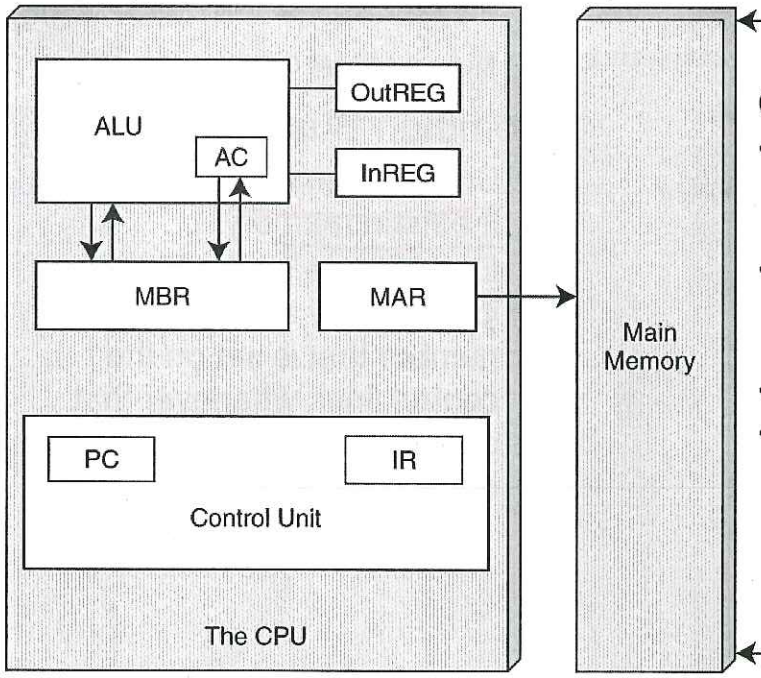
Common Bus Scheme:

- Connect the registers and the main memory.
- Each device on the bus is identified by a unique number that is set on the control lines whenever that device is required to carry out an operation.

Data Path in MARIE



Paths in MARIE



Other pathways:

- MAR → Memory: MAR provides the inputs to the address lines for memory.
- MBR ↔ ALU: allow the data in the MBR to be used in arithmetic operations.
- MBR ↔ AC
- AC ↔ ALU

- Information can be put on the common bus and these pathways in the same clock cycle to allowing these events to take place in parallel.

Instruction Set Architecture

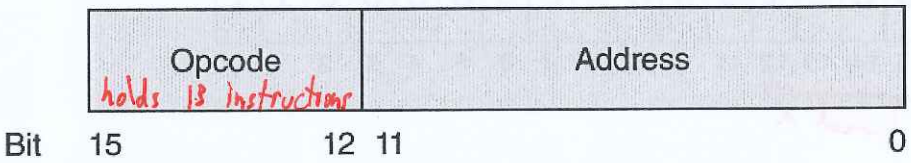
machine specific code to specify all instructions it needs to know

- Instruction set architecture (ISA) specifies different for each machine
  - The format of its instructions
  - The primitive operations that the machine can perform.
- The ISA is an interface between a computer's hardware and its software.
- Some ISAs include hundreds of different instructions for processing data and controlling program execution.
- The MARIE ISA consists of only 13 instructions and each instruction consists of 16 bits.

MARIE Instruction Format

- Each instruction consists of 16 bits.
- The format of a MARIE instruction:

4 bits opcode + 12 bits address



- 12 bits address allows for a maximum of $2^{12} - 1$ memory items.
- Most ISAs consist of instructions for
  - Processing data
  - Moving data
  - Control branches

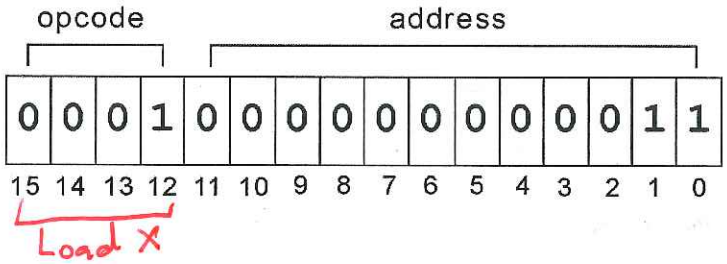
MARIE Instructions

- The fundamental MARIE instructions are:

	Instruction Number			
	Binary	Hex	Instruction	Meaning
Moving data	0001	1	Load X	Load contents of address X into AC.
	0010	2	Store X	Store the contents of AC at address X.
Process data	0011	3	Add X	Add the contents of address X to AC.
	0100	4	Subt X	Subtract the contents of address X from AC.
I/O	0101	5	Input	Input a value from the keyboard into AC.
	0110	6	Output	Output the value in AC to the display.
Control branch	0111	7	Halt	Terminate program.
	1000	8	Skipcond	Skip next instruction on condition.
	1001	9	Jump X	Load the value of X into PC.

- Add X and subt X: the data located at address X is copied into the MBR where it is held until the arithmetic operation is executed.

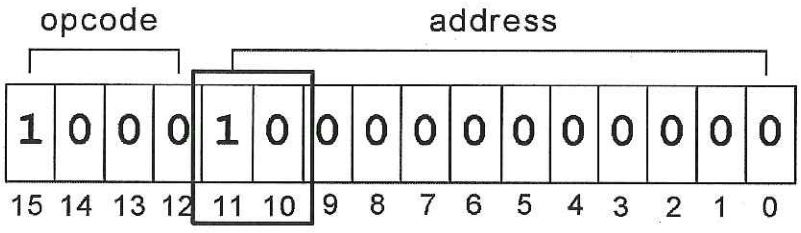
Load Instruction



- Opcode 1 indicates a load instruction.
- Copy the data value found in main memory, address 3, to the AC.

- Machine instruction: binary instruction.
- Assembly language instruction: mnemonic instructions.
- One-to-one mapping between assembly and machine instructions.

Skipcond Instruction



- Opcode 8 indicates a skipcond instruction.
- Bits 11 and 10 are 10, meaning that the next instruction will be skipped if the value in the AC is greater than zero.

Register Transfer Notation

- Each of the instructions actually consists of a sequence of smaller instructions called *microoperations*.
  - Load instruction: loads the contents of the given memory location into the AC register.
    - Minioperations:
      - The address from the instruction must be loaded into MAR.
      - The data in memory at this location must be loaded into MBR.
      - The data in MBR must loaded into the AC.
- The exact sequence of microoperations that are carried out by an instruction can be specified using register transfer language (RTL) or register transfer notation (RTN).
- Notations:
  - $M[X]$: the actual data value stored in memory location X,
  - $\leftarrow$: a transfer of information.

Register Transfer Notation

- The RTL for the **LOAD** X instruction is:

```
MAR ← X
MBR ← M[MAR]
AC ← MBR
```

- The RTL for the Store X instruction is:

```
MAR ← X
MBR ← AC
M[MAR] ← MBR
```

Register Transfer Notation

- The RTL for the **ADD** X instruction is:

```
MAR ← X
MBR ← M[MAR]
AC ← AC + MBR
```

- The RTL for the **Subt** X instruction is:

```
MAR ← X
MBR ← M[MAR]
AC ← AC - MBR
```


Register Transfer Notation

- The RTL for the **Input** instruction is:

```
AC ← InREG
```

- The RTL for the **Output** instruction is:

```
OutREG ← AC
```

- Halt: no operations are performed on registers; the machine simply ceases execution of the program.
- Jump X instruction causes an unconditional branch to the given address, X.

```
PC ← X
```

Register Transfer Notation

- **SKIPCOND** skips the next instruction according to the value of the AC.
- Bits in positions 11 and 10 in the address field are used to determine what comparison to perform on the AC.

```
If IR[11 - 10] = 00 then          [if bit 11 = 0 & bit 10 = 0]
      If AC < 0 then PC ← PC + 1   if less than zero then skip
else If IR[11 - 10] = 01 then     [if bit 11 = 0 & bit 10 = 1]
      If AC = 0 then PC ← PC + 1
else If IR[11 - 10] = 10 then     [if bit 11 = 1 & bit 10 = 0]
      If AC > 0 then PC ← PC + 1
```

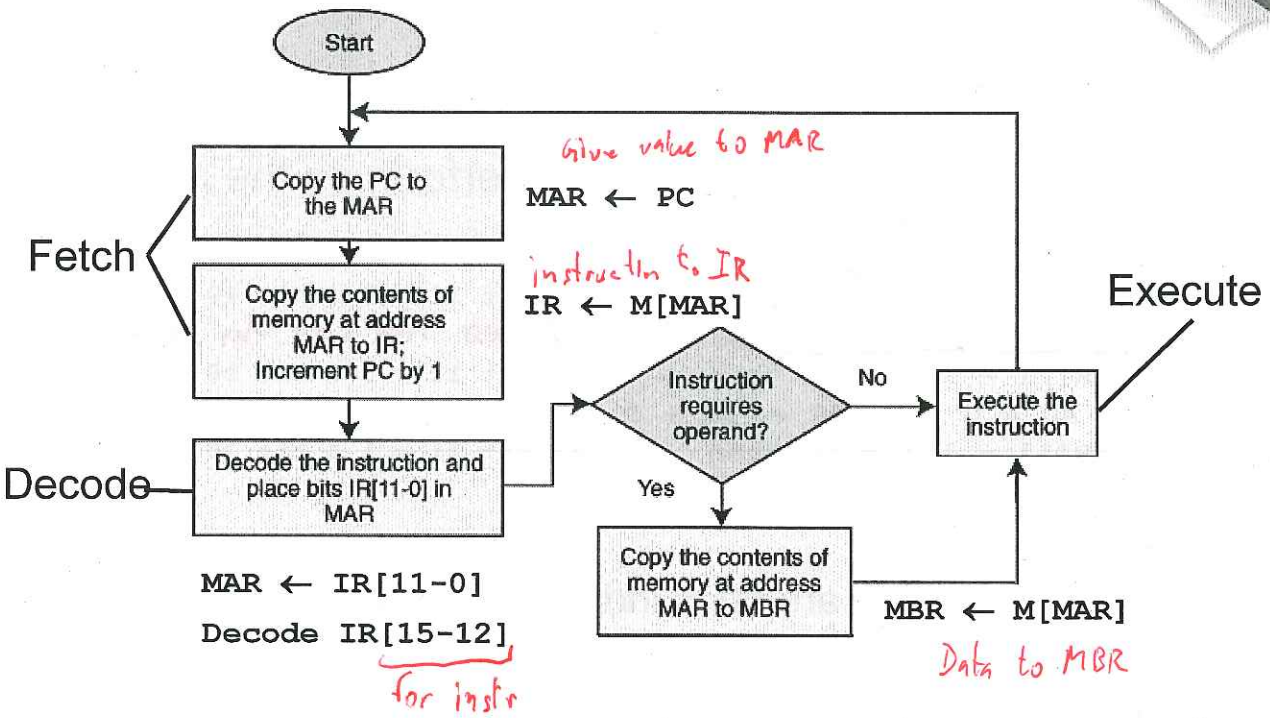
Both bits in 11 and 10 are ones indicate an error condition results.

Instruction Processing

- The fetch-decode-execute cycle is the series of steps that a computer carries out when it runs a program.
- The CPU fetches an instruction from memory, and place it into the IR.
- Once in the IR, it is decoded to determine what needs to be done next.
- If a memory value (operand) is involved in the operation, it is retrieved and placed into the MBR.
- With everything in place, the instruction is executed.

The next slide shows a flowchart of this process.

The Fetch-Decode-Execute Cycle



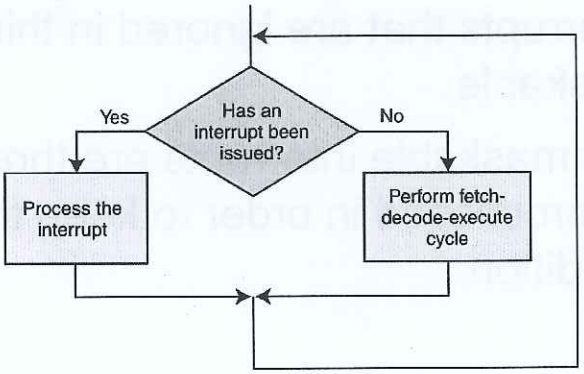
Interrupts and the Instruction Cycle

- All computers provide a way of interrupting the fetch-decode-execute cycle.
- Interrupts occur when:
  - A user break (e.,g., Control+C) is issued
  - I/O is requested by the user or a program
  - A critical error occurs
- Interrupts can be caused by hardware or software.
  - Software interrupts are also called traps.

Fetch-Decode-Execute Cycle with Interrupt Checking

Interrupt processing involves adding another step to the fetch-decode-execute cycle:

- the CPU finishes execution of current instruction and checks, at the beginning of every fetch-decode-execute cycle, to see if an interrupt has been generated.
- Once the CPU acknowledges the interrupt, it must then process the interrupt.

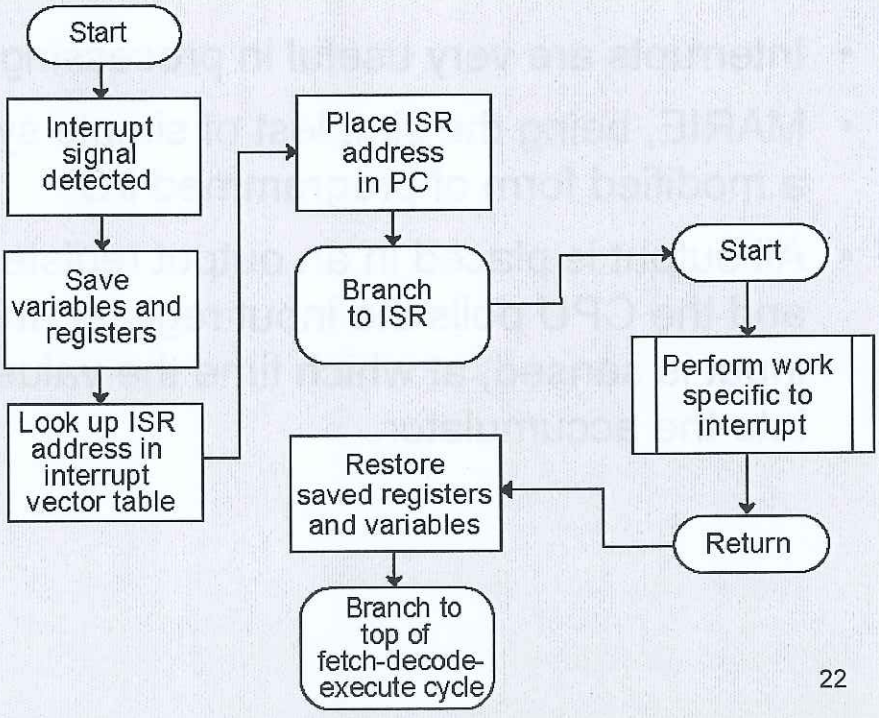


The next slide shows a flowchart of "Process the interrupt."

Interrupt Processing

Interrupt Service Routine (ISR)

The interrupt, along with their associated interrupt service routines (ISRs), are stored in an interrupt vector table.



Interrupt Masking

- For general-purpose systems, it is common to disable all interrupts during the time in which an interrupt is being processed.
  - Typically, this is achieved by setting a bit in the flags register, called interrupt masking.
- Interrupts that are ignored in this case are called maskable.
- Nonmaskable interrupts are those interrupts that must be processed in order to keep the system in a stable condition.

I/O Related Software Interrupt

- Interrupts are very useful in processing I/O.
- MARIE, being the simplest of simple systems, uses a modified form of programmed I/O.
- All output is placed in an output register, OutREG, and the CPU polls the input register, InREG, until input is sensed, at which time the value is copied into the accumulator.

Lecture 15

- Today's lecture
  - MARIE programming
  - Assembler
  - Extending the instruction set

1

MARIE Instructions

- The fundamental MARIE instructions are:

	Instruction Number			
	Binary	Hex	Instruction	Meaning
Moving data	0001	1	Load X	Load contents of address X into AC.
	0010	2	Store X	Store the contents of AC at address X.
Process data	0011	3	Add X	Add the contents of address X to AC.
	0100	4	Subt X	Subtract the contents of address X from AC.
I/O	0101	5	Input	Input a value from the keyboard into AC.
	0110	6	Output	Output the value in AC to the display.
Control branch	0111	7	Halt	Terminate program.
	1000	8	Skipcond	Skip next instruction on condition.
	1001	9	Jump X	Load the value of X into PC.

- Add X and subt X: the data located at address X is copied into the MBR where it is held until the arithmetic operation is executed.

2

Exercise

List the hexadecimal code for the following program (hand assemble it).

3-bit hex address = 12 bit binary

Hex Address	Label	Instruction
100		Load A
101		Add One
102		Jump S1
103	S2,	Add One
104		Store A
105		Halt
106	S1,	Add A
107		Jump S2
108	A,	HEX 0023
109	One,	HEX 0001

4-bit Hex = 16-bit binary data

3

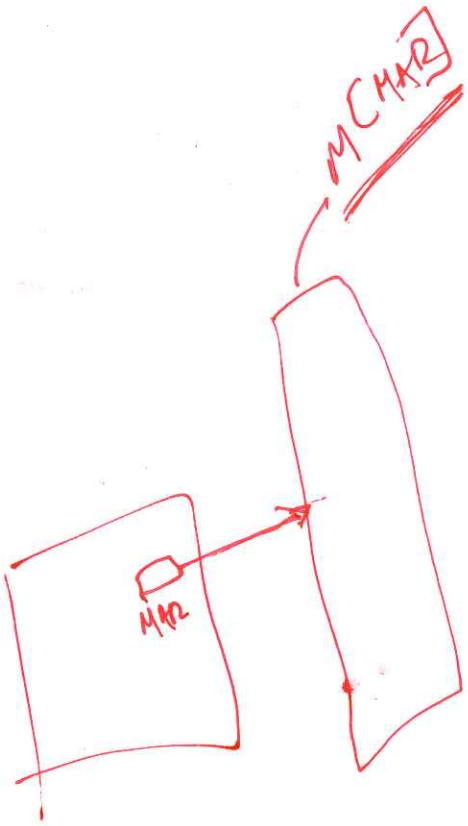
Exercise - Solution

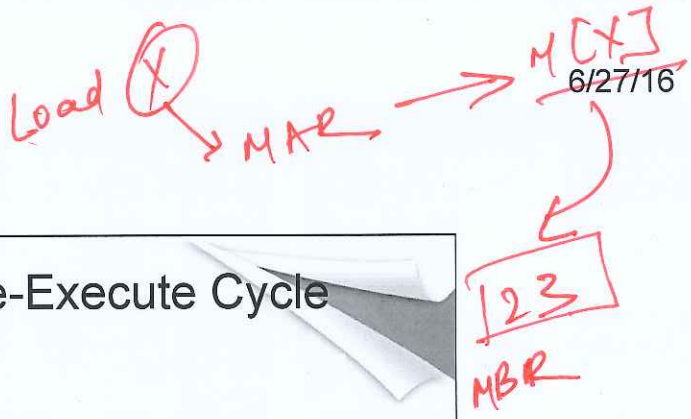
List the hexadecimal code for the following program (hand assemble it).

Load instruction → from Tables
Add instruction →

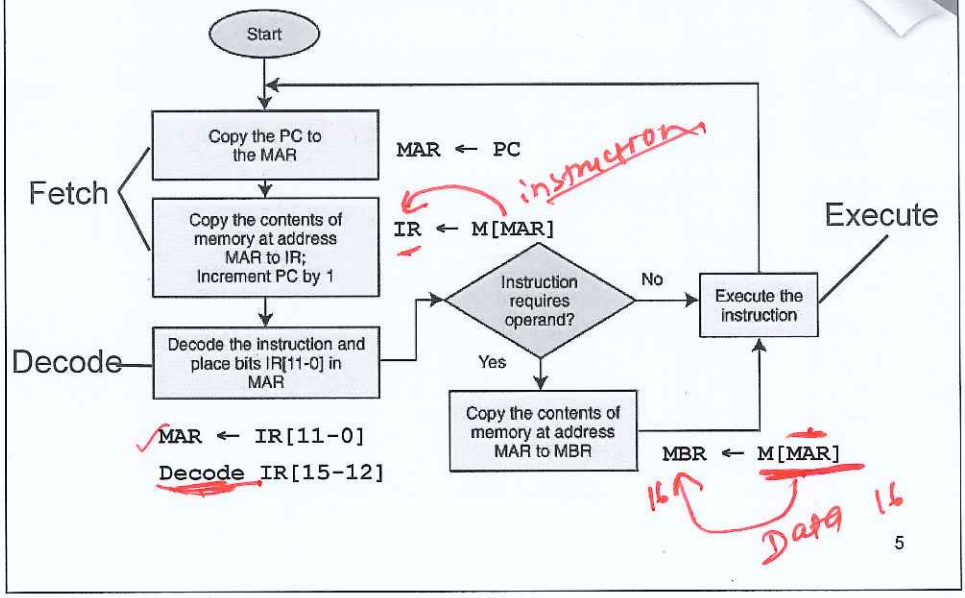
Hex Address	Label	Instruction	
100		Load A	1108
101		Add One	3109
102		Jump S1	9106
103	S2,	Add One	3109
104		Store A	2108
105		Halt	7000
106	S1,	Add A	3108
107		Jump S2	9103
108	A,	HEX 0023	0023
109	One,	HEX 0001	0001

4


MCMR
MR



The Fetch-Decode-Execute Cycle



5

A Simple Program

- Consider the simple MARIE program given below: add two numbers together and the sum in memory.
- The instructions stored at addresses 0x100 – 0x106 (hex):

Address	Instruction	Binary Contents of Memory Address	Hex Contents of Memory
100	Load 104	0001000100000100	1104
101	Add 105	0011000100000101	3105
102	Store 106	0100000100000110	4106
103	Halt	0111000000000000	7000
104	0023	0000000000100011	0023
105	FFE9	1111111111101001	FFE9
106	0000	0000000000000000	0000

6

A Simple Program

- What happens inside the computer when the program runs.
- This is the **LOAD 104** instruction:

@ 100

Step	RTN	PC	IR	MAR	MBR	AC
(initial values)		100	------	------	------	------
Fetch	MAR ← PC	100	------	100	------	------
	IR ← M[MAR]	100	1104	100	------	------
	PC ← PC + 1	101	1104	100	------	------
Decode	MAR ← IR[11–0]	101	1104	104	------	------
Ctrl Signal	(Decode IR[15–12])	101	1104	104	------	------
Get operand	MBR ← M[MAR]	101	1104	104	0023	------
Execute	AC ← MBR	101	1104	104	0023	0023

Data in 104 = 0023

Add 0023 & FFE9

7

A Simple Program

- Our second instruction is **ADD 105**:

Step	RTN	PC	IR	MAR	MBR	AC
(initial values)		101	1104	104	0023	0023
Fetch	MAR ← PC	101	1104	101	0023	0023
	IR ← M[MAR]	101	3105	101	0023	0023
	PC ← PC + 1	102	3105	101	0023	0023
Decode	MAR ← IR[11–0]	102	3105	105	0023	0023
	(Decode IR[15–12])	102	3105	105	0023	0023
Get operand	MBR ← M[MAR]	102	3105	105	FFE9	0023
Execute	AC ← AC + MBR	102	3105	105	FFE9	000C

result in Accumulator

8

A Simple Program

- The third instruction is **`Store 106.`**
- After the program is done, the binary content of location 0x106 change to 0x000C (12 in decimal).

Step	RTN	PC	IR	MAR	MBR	AC
(initial values)		102	3105	105	FFE9	000C
Fetch	MAR ← PC	102	3105	102	FFE9	000C
	IR ← M[MAR]	102	2106	102	FFE9	000C
	PC ← PC + 1	103	2106	102	FFE9	000C
Decode	MAR ← IR[11-0]	103	2106	106	FFE9	000C
	(Decode IR[15-12])	103	2106	106	FFE9	000C
Get operand	(not necessary)	103	2106	106	FFE9	000C
Execute	MBR ← AC	103	2106	106	000C	000C

9

A Discussion on Assemblers

- Mnemonic instructions, such as **`LOAD 104`**, are easy for humans to write and understand.
- They are impossible for computers to understand.
- Assemblers translate assembly language that are comprehensible to humans into the machine language that is comprehensible to computers.
- The assembler reads a source file (assembly program) and produces an object file (the machine code).

10

Creating an Object Program Files

- Assemblers create an object program file from mnemonic *source code* in two passes.
- During the first pass, the assembler assembles as much of the program as it can, while it builds a symbol table that contains memory references for all symbols in the program.
- During the second pass, the instructions are completed using the values from the symbol table.

11

Creating an Object Program Files - Example

- The first pass, creates a symbol table and the partially-assembles instructions.
- After the second pass, the assembly is complete.

Address	Instruction
100	Load X
101	Add Y
102	Store Z
103	Halt
104 X,	DEC 35
105 Y,	DEC -23
106 Z,	HEX 0000

1	X
3	Y
2	Z
7000	

X	104
Y	105
Z	106

1104
3105
2106
7000
0023
FFE9
0000

Hex

12

Extending Our Instruction Set

- So far, all of the MARIE instructions that we have discussed use a direct addressing *mode*.
- This means that the address of the operand is explicitly stated in the instruction.
- It is often useful to employ a indirect addressing, where the address of the address of the operand is given in the instruction.
  - If you have ever used pointers in a program, you are already familiar with indirect addressing.
- Six new instructions using indirect addressing.

13

I = indirect addressing

LOADI and STOREI

pointer → LOADI X (load indirect): go to address X, use the value at X as the actual address of the operand to load into the AC.

- STOREI X (Store indirect): go to address X, use the value at X as the destination address for storing the value in the AC.
- In RTL:

```
MAR ← X
MBR ← M[MAR]
MAR ← MBR
MBR ← M[MAR]
AC ← MBR
```

memory

LOADI X

```
MAR ← X
MBR ← M[MAR]
MAR ← MBR
MBR ← AC
M[MAR] ← MBR
```

STOREI X

14

ADDI and JUMPI

- The **ADDI** instruction is a combination of **LOADI X** and **ADD X**.
- **JUMPI** instruction: go to address X, use the value at X as the actual address of the location to jump to.
- In RTL:

```
MAR ← X
MBR ← M[MAR]
MAR ← MBR
MBR ← M[MAR]
AC ← AC + MBR
```

ADDI X

```
MAR ← X
MBR ← M[MAR]
PC  ← MBR
```

JUMPI X

15

JNS and CLEAR

- JNS: The jump-and-store instruction gives us limited subroutine functionality: store the PC at address X and jump to X+1
- CLEAR: set AC to zero.
- In RTL:

```
MBR ← PC
MAR ← X
M[MAR] ← MBR
MBR ← X
AC ← 1
AC ← AC + MBR
PC ← AC
```

JNS X

```
AC ← 0
```

CLEAR

16

MARIE Programming Examples

Example: using a loop to add five numbers:

```
100 |       LOAD Addr      |  10E |       SKIPCOND 000
101 |       STORE Next     |  10F |       JUMP Loop
102 |       LOAD Num       |  110 |       HALT
103 |       SUBT One       |  111 |Addr   HEX 117
104 |       STORE Ctr      |  112 |Next   HEX 0
105 |Loop   LOAD Sum       |  113 |Num    DEC 5
106 |       ADDI Next      |  114 |Sum    DEC 0
107 |       STORE Sum      |  115 |Ctr    HEX 0
108 |       LOAD Next      |  116 |One    DEC 1
109 |       ADD One        |  117 |       DEC 10
10A |       STORE Next     |  118 |       DEC 15
10B |       LOAD Ctr       |  119 |       DEC 2
10C |       SUBT One       |  11A |       DEC 25
10D |       STORE Ctr      |  11B |       DEC 30
```

→ when ctr < 0, skip loop

ctr - -

MARIE Programming Examples

Example: use of an if/else construct to allow for selection.

if $X = Y$ then

$X = X \times 2$

else

```
          Address  Instruction  Operand   Comments
If,       100      Load         X         /Load the first value
          101      Subt         Y         /Subtract value of Y and store result in AC
          102      Skipcond     400       /If AC = 0, skip the next instruction
          103      Jump         Else      /Jump to the Else part if AC is not equal to 0
Then,     104      Load         X         /Reload X so it can be doubled
          105      Add          X         /Double X
          106      Store        X         /Store the new value
          107      Jump         Endif     /Skip over Else part to end of If
Else,     108      Load         Y         /Start the Else part by loading Y
          109      Subt         X         /Subtract X from Y
          10A      Store        Y         /Store Y - X in Y
Endif,    10B      Halt                   /Terminate program (it doesn't do much!)
X,        10C      Dec          12        /Load the loop control variable
Y,        10D      Dec          20        /Subtract one from the loop control variable
```

skip to 104 if T

→ only true when X = Y

Exercise 1 - Solution

Consider the MARIE program below.

Hex Address	Label	Instruction	
100	Start,	LOAD A	1108
101		ADD B	3109
102		STORE D	210B
103		CLEAR	A000
104		OUTPUT	6000
105		ADDI D	B10B
106		STORE B	2109
107		HALT	7000
108	A,	HEX 00FC	00FC
109	B,	DEC 14	000E
10A	C,	HEX 0108	0108
10B	D, A+B	HEX 0000	0000

SYMBOL TABLE

Symbol	Location
A	108
B	109
C	10A
D	10B
Start	100

AC = 0108 upon termination

010A → 10A

A+B = 00FC + 000E = 010A

a) List the hexadecimal code for each instruction.
b) Draw the symbol table.
c) What is the value stored in the AC when the program terminates?

21

Exercise 2

Write the following code segment in MARIE's assembly language:

```
if x <= y then
    y = y + 1;
else if x != z
    then y = y – 1;
else z = z + 1;
```

22

Exercise 2 - Solution

```
        ORG 100
If1,    Load     X       /Load X
        Subt     Y       /Subtract Y, store result in AC
        Skipcond 800     /If AC>0 (X>Y), skip the next instruction
        Jump     Then1   /Jump to Then1 because X<=Y
Else1,  Load     X       /End up here if X>Y; need to test if X=Z
        Subt     Z       /Compare X to Z
        Skipcond 400     /If AC=0 (X=Z), skip the next instruction
        Jump     Then2   /Jump to Else2
        Load     Z       /Execute Z=Z+1
        Add      One
        Store    Z
        Jump     Done    /Done with if statement
Then1,  Load     Y       /End up here if X<=Y, so execute Y=Y+1
        Add      One
        Store    Y
        Jump     Done    /Done with if statement
Then2,  Load     Y
        Subt     One
        Store    Y
Done,   Halt             /terminate program

X,      Dec      ?       /X has starting value, not given in problem
Y,      Dec      ?       /Y has starting value, not given in problem
One,    Dec      1       /Use as a constant
```

```
if x <= y then
   y = y + 1;
else if x != z
   then y = y – 1;
else z = z + 1;
```

PC = X+1

MARIE Programming Examples

Example: a simple subroutine to double the value stored at X.

```
        100  Load   X      /Load the first number to be doubled
        101  Store  Temp   /Use Temp as a parameter to pass value to Subr
        102  JnS    Subr   /Store return address, jump to procedure
        103  Store  X      /Store first number, doubled
        104  Load   Y      /Load the second number to be doubled
        105  Store  Temp   /Use Temp as a parameter to pass value to Subr
        106  JnS    Subr   /Store return address, jump to procedure
        107  Store  Y      /Store second number, doubled
        108  Halt          /End program
X,      109  Dec    20
Y,      10A  Dec    48
Temp,   10B  Dec    0
Subr,   10C  Hex    0      /Store return address here
        10D  Clear         /Clear AC as it was modified by JnS
        10E  Load   Temp   /Actual subroutine to double numbers
        10F  Add    Temp   /AC now holds double the value of Temp
        110  JumpI  Subr   /Return to calling code
             END
```

Exercise 1

Consider the MARIE program below.

Hex Address	Label	Instruction
100	Start,	LOAD A
101		ADD B
102		STORE D
103		CLEAR
104		OUTPUT
105		ADDI D
106		STORE B
107		HALT
108	A,	HEX 00FC
109	B,	DEC 14
10A	C,	HEX 0108
10B	D,	HEX 0000

Load 108 = 1108

3109

a) List the hexadecimal code for each instruction.

b) Draw the symbol table.

c) What is the value stored in the AC when the program terminates?

20

Lecture 16

- Today's topics:
  - MARIE Instruction Decoding and Control
  - Hardwired control
  - Micro-programmed control

Instruction Decoding

- The control unit
  - Driven by the processor's clock
  - Decodes the instruction
  - Creates control signals
- The microoperations given by each register transfer language define the operation of MARIE's control unit.
- A sequence of "control" steps for each instruction.
- The control unit generates a set of signals for each control step that execute the appropriate microoperation.

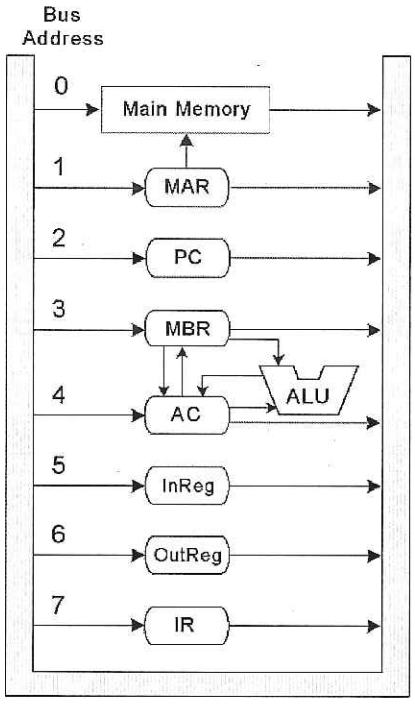
RTL →

```
MAR ← X
MBR ← M[MAR]
AC ← AC + MBR
```

Register Selection

- Each of MARIE's registers and main memory have a unique address along the datapath.
- The addresses take the form of signals issued by the control unit.

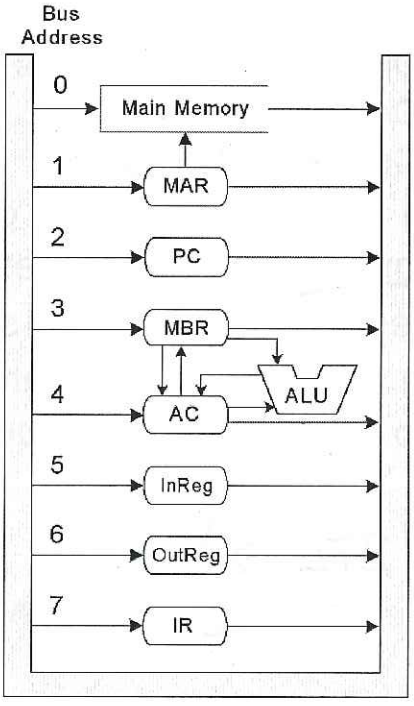
How many signal lines does MARIE's control unit need?



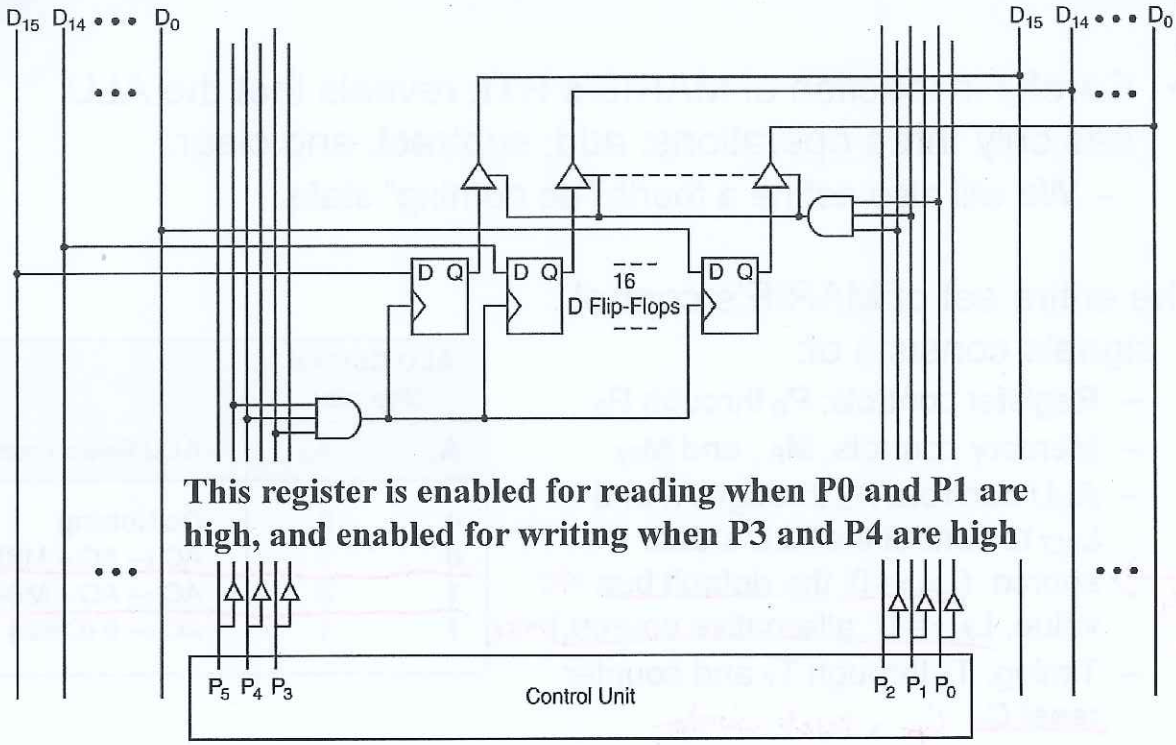
Register Selection

- Two sets of three signals.
  - P_2, P_1, P_0: controls reading from memory or a register
  - P_5, P_4, P_3: controls writing to memory or a register.

The next slide shows a close up view of MARIE's MBR.



Connection of MBR to Datapath



Q: what is the connection of PC to the datapath and control lines?

MARIE's Full Instruction Set

How many types of operations of ALU can perform?

Opcode	Instruction	RTN
0000	JnS *X*	MBR ← PC MAR ← *X* M[MAR] ← MBR MBR ← *X* AC ← 1 AC ← AC + MBR PC ← AC
0001	Load *X*	MAR ← *X* MBR ← M[MAR] AC ← MBR
0010	Store *X*	MAR ← *X*, MBR ← AC M[MAR] ← MBR
0011	Add *X*	MAR ← *X* MBR ← M[MAR] AC ← AC + MBR
0100	Subt *X*	MAR ← *X* MBR ← M[MAR] AC ← AC - MBR
0101	Input	AC ← InREG
0110	Output	OutREG ← AC
0111	Halt	
1000	Skipcond	If IR[11-10] = 00 then If AC < 0 then PC ← PC + 1 Else If IR[11-10] = 01 then If AC = 0 then PC ← PC + 1 Else If IR[11-10] = 10 then If AC > 0 then PC ← PC + 1
1001	Jump *X*	PC ← IR[11-0]
1010	Clear	AC ← 0
1011	AddI *X*	MAR ← X MBR ← M[MAR] MAR ← MBR MBR ← M[MAR] AC ← AC + MBR
1100	JumpI *X*	MAR ← *X* MBR ← M[MAR] PC ← MBR
1101	LoadI *X*	MAR ← X MBR ← M[MAR] MAR ← MBR MBR ← M[MAR] AC ← MBR
1110	StoreI *X*	MAR ← X MBR ← M[MAR] MAR ← MBR MBR ← AC M[MAR] ← MBR

ALU Control

- Careful inspection of MARIE's RTL reveals that the ALU has only three operations: add, subtract, and clear.
  - We will also define a fourth "do nothing" state.

The entire set of MARIE's control signals consists of:

- Register controls: P_0 through P_5,
- Memory controls: M_R , and M_W.
- ALU controls: A_0 through A_1 and L_{ALT} to control the ALU's data source. (L_{ALT}=0: the default bus value, L_{ALT}=1 : alternative source.)
- Timing: T_0 through T_7 and counter reset C_r

To pick AC or MBR as source

AC MBR C_r = reset counter

ALU Control Signals		ALU Response
A_1	A_0	
0	0	Do Nothing
0	1	AC ← AC + MBR
1	0	AC ← AC – MBR
1	1	AC ← 0 (Clear)

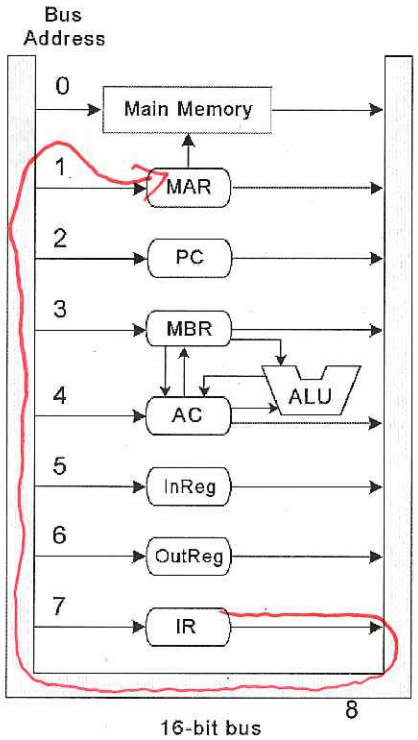
ADD SUB

MARIE's Add Instruction Control

- Consider MARIE's Add instruction. Its RTL is:

```
MAR ← X
MBR ← M[MAR]
AC ← AC + MBR
```

- After an Add instruction is fetched, the address, X, is in the rightmost 12 bits of the IR, which has a datapath address of 7.
- X is copied to the MAR, which has a datapath address of 1.
- Thus we need to raise signals P_0, P_1, and P_2 to read from the IR, and signal P_3 to write to the MAR.



7 to 1

Control Signal Sequence

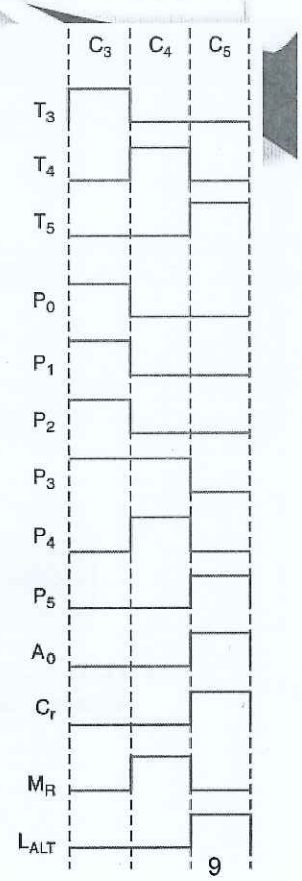
- The signal sequence for MARIE's Add instruction:

P_4 P_3 P_2 P_1 P_0 T_3 : MAR ← X

P_4 P_3 T_4 M_R : MBR ← M[MAR]

C_r A_0 P_5 T_5 L_{ALT} : AC ← AC + MBR [Reset counter]

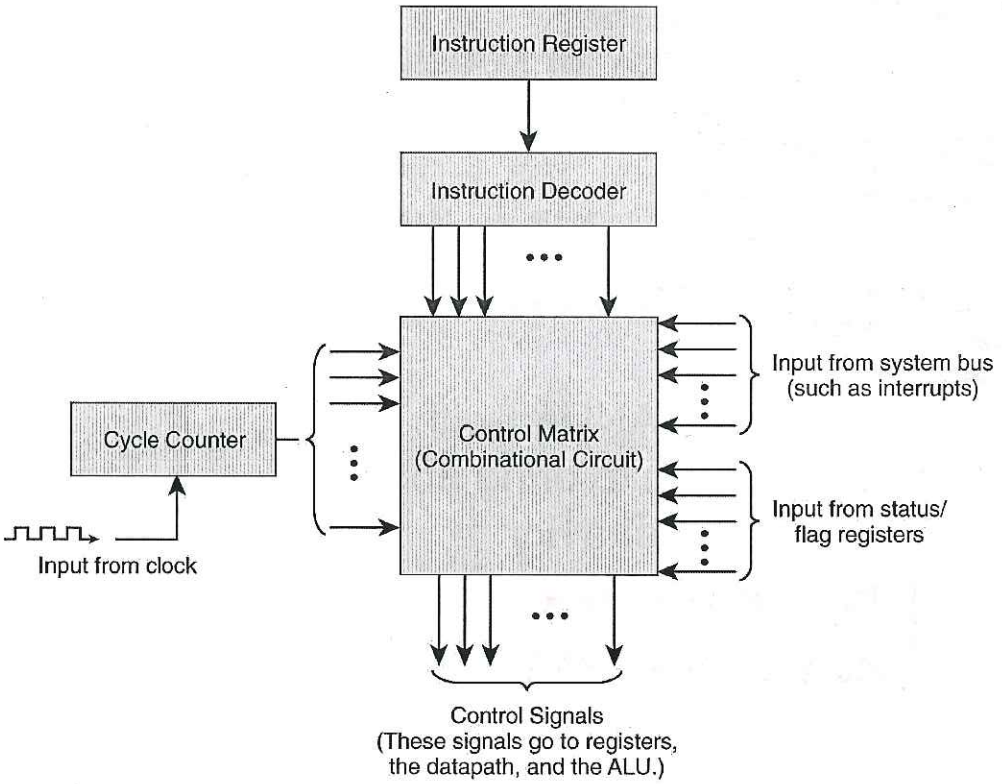
- These signals are ANDed with combinational logic to bring about the desired machine behavior.



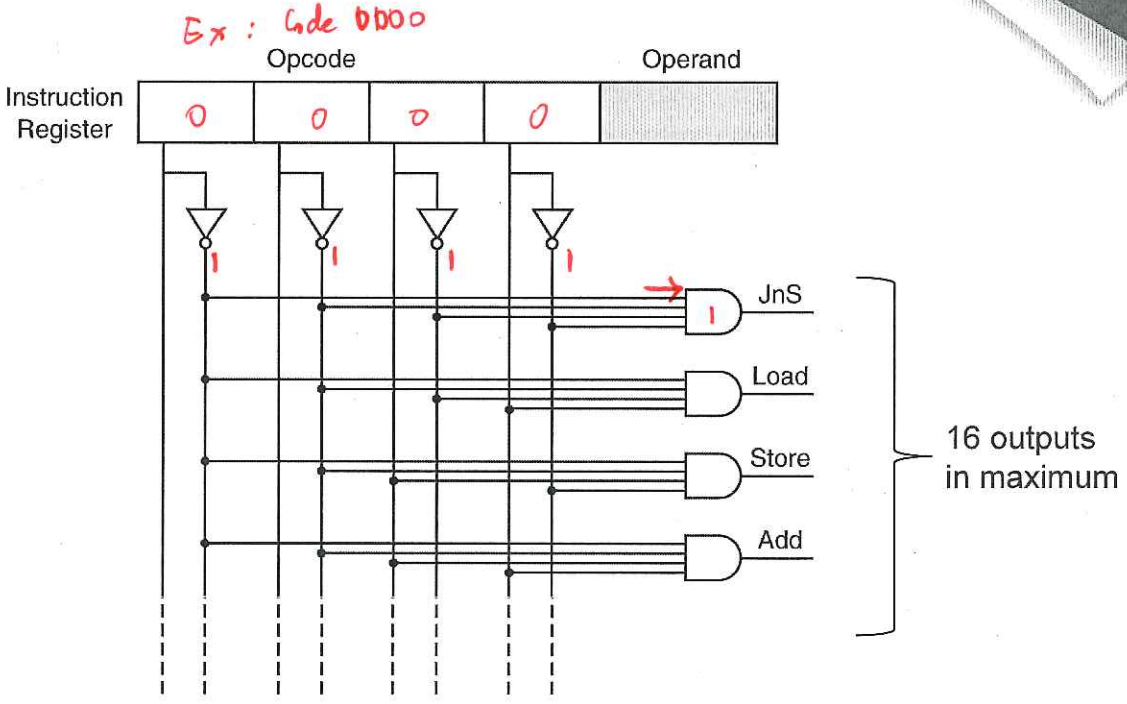
Control Unit Implementation

- Two general ways control unit implementation, hardwired control and microprogrammed control.
- Hardwired control:
  - The bit pattern of machine instruction in the IR is decoded by combinational logic.
  - The decoder output works with the control signals of the current system state to produce a new set of control signals.
- Microprogrammed control:
  - Employs software consisting of microinstructions that carry out an instruction's microoperations.
- Note that the signal pattern just described is the same whether our machine used hardwired or microprogrammed control.

Hardwired Control

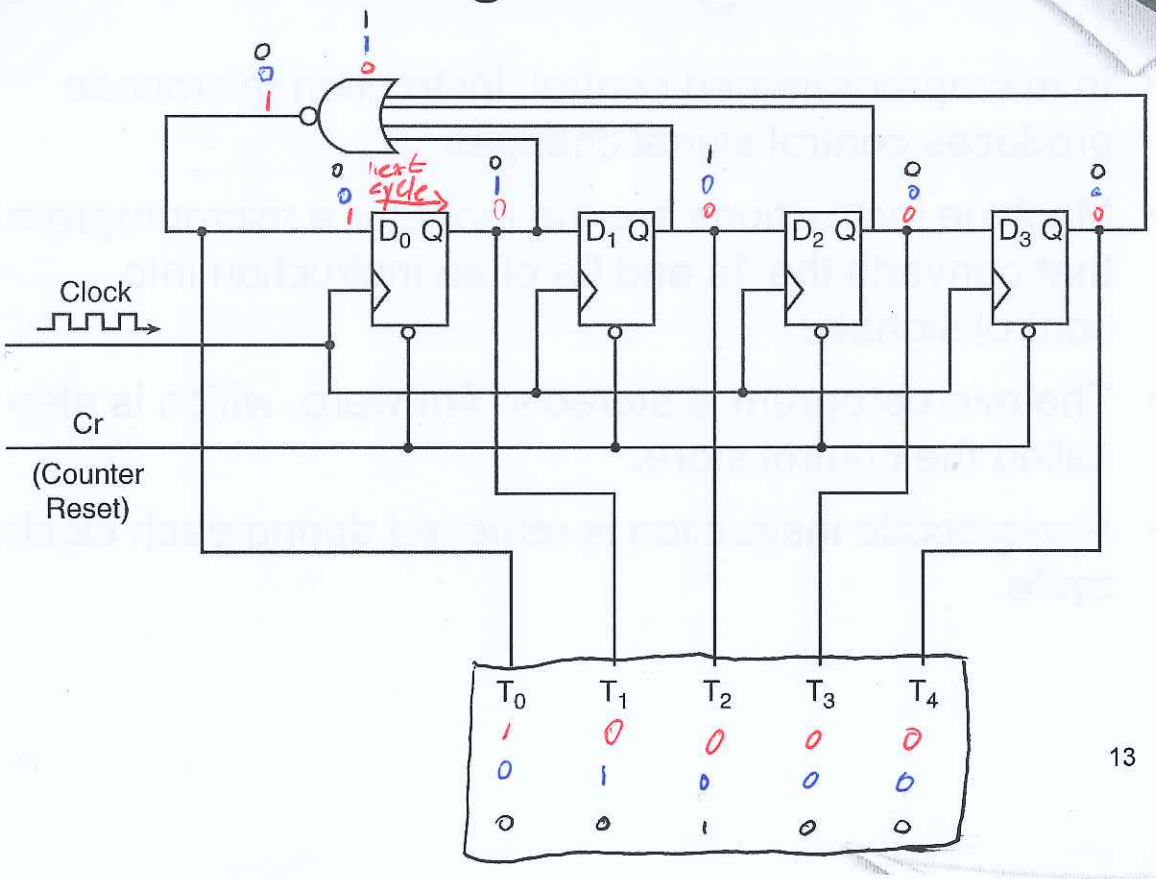


Hardwired Control

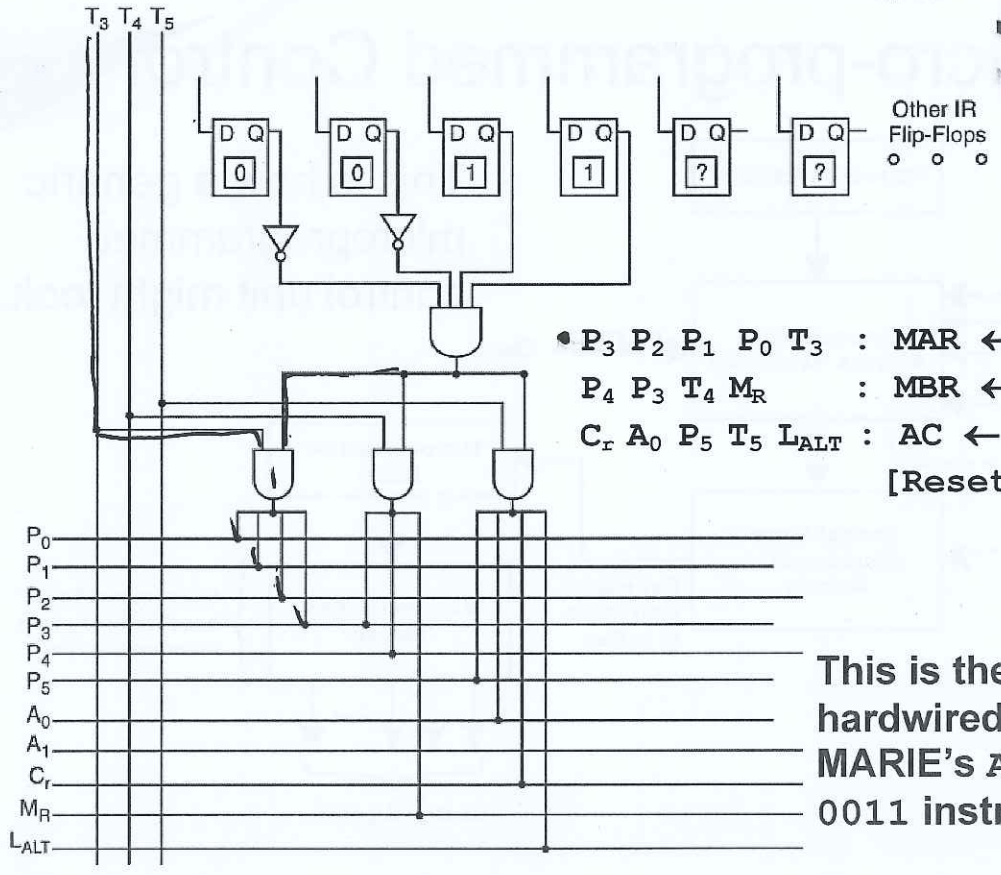


MARIE's instruction decoder. (Partial.)

Ring Counter



13



- $P_3\ P_2\ P_1\ P_0\ T_3$: MAR ← X
  $P_4\ P_3\ T_4\ M_R$: MBR ← M[MAR]
  $C_r\ A_0\ P_5\ T_5\ L_{ALT}$: AC ← AC + MBR
  [Reset counter]

This is the hardwired logic for MARIE's Add = 0011 instruction.

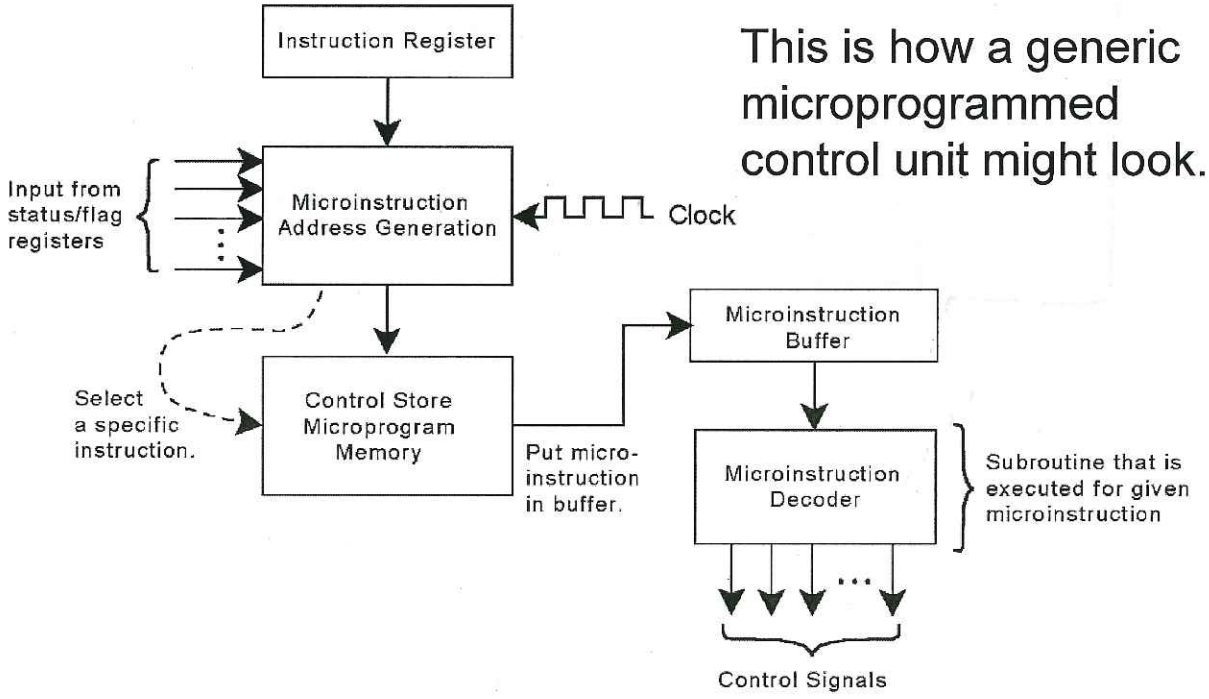
14

Q: Load and JumpI instruction? Draw the timing diagram for load and jumpI

Micro-programmed Control

- In microprogrammed control, instruction microcode produces control signal changes.
- Machine instructions are the input for a microprogram that converts the 1s and 0s of an instruction into control signals.
- The microprogram is stored in firmware, which is also called the control store.
- A microcode instruction is retrieved during each clock cycle.

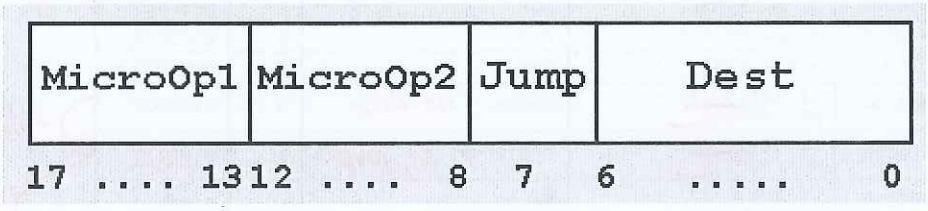
Micro-programmed Control



This is how a generic microprogrammed control unit might look.

Microinstruction Format

- If MARIE were microprogrammed, the microinstruction format might look like this:



- **MicroOp1** and **MicroOp2** contain binary codes for each microoperation specified in the RTN.
- **Jump** is a single bit indicating that the value in the **Dest** field is a valid address and should be placed in the microsequencer. is PC in hardwired
- Control then branches to the address found in the Dest field.

Microoperation Codes

- The table below contains MARIE's microoperation codes along with the corresponding RTL:

MicroOp Code	Microoperation	MicroOp Code	Microoperation
00000	NOP	01101	MBR ← M[MAR]
00001	AC ← 0	01110	OutREG ← AC
00010	AC ← MBR	01111	PC ← IR[11-0]
00011	AC ← AC - MBR	10000	PC ← MBR
00100	AC ← AC + MBR	10001	PC ← PC + 1
00101	AC ← InREG	10010	If AC = 00
00110	IR ← M[MAR]	10011	If AC > 0
00111	M[MAR] ← MBR	10100	If AC < 0
01000	MAR ← IR[11-0]	10101	If IR[11-10] = 00
01001	MAR ← MBR	10110	If IR[11-10] = 01
01010	MAR ← PC	10111	If IR[11-10] = 10
01011	MAR ← X	11000	If IR[15-12] = MicroOp2[4-1]
01100	MBR ← AC		

Jump Table

- The first four lines are the fetch-decode-execute cycle.
- The remaining lines are the beginning of a jump table.

Address	MicroOp1	MicroOp2	Jump	Dest
0000000	MAR ← PC	NOP	0	0000000
0000001	IR ← M[MAR]	NOP	0	0000000
0000010	PC ← PC + 1	NOP	0	0000000
0000011	MAR ←IR[11-0]	NOP	0	0000000
0000100	If IR[15-12] = MicroOP2[4-1]	00000 ? no, false	1	0100000
0000101	If IR[15-12] = MicroOP2[4-1]	00010 → Yes, jump	1	0100111
0000110	If IR[15-12] = MicroOP2[4-1]	00100	1	0101010
0000111	If IR[15-12] = MicroOP2[4-1]	00110	1	0101100
0001000	If IR[15-12] = MicroOP2[4-1]	01000	1	0101111
...	...	...	...	...
...	...	...	...	...
0101010	MAR ←X	MBR ← AC	0	0000000
0101011	M[MAR]← MBR	NOP	1	0000000
0101100	MAR ←X	NOP	0	0000000
0101101	MBR ←M[MAR]	NOP	0	0000000
0101110	AC ←AC + MBR	NOP	1	0000000
0101111	MAR ←MAR	NOP	0	0000000
...	...	...	...	...

Regular fetch decode

do each & go to next if no jump

false so next

Execute, then back to 00000

Jump Table Exercise

Address	MicroOp 1	MicroOp 2	Jump	Dest
0000000	01010	00000	0	0000000
0000001	00110	00000	0	0000000
0000010	10001	00000	0	0000000
0000011	01111	00000	0	0000000
0000100	11000	00000	1	0100000
0000101	11000	00010	1	0100111
0000110	11000	00100	1	0101010
0000111	11000	00110	1	0101100
0001000	11000	01000	1	0101111

Q: write the microcode for the jump table for the MARIE instructions for Jump X, Clear, and AddI X. (Use all 1s for the Destination value.)

OP1 vs OP2 ?

Address	MicroOp1	MicroOp2	Jump	Dest
0001001	11000	01001 Jump	1	1111111
0001010	11000	01010 Clear	1	1111111
0001011	11000	01011 ADDI	1	1111111

Exercise

Using Figure 4.23 as a guide, write the binary microcode for MARIE's Load instruction. Assume that the microcode begins at instruction line number 0110000_2.

LOAD
MAR ← X
MBR ← M[MAR]
AC ← MBR

MicroOp Code	Microoperation	MicroOp Code	Microoperation
00000	NOP	01101	MBR ← M[MAR]
00001	AC ← 0	01110	OutREG ← AC
00010	AC ← MBR	01111	PC ← IR[11-0]
00011	AC ← AC - MBR	10000	PC ← MBR
00100	AC ← AC + MBR	10001	PC ← PC + 1
00101	AC ← InREG	10010	If AC = 00
00110	IR ← M[MAR]	10011	If AC > 0
00111	M[MAR] ← MBR	10100	If AC < 0
01000	MAR ← IR[11-0]	10101	If IR[11-10] = 00
01001	MAR ← MBR	10110	If IR[11-10] = 01
01010	MAR ← PC	10111	If IR[11-10] = 10
01011	MAR ← X	11000	If IR[15-12] = MicroOp2[4-1]
01100	MBR ← AC		

Exercise - Solution

MicroOp Code	Microoperation	MicroOp Code	Microoperation
00000	NOP	01101 (2)	MBR ← M[MAR]
00001	AC ← 0	01110	OutREG ← AC
00010 (3)	AC ← MBR	01111	PC ← IR[11-0]
00011	AC ← AC - MBR	10000	PC ← MBR
00100	AC ← AC + MBR	10001	PC ← PC + 1
00101	AC ← InREG	10010	If AC = 00
00110	IR ← M[MAR]	10011	If AC > 0
00111	M[MAR] ← MBR	10100	If AC < 0
01000	MAR ← IR[11-0]	10101	If IR[11-10] = 00
01001	MAR ← MBR	10110	If IR[11-10] = 01
01010	MAR ← PC	10111	If IR[11-10] = 10
01011 (1)	MAR ← X	11000	If IR[15-12] = MicroOp2[4-1]
01100	MBR ← AC		

Address	MicroOp 1	MicroOp 2	Jump	Dest
0110000	1 01011	00000	0	0000000
0110001	2 01101	00000	0	0000000
0110010	3 00010	00000	1 →	0000000

Hardware vs. Micro-programmed Control

- Microinstructions are fetched, decoded, and executed in the same manner as regular instructions.
- This extra level of instruction interpretation is what makes microprogrammed control slower than hardwired control.
- The advantages of microprogrammed control are that it can support very complicated instructions and only the microprogram needs to be changed if the instruction set changes (or an error is found).

Lecture 17

- Today's Lecture
  - Instruction formats
    - Little versus big endian
    - Internal storage in the CPU: stacks vs. registers
    - Number of operands and instruction length
    - Expanding opcodes

Instruction sets

Instruction sets are differentiated by the following:

- Number of bits per instruction.
  - Instruction length: 16, 32 and 64 bits are the most common.
- Stack-based or register-based.
  - Operand storage: register or stack?
- Number of explicit operands per instruction.
  - Zero, one, two, and three are the most common.
- Operand location.
  - Combinations of operands allowed per instruction: register-to-register, register-to-memory, or memory-to-memory?
- Types of operations
  - Arithmetic, data transfer, floating pointing operation, logic operation, etc.
- Type and size of operands.
  - Operands can be addresses, numbers, or characters.

Byte Ordering

- Little endian machines store the most significant byte at the least significant byte.
- Big endian machines store the most significant byte first (at the lower address).
- As an example, suppose we have the hexadecimal number 0x12345678.

MSB

Address →	00	01	10	11
Big Endian	12	34	56	78
Little Endian	78	56	34	12

Ex: what if we have to split a number into multiple memory sections. How do I read it? Bottom? Top? This is why this system is used

Instruction Formats Exercise

- A larger example: A computer uses 32-bit integers. The values 0xABCD1234, 0x00FE4321, and 0x10 would be stored sequentially in memory, starting at address 0x200 as below.

Address	Big Endian	Little Endian
0x200	AB	34
0x201	CD	12
0x202	12	CD
0x203	34	AB
0x204	00	21
0x205	FE	43
0x206	43	FE
0x207	21	00
0x208	00	10
0x209	00	00
0x20A	00	00
0x20B	10	00

Internal Storage in CPU

- Stack architecture
  - A stack is used to execute instructions, and the operands are found on top of the stack.
  - A stack can not be access randomly.
  - A stack is a bottleneck during execution.
- Accumulator architecture
  - One operand implicitly in the accumulator
  - One operand is in memory, creating lots of bus traffic.
- General purpose register (GPR) architecture
  - Faster than accumulator architecture.
  - Results in longer instructions.

General Purpose Architecture

- Most systems today are GPR systems.
- There are three types depending on where the operands are located.
  - Memory-memory where two or three operands may be in memory.
  - Register-memory where at least one operand must be in a register.
  - Load-store architectures require data to be moved into registers before any operations on these data are performed.
    - Most architectures today.
    - SPARC, MIPS, Alpha, PowerPC
- The number of operands and the number of available registers has a direct affect on instruction length.

Number of Operands & Instruction Length

- Instruction length
  - Fixed length: waste space but is fast and results in better performance when instruction-level pipelining is used.
  - Variable length: more complex to decode but saves storage space.
- Number of operands: most common instruction formats include zero, one, two, or three operands.
  - Zero: OPCODE only Ex: ADD - assumes operands are in stack
  - One: OPCODE + 1 address (usually a memory address) ADD A → Get A & add
  - Two: OPCODE + 2 addresses (usually registers, or one register and one memory address) ADD A B
  - Three: OPCODE+3 addresses (usually registers, or combinations of registers and memory) ADD A B C
- Note: all architectures have a limit on the maximum number of operands allowed per instruction.

Number of Operands & Instruction Length

More on stack-based architectures:

- Most instructions consist of opcodes only = Zero based
- Some special instructions have one or two operand.
  - **LOAD** and **STORE** instructions require a single memory address operand.
  - **PUSH** and **POP** operations involve only the stack's top element.
    - Push X: place the data value found at memory location X onto the stack.
    - Pop X: removes the top element in the stack and stores it at location X.
  - Binary instructions (e.g., **ADD**, **MULT**) use the top two items on the stack.
    - ADD: the CPU adds the top two elements of the stack, popping them both and then pushing the sum onto the top of the stack.

Stack Arithmetic Expression

- Infix notation, such as: Z = X ⊕ Y.
- Stack arithmetic uses postfix notation: Z = XY+.
  - This is also called reverse Polish notation, in honor of its Polish inventor, Jan Lukasiewicz (1878 - 1956).
- The principal advantage of postfix notation is that parentheses are not used.
- For example, the infix expression,

  Z = (X × Y) + (W × U),

  becomes:

  Z = X Y × W U × +

  in postfix notation.

Infix & Postfix Expressions – Exercise 1

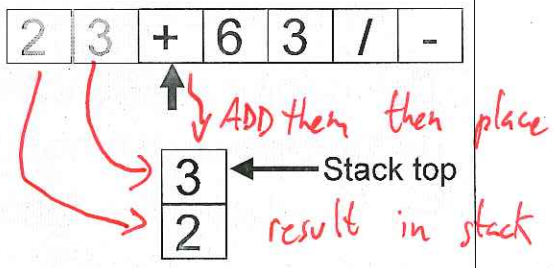
Convert the infix expression (2+3) - 6/3 to postfix:

2 3+ - 6/3	The sum 2 + 3 in parentheses takes precedence; we replace the term with 2 3 +.
2 3+ - 6 3 /	The division operator takes next precedence; we replace 6/3 with 6 3 /.
2 3+ 6 3 / -	The quotient 6/3 is subtracted from the sum of 2 + 3, so we move the - operator to the end.

Infix & Postfix Expressions – Exercise 2

- Example: Use a stack to evaluate the postfix expression 2 3 + 6 3 / - :

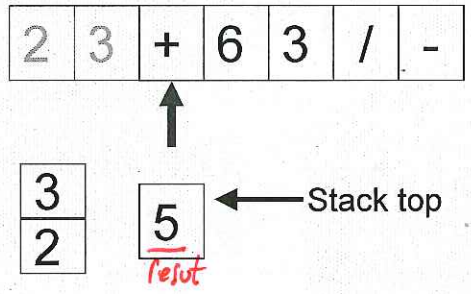
Scanning the expression from left to right, push operands onto the stack, until an operator is found



Infix & Postfix Expressions – Exercise 2

- Example: Use a stack to evaluate the postfix expression 2 3 + 6 3 / - :

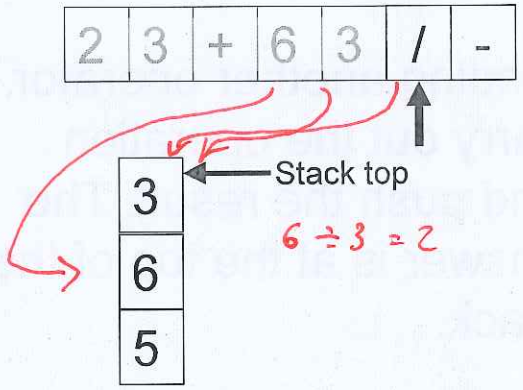
Pop the two operands and carry out the operation indicated by the operator. Push the result back on the stack.



Infix & Postfix Expressions – Exercise 2

- Example: Use a stack to evaluate the postfix expression 2 3 + 6 3 / - :

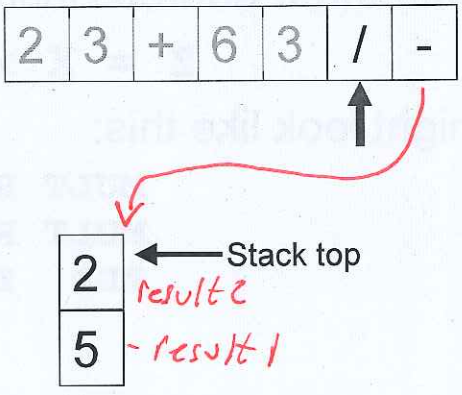
Push operands until another operator is found.



Infix & Postfix Expressions – Exercise 2

- Example: Use a stack to evaluate the postfix expression 2 3 + 6 3 / - :

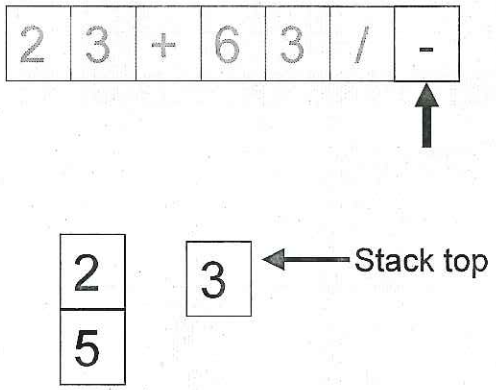
Carry out the operation and push the result.



Infix & Postfix Expressions – Exercise 2

- Example: Use a stack to evaluate the postfix expression 2 3 + 6 3 / - :

Finding another operator, carry out the operation and push the result. The answer is at the top of the stack.



Infix & Postfix Expressions – Exercise 3

- Let's see how to evaluate an infix expression using different instruction formats.
- With three-address instructions, the infix expression,

$$Z = X \times Y + W \times U$$

might look like this:

```
MULT R1,X,Y
MULT R2,W,U
ADD  Z,R1,R2
```

Infix & Postfix Expressions – Exercise 3

- With two-address instructions, the infix expression,

  Z = X × Y + W × U

  might look like this:

```
LOAD  R1,X     R1 = X
MULT  R1,Y     R1 = R1 * Y
LOAD  R2,W     R2 = W
MULT  R2,U     R2 = R2 * U
ADD   R1,R2    R1 = R1 + R2
STORE Z,R1     Z = R1
```


Infix & Postfix Expressions – Exercise 3

- With one-address instructions (as in MARIE), the infix expression,

  Z = X × Y + W × U

  looks like this:

```
LOAD X       AC = X
MULT Y       AC = AC * Y
STORE TEMP   TEMP = AC
LOAD W       AC = W
MULT U       AC = AC * U
ADD TEMP     AC = AC + TEMP
STORE Z      Z = AC
```

Note: Must assume that a register is implied as the destination for the result of the instruction.

accumulator assumed

Infix & Postfix Expressions – Exercise 3

- In a stack ISA (no operand), the postfix expression,

  Z = X Y × W U × +

  might look like this:

```
PUSH X
PUSH Y
MULT
PUSH W
PUSH U
MULT
ADD
PUSH Z
```

Should say POP

Storing?

Would this program require more execution time than the corresponding (shorter) program that we saw in the 3-address ISA?

of Operands and # of Instructions

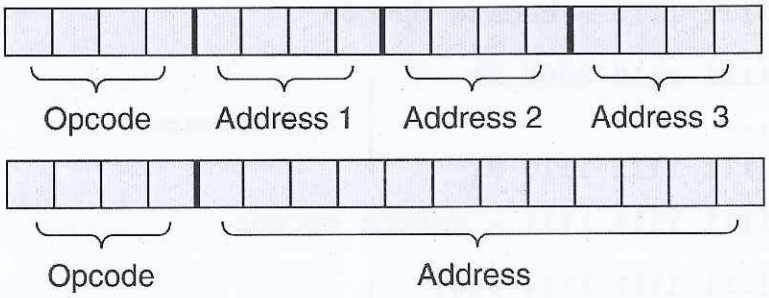
- The number of instructions required to execute the desired code in this example
  - Thee-address operands: 3 instructions
  - Two-address operands: 6 instructions
  - One-address operands: 7 instructions
  - Zero-address operand: 8 instructions
- Reduce the number of operands allowed per instruction, the number of instructions required to execute the desired code increase.

Expanding Opcodes

- In any instruction set, not all instructions require the same number of operands.
- Operations that require no operands, such as `HALT`, necessarily waste some space when fixed-length instructions are used.
- One way to recover some of this space is to use expanding opcodes.

Expanding Opcodes

- A system has 16 registers and 4K of memory.
- We need 4 bits to access one of the registers. We also need 12 bits for a memory address.
- If the system is to have 16-bit instructions, we have two choices for our instructions:



Expanding Opcodes & Escape Opcodes

- If the length of the opcode is allowed to vary, we could create a very rich instruction set.
- How to determine when the instruction should be interpreted as having a 4-bit, 8-bit, 12-bit or 16-bit opcode?
  - ➢ Escape opcode to indicate which format should be use.
- Suppose we wish to encode the following instructions:
  - 15 instructions with three addresses
  - 14 instructions with two addresses
  - 31 instructions with one addresses
  - 16 instructions with zero addresses

Expanding Opcodes & Escape Opcodes

```
0000 R1   R2   R3
...                          } 15 three-address codes
1110 R1   R2   R3
1111 - escape opcode
1111 0000 R1   R2
...                          } 14 two-address codes
1111 1101 R1   R2
1111 1110 - escape opcode
1111 1110 0000 R1
...                          } 31 one-address codes
1111 1111 1110 R1
1111 1111 1111 - escape opcode
1111 1111 1111 0000
...                          } 16 zero-address codes
1111 1111 1111 1111
```

Expanding Opcodes & Escape Opcodes

```
If ( leftmost four bits != 1111 ) {
        Execute appropriate three-address instruction }
else if ( leftmost seven bits != 1111 111 ) {
        Execute appropriate two-address instruction }
else if ( leftmost twelve bits != 1111 1111 1111 ) {
        Execute appropriate one-address instruction }
else {
        Execute appropriate zero-address instruction
}
```


Expanding Opcodes & Escape Opcodes

Question: how do we know if the instruction set we want is possible when using expanding opcode?

- Determine if we have enough bits to create the desired number of bit patterns.
- Once we determine it is possible, we can create the appropriate escape opcodes for the instruction set.

Expanding Opcodes & Escape Opcodes – Example 1

- 12-bit instruction and 3-bit register operand
  - 4 instructions with three registers: $4 \times 2^3 \times 2^3 \times 2^3 = 2048$
  - 255 instructions with one register: $255 \times 2^3 = 2040$
  - 16 instructions with zero registers: 16 bit patterns
  - In total: 2048 + 2040 + 16 = 4194 > $2^{12} = 4096$
  - An expanding opcode instruction set cannot be designed to meet the specified requirement.

Expanding Opcodes & Escape Opcodes – Example 2

- Example: Given 8-bit instructions, is it possible to allow the following to be encoded?
  - 3 instructions with two 3-bit operands.
  - 2 instructions with one 4-bit operand.
  - 4 instructions with one 3-bit operand.

We need:

$3 \times 2^3 \times 2^3$ = 192 bit patterns for the 3-bit operands
2×2^4 = 32 bit patterns for the 4-bit operands
4×2^3 = 32 bit patterns for the 3-bit operands.

Total: 256 bits = 2^8 bits. We have an exact match .

Expanding Opcodes & Escape Opcodes – Example 2

```
00 xxx xxx
01 xxx xxx
10 xxx xxx
11 - escape opcode
1100 xxxx
1101 xxxx
1110 - escape opcode
1111 - escape opcode
11100 xxx
11101 xxx
11110 xxx
11111 xxx
```

} 3 instructions with two 3-bit operands

} 2 instructions with one 4-bit operand

} 4 instructions with one 3-bit operand

Lecture 18

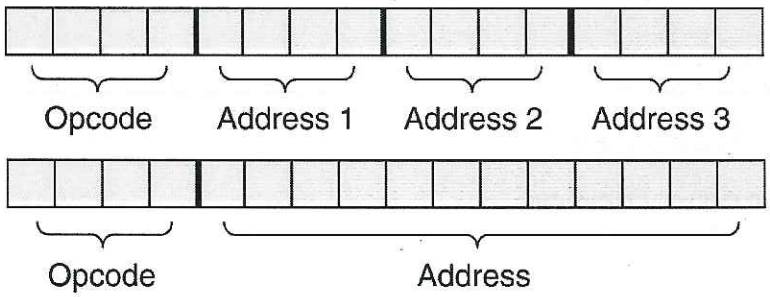
- Last Lecture
  - Instruction formats
    - Little versus big endian
    - Internal storage in the CPU: stacks vs. registers
    - Number of operands and instruction length
    - Expanding opcodes
- Today's Topic
  - Instruction types
  - Addressing modes

Expanding Opcodes

- In any instruction set, not all instructions require the same number of operands.
- Operations that require no operands, such as **`HALT`**, necessarily waste some space when fixed-length instructions are used.
- One way to recover some of this space is to use expanding opcodes.

Expanding Opcodes

- A system has 16 registers and 4K of memory.
- We need 4 bits to access one of the registers. We also need 12 bits for a memory address.
- If the system is to have 16-bit instructions, we have two choices for our instructions:



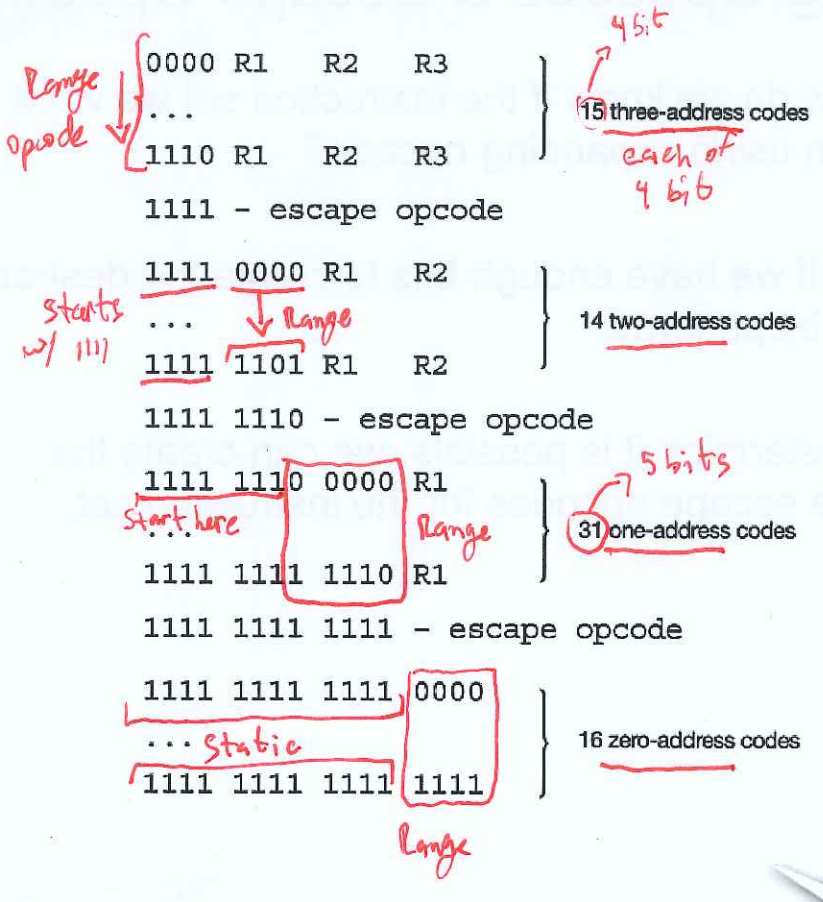
Expanding Opcodes & Escape Opcodes

- If the length of the opcode is allowed to vary, we could create a very rich instruction set.

- How to determine when the instruction should be interpreted as having a 4-bit, 8-bit, 12-bit or 16-bit opcode?
  - Escape opcode to indicate which format should be use.

- Suppose we wish to encode the following instructions:
  - 15 instructions with three addresses
  - 14 instructions with two addresses
  - 31 instructions with one addresses
  - 16 instructions with zero addresses

Expanding Opcodes & Escape Opcodes



Expanding Opcodes & Escape Opcodes

```
If ( leftmost four bits != 1111 ) {
        Execute appropriate three-address instruction }
else if ( leftmost seven bits != 1111 111 ) {
        Execute appropriate two-address instruction }
else if ( leftmost twelve bits != 1111 1111 1111 ) {
        Execute appropriate one-address instruction }
else {
        Execute appropriate zero-address instruction
}
```

Expanding Opcodes & Escape Opcodes

Question: how do we know if the instruction set we want is possible when using expanding opcode?

- Determine if we have enough bits to create the desired number of bit patterns.
- Once we determine it is possible, we can create the appropriate escape opcodes for the instruction set.

Expanding Opcodes & Escape Opcodes – Example 1

- 12-bit instruction and 3-bit register operand
  - 4 instructions with three registers: $4 \times 2^3 \times 2^3 \times 2^3 = 2048$
  - 255 instructions with one register: $255 \times 2^3 = 2040$
  - 16 instructions with zero registers: 16 bit patterns
  - In total: 2048 + 2040 + 16 = 4194 > $2^{12} = 4096$
  - An expanding opcode instruction set cannot be designed to meet the specified requirement.

Expanding Opcodes & Escape Opcodes – Example 2

- Example: Given 8-bit instructions, is it possible to allow the following to be encoded?
  - 3 instructions with two 3-bit operands.
  - 2 instructions with one 4-bit operand.
  - 4 instructions with one 3-bit operand.

We need:

$3 \times 2^3 \times 2^3 = 192$ bit patterns for the 3-bit operands
$2 \times 2^4 = 32$ bit patterns for the 4-bit operands
$4 \times 2^3 = 32$ bit patterns for the 3-bit operands.

Total: 256 bits = 2^8 bits. We have an exact match.

Expanding Opcodes & Escape Opcodes – Example 2

```
00 xxx xxx
01 xxx xxx
10 xxx xxx
11 - escape opcode
1100 xxxx
1101 xxxx
1110 - escape opcode
1111 - escape opcode
11100 xxx
11101 xxx
11110 xxx
11111 xxx
```

} 3 instructions with two 3-bit operands

} 2 instructions with one 4-bit operand

} 4 instructions with one 3-bit operand

Instruction Types

- Data movement
  - Move data from memory to registers, from registers to registers, from registers to memory.
  - Examples: Move, load, store, push, pop, exchange.
- Input/Output:
  - Input: transfers data from a device or port to either memory or a register.
  - Output: transfers data from a register or memory to a specific port or a device.
- Arithmetic Operations
  - Add, subtract, multiply, divide, increment, decrement, negate.
- Control transfer → PC value change
  - Alter the normal sequence of program execution.
  - Branches, skips, procedure calls, returns, and program termination.

Instruction Types (Cont')

- Boolean logic
  - Perform boolean operations.
  - AND, NOT, OR, TEST, and COMPARE
- Bit manipulation
  - Setting or extracting individual bits within a given data word.
  - Arithmetic and logic shift and rotate (a circle shift).
  - Arithmetic shift, commonly used to multiply or divide by 2, treat data as signed two's complement numbers.
  - Logical shift instructions shift bits to either the left or to the right by a specific number of bits, shifting in zeros on the opposite end.
  - Rotate instructions are simply shift instructions that shift in the bits that are shift out.
- Special purpose
  - String processing, high-level language support, protection, flag control, word/byte conversion, cache management, register access, address calculation, no-ops.

Instruction Set Orthogonality

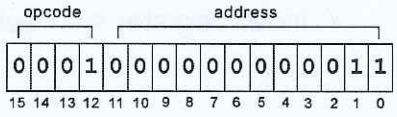
- Each instruction should perform unique function without duplicating any other instructions. Means no redundant instructions.
- Instruction set must be consistent: the operand/opcode relationship cannot be restricted (there are no special registers for particular instructions.)

Addressing Modes

- Addressing modes specify where an operand is located.
- They can specify a constant, a register, or a memory location.
- Addressing modes:
  - Immediate addressing is where the data is part of the instruction.
    - Example: `addi $t0, $t1, 64`

1. Immediate addressing

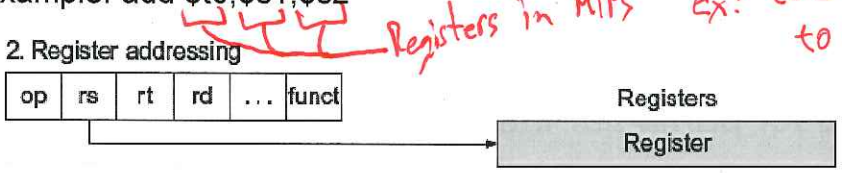
  - Direct addressing is where the address of the data is given in the instruction.
    - Example: Load 3



  - Indirect addressing gives the address of the address of the data in the instruction.
    - Example: loadi 3

Addressing Modes

- Addressing modes specify where an operand is located.
- They can specify a constant, a register, or a memory location.
- Addressing modes:
  - Register addressing is where the data is located in a register.
    - Example: add $t0,$s1,$s2



Registers in MIPS

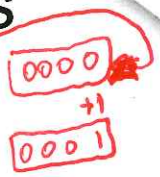
Ex: t0 = s1 + s2
t0 = 5 + 6
t0 = 11

  - Register indirect addressing uses a register to store the address of the address of the data. (Effective PC address = contents of register 'reg')
    - Example: add AX,[BX]

pointer. value @ this location should be treated as address

Addressing Modes

a[i]



- Addressing modes (con't)
  - Indexed addressing: use an index register to store an offset, which is added to the address in the operand to determine the effective address of the data.
    - Index + signed displacement
  - Useful to access elements of an array
    - Displacement ==> points to the beginning of the array
    - Index register ==> selects an element of the array (array index)

Addressing Modes

- Addressing modes (con't)
  - Based addressing is similar except that a base address register is used instead of an index register.
    - base + signed displacement
  - Useful to access fields of a structure or record
    - » Base register ==> points to the base address of the structure
    - » Displacement ==> relative offset within the structure

The difference between these two is that
- an index register holds an offset relative to the address given in the instruction,
- a base register holds a base address where the address field represents an offset from this base.

further explanation

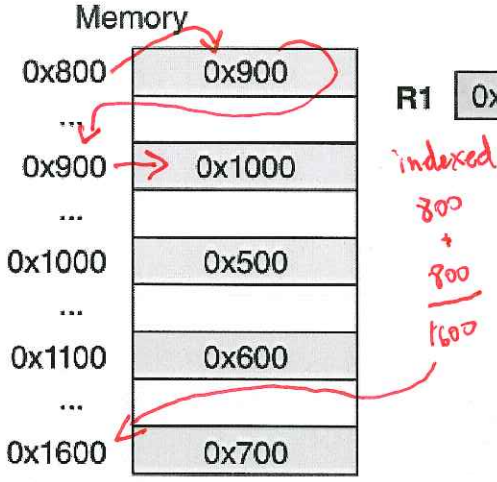
Addressing Modes

- Addressing modes (con't)
  - Stack addressing: the operand is assumed to be on top of the stack.
  - Variations to these addressing modes:
    - Indirect indexed: use both indirect and indexed addressing.
    - Base/offset: offset + a base register + the specified instruction.
    - Self-relative: the address of the operand is an offset from the current instruction.
    - Auto increment – decrement: automatically increment or decrement the register used.

Test Question

Addressing Mode – Exercise 1

- For the instruction shown, what value is loaded into the accumulator for each addressing mode?



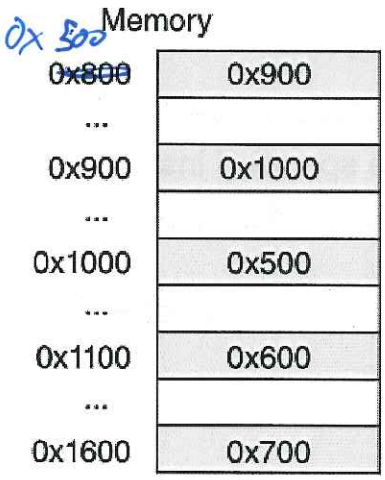
LOAD 800

Mode	Value Loaded into AC
Immediate	800
Direct	900
Indirect	1000
Indexed	~~1600~~ 700

index always add with referenced Register

Addressing Mode – Exercise 1 Solution

- For the instruction shown, what value is loaded into the accumulator for each addressing mode?



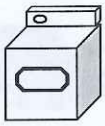
LOAD 800

Mode	Value Loaded into AC
Immediate	0x800
Direct	0x900
Indirect	0x1000
Indexed	0x700

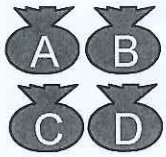
Pipelining Example

- Laundry Example: Three Stages

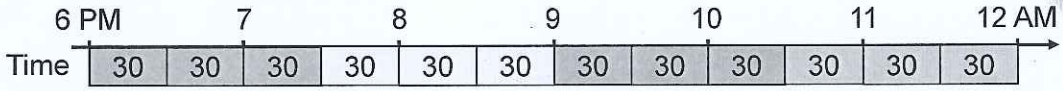
1. Wash dirty load of clothes
2. Dry wet clothes
3. Fold and put clothes into drawers

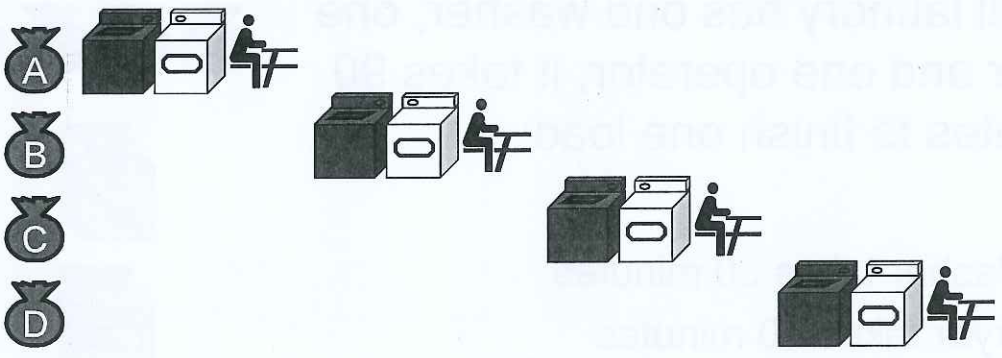




- Each stage takes 30 minutes to complete
- Four loads of clothes to wash, dry, and fold



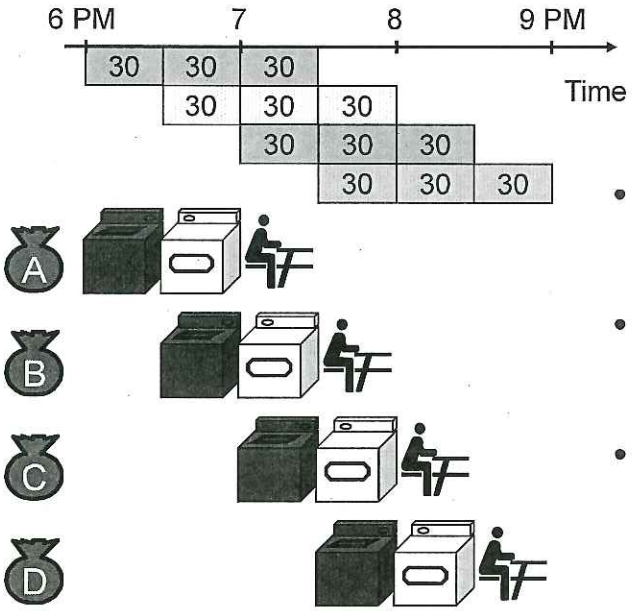
Sequential Laundry





- Sequential laundry takes 6 hours for 4 loads
- Intuitively, we can use pipelining to speed up laundry

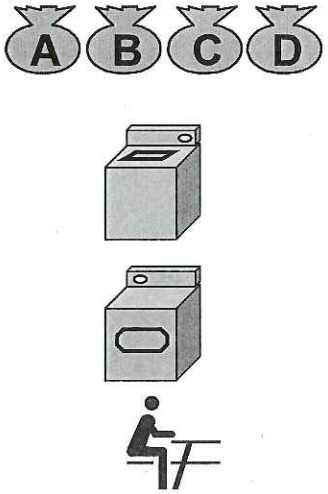
Pipelined Laundry: Start Load ASAP



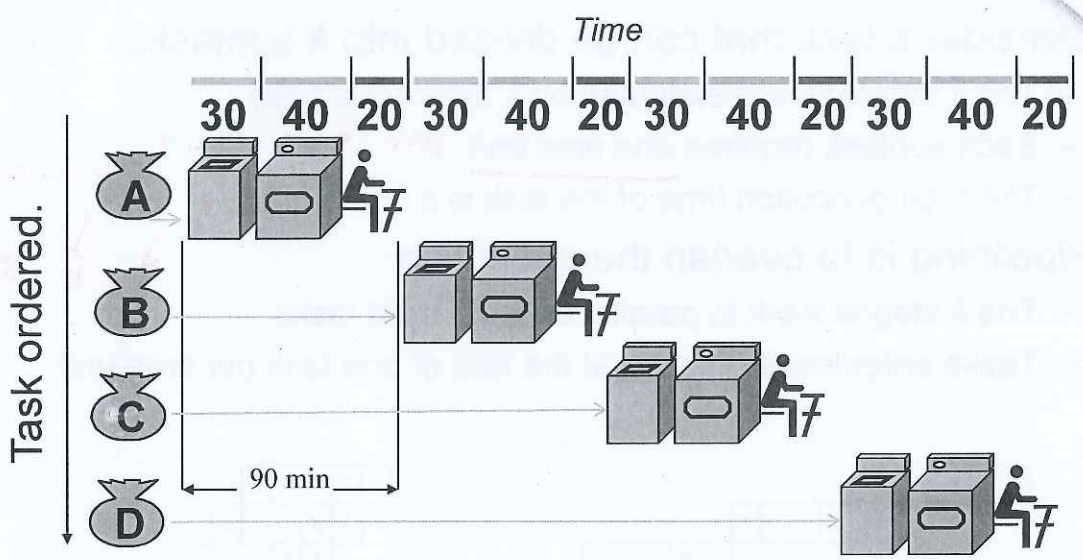
- Pipelined laundry takes 3 hours for 4 loads
- Speedup factor is 2 for 4 loads
- Time to wash, dry, and fold one load is still the same (90 minutes)

Pipelining: Laundry Example

- Small laundry has one washer, one dryer and one operator, it takes 90 minutes to finish one load:
  - Washer takes 30 minutes
  - Dryer takes 40 minutes
  - "operator folding" takes 20 minutes

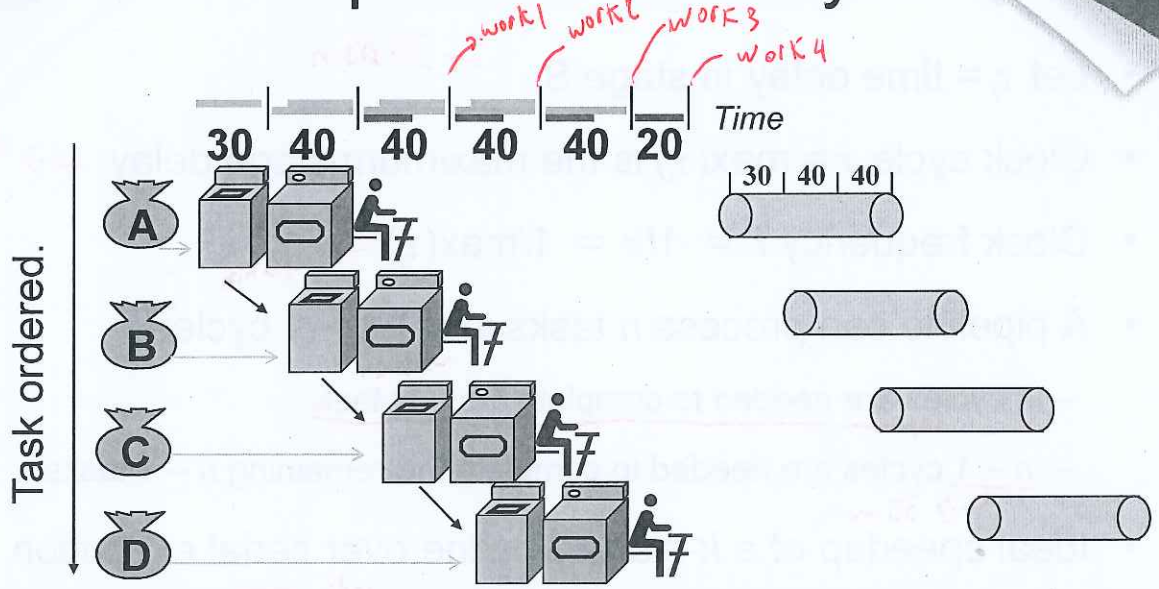


Sequential Laundry



Sequential laundry takes 6 hours for 4 loads

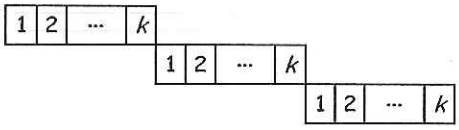
Pipelined Laundry



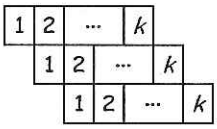
- Another operator asks for the delivery of loads to the laundry every 40 minutes!?.
- Pipelined laundry takes 3.5 hours for 4 loads
- Speedup factor is 6/3.5 = 1.7 for 4 loads
- Time to wash, dry, and fold one load is still the same (90 minutes)

Serial Execution versus Pipelining

- Consider a task that can be divided into k subtasks
  - The k subtasks are executed on k different stages
  - Each subtask requires one time unit
  - The total execution time of the task is k time units
- Pipelining is to overlap the execution
  - The k stages work in parallel on k different tasks
  - Tasks enter/leave pipeline at the rate of one task per time unit



Without Pipelining
One completion every k time units



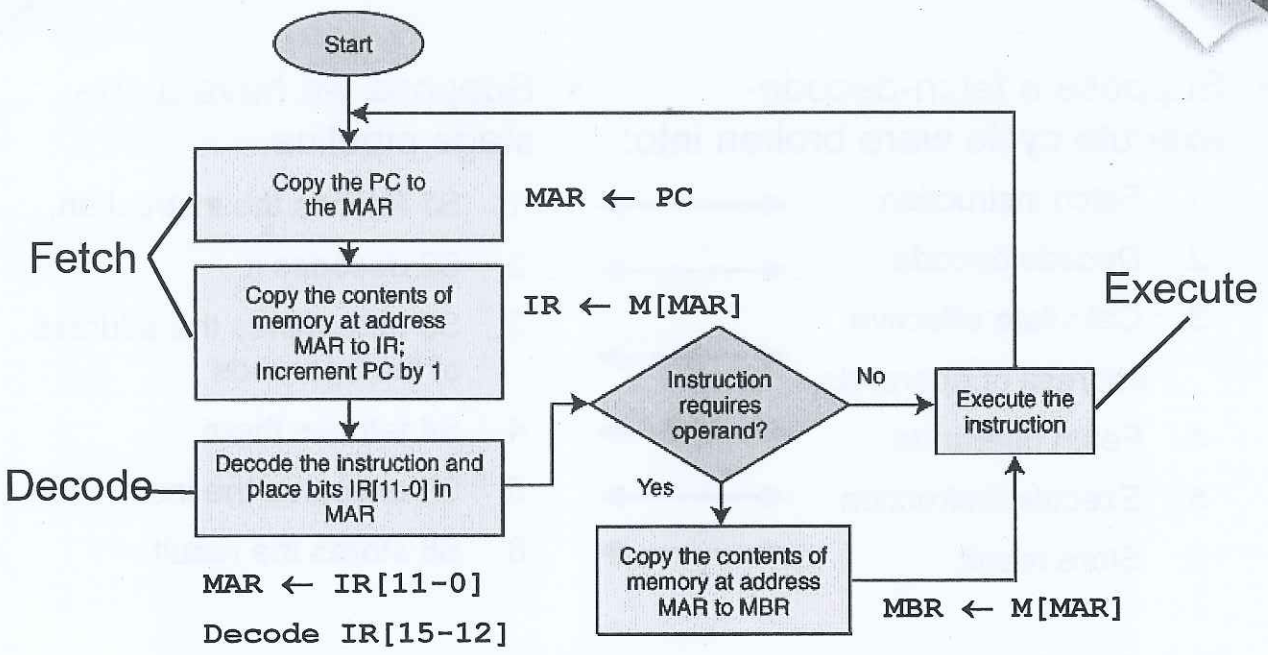
With Pipelining
One completion every 1 time unit

Pipeline Performance

- Let τ_i = time delay in stage S_i
- Clock cycle $\tau = \max(\tau_i)$ is the maximum stage delay
- Clock frequency $f = 1/\tau = 1/\max(\tau_i)$
- A pipeline can process n tasks in $k + n - 1$ cycles
  - k cycles are needed to complete the first task
  - $n - 1$ cycles are needed to complete the remaining $n - 1$ tasks
- Ideal speedup of a k-stage pipeline over serial execution

$$S_k = \frac{\text{Serial execution in cycles}}{\text{Pipelined execution in cycles}} = \frac{nk}{k + n - 1} \rightarrow k \text{ for large } n$$

The Fetch-Decode-Execute Cycle



Instruction Pipelining

- Some CPUs divide the fetch-decode-execute cycle into smaller steps.
- These smaller steps can often be executed in parallel to increase throughput.
- Such parallel execution is called instruction pipelining.
- Instruction pipelining provides for instruction level parallelism (ILP)

6-Stage Pipelining

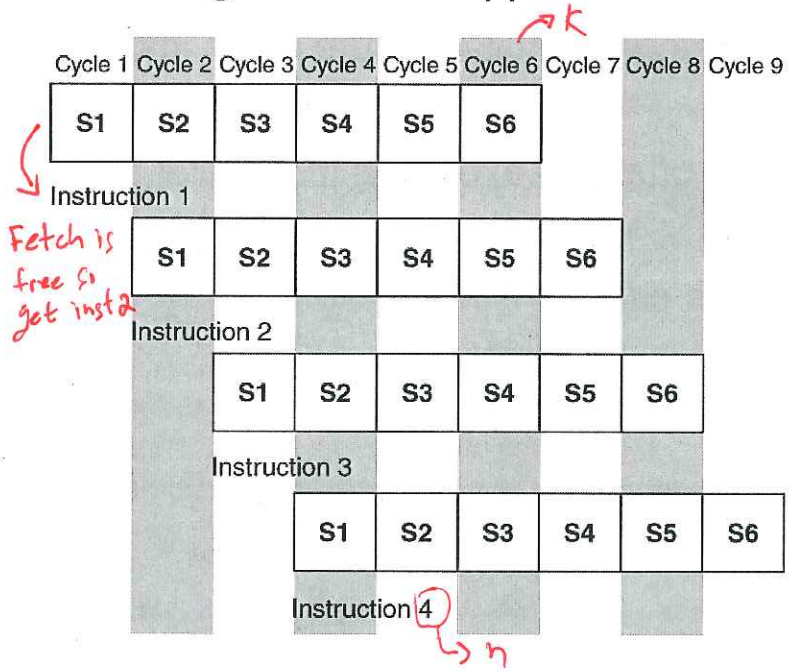
- Suppose a fetch-decode-execute cycle were broken into:
  1. Fetch instruction.
  2. Decode opcode.
  3. Calculate effective address of operands.
  4. Fetch operands
  5. Execute instruction
  6. Store result.

- Suppose we have a six-stage pipeline.
  1. S1 fetches the instruction,
  2. S2 decodes it
  3. S3 determines the address of the operands
  4. S4 fetches them
  5. S5 executes the instruction
  6. S6 stores the result.

Note: Not all instructions must go through each stage of the pipe.

6-Stage Pipelining

- For every clock cycle, one small step is carried out, and the stages are overlapped.



$\frac{k \times n}{6+(4-1)} = \frac{24}{9} = \frac{8}{3}$

- S1 fetches the instruction,
- S2 decodes it
- S3 determines the address of the operands
- S4 fetches them
- S5 executes the instruction
- S6 stores the result.

Serial Execution versus Pipelining

- Assumption:
  - A k-stage pipeline
  - the clock cycle time is t_p.
  - n instructions
- Each instruction represents a task, T, in the pipeline.
- Each task requires $k \times t_p$ time to complete. wash, dry, fold = 3h
  3×30min = 90min
- n tasks require $n\ t_p$ time to emerge to the pipeline.
- The last task requires $(k-1)t_p$ time to complete.
- So, the total time to complete n tasks using a *k*-stage pipeline requires: $(n+k-1)t_p$.

Pipeline Performance

- The speedup gain using a pipeline.
  - Without a pipeline, the total time to complete n tasks is $n \times k \times t_p$.
  - Using a k-stage pipeline, the total time to complete n tasks is $(n+k-1)t_p$.

4 6 2ns

$$\text{Speedup} = \frac{n \times k \times t_p}{(n+k-1)t_p}$$

- Take the limit as n approaches infinity, $(n+k-1)$ approaches n, resulting in a theoretical speedup of:

$$\text{Speedup} = \frac{k \times t_p}{t_p} = k$$

Pipeline Performance Summary

- Pipelining doesn't improve latency of a single instruction
- However, it improves throughput of entire workload
  - Instructions are initiated and completed at a higher rate
- In a *k*-stage pipeline, *k* instructions operate in parallel
  - Overlapped execution using multiple hardware resources
  - Potential speedup = number of pipeline stages *k*
  - Unbalanced lengths of pipeline stages reduces speedup
- Pipeline rate is limited by slowest pipeline stage
- Unbalanced lengths of pipeline stages reduces speedup
- Also, time to fill and drain pipeline reduces speedup

Pipeline Exercise

A nonpipelined system takes 200ns to process a task. The same task can be processed in a 5-segment pipeline with a clock cycle of 40ns.

1. Determine the speedup ratio of the pipeline for 200 tasks.

$$\text{Speedup} = \frac{n \times k \times t_p}{(n+k-1)t_p} = \frac{200 \times 5 \times 40}{(200+5-1) \times 40} = 4.9019$$

2. What is the maximum speedup that could be achieved with the pipeline unit over the nonpipelined unit?

$$\text{Speedup}_{max} = k = 5$$

Hazards

- Hazards: situations that causes incorrect execution
- A **bubble** or **pipeline stall** is a delay in execution of an instruction in an instruction pipeline in order to resolve a hazard.
  - Structure hazards
    - A required resource is busy
  - Data hazard
    - Need to wait for previous instruction to complete its data read/write
  - Control hazard
    - Deciding on control action depends on previous instruction

A 4-Stage Pipeline

- S1: fetch instruction
- S2: decode and calculate effective address
- S3: fetch operand
- S4: execute instruction and store results

Time Period	1	2	3	4	5	6	7	8	9	10	11	12	13
Instruction: 1	S1	S2	(S3)	S4									
2		S1	S2	(S3)	S4								
(branch) 3			(S1)	S2	(S3)	S4							
4				(S1)	S2	(S3)							
5					(S1)	S2							
6						(S1)							
8							S1	S2	S3	S4			
9								S1	S2	S3	S4		
10									S1	S2	S3	S4	

Structure Hazards

- Conflict for use of a resource
- In instruction pipeline with a single memory
  - Example: one instruction is storing a value to memory while another is being fetched from memory, both need access to memory.
- Pipelined datapaths require separate instruction/data memories, Or separate instruction/data caches

Data Hazards

- Dependency between instructions causes a data hazard
- The dependent instructions are close to each other
  - Pipelined execution might change the order of operand access
- Read After Write – RAW Hazard
  - Given two instructions *I* and *J*, where *I* comes before *J*
  - Instruction *J* should read an operand after it is written by *I*
  - Called a data dependence in compiler terminology

    `I: X = Y + 3` → once complete stores X

    `J: Z = 2 * X` → mult X is 3rd cycle. X not stored yet
  - Hazard occurs when *J* reads the operand before *I* writes it

Data Hazards

Time Period	1	2	3	4	5
X = Y + 3	fetch instruction	decode	fetch Y	execute & store X	
Z = 2 * X		fetch instruction	decode	fetch X	

- The problem arises at time period 4.
- The second instruction needs to fetch X, but the first instruction does not store the result until the execution is finished, so X is not available at the beginning of the time period.

- Solutions:
  - Stalling the pipeline
  - Data hazard detection: Detecting instructions whose source operands are destinations for instructions farther up the pipeline.
  - Forwarding: Routing data through special paths that exist between various stages of the pipeline.
  - Code scheduling

MIPS Processor Pipeline

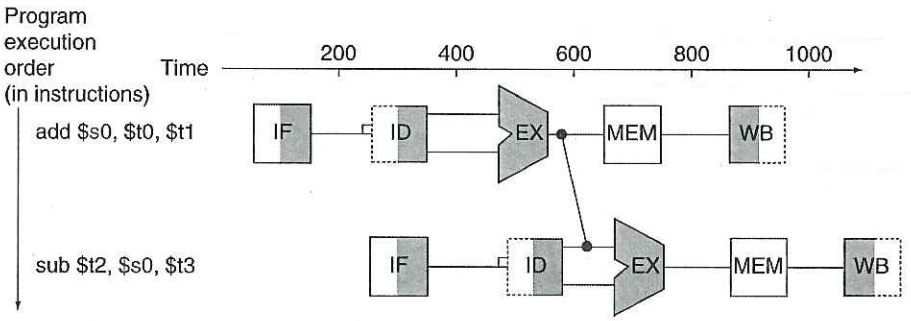
- Five stages, one cycle per stage

1. IF: Instruction Fetch from instruction memory
2. ID: Instruction Decode and register read
3. EX: Execute operation or calculate load/store address
4. MEM: Memory access for load and store
5. WB: Write Back result to register

Put/Store
load
8 + R2
→ LD R1 8(R2)
→ Actually stores on this step

Forwarding

- Use result when it is computed
  - Don't wait for it to be stored in a register
  - Requires extra connections in the datapath



wire that connects data output from ALU straight to the data needed by ALU in next instruction

Code Scheduling to Avoid Stalls

- Reorder code to avoid use of load result in the next instruction
- C code for `A = B + E; C = B + F;`

```
lw  $t1, 0($t0)
lw  $t2, 4($t0)
stall
add $t3, $t1, $t2
sw  $t3, 12($t0)
lw  $t4, 8($t0)
stall
add $t5, $t1, $t4
sw  $t5, 16($t0)
```

13 cycles

load first

```
lw  $t1, 0($t0)
lw  $t2, 4($t0)
lw  $t4, 8($t0)
add $t3, $t1, $t2
sw  $t3, 12($t0)
add $t5, $t1, $t4
sw  $t5, 16($t0)
```

11 cycles

Control Hazards

- Branch determines flow of control
  - Fetching next instruction depends on branch outcome
  - Pipeline can't always fetch correct instruction
    - Still working on ID stage of branch

Time Period	1	2	3	4	5	6	7	8	9	10	11	12	13
Instruction: 1	S1	S2	S3	S4									
2		S1	S2	S3	S4								
if → (branch) 3			S1	S2	S3	S4							
4				S1	S2	S3	Stop						
5					S1	S2	Stop						
6						S1	Stop						
else → 8							S1	S2	S3	S4			
9								S1	S2	S3	S4		
10									S1	S2	S3	S4	

Branch Prediction

- Predict outcome of branch
  - Only stall if prediction is wrong
- Static branch prediction
  - Based on typical branch behavior
  - Example: loop and if-statement branches
- Dynamic branch prediction
  - Hardware measures actual branch behavior
    - e.g., record recent history of each branch
  - Assume future behavior will continue the trend
    - When wrong, stall while re-fetching, and update history

90% it was true so just go ahead & fetch true block

Lecture 19

- Today's topics
  - Types of memory
  - Memory hierarchy

6.1 Introduction

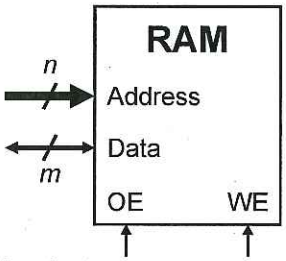
- Memory organization
- A clear understanding of these ideas is essential for the analysis of system performance.

Types of Memory

- There are two kinds of main memory:
  - Random access memory (RAM)
    - Dynamic RAM (DRAM)
    - Static RAM (SRAM)
  - Read-only-memory (ROM) → no capacitors used
- ROM does not need to be refreshed.
- ROM is used to store permanent, or semi-permanent data that persists even while the system is turned off.

Random Access Memory

- Large arrays of storage cells
- Volatile memory
  - Hold the stored data as long as it is powered on
- Random Access
  - Access time is practically the same to any data on a RAM chip
- Output Enable (OE) control signal
  - Specifies read operation
- Write Enable (WE) control signal
  - Specifies write operation
- $2^n \times m$ RAM chip: n-bit address and m-bit data



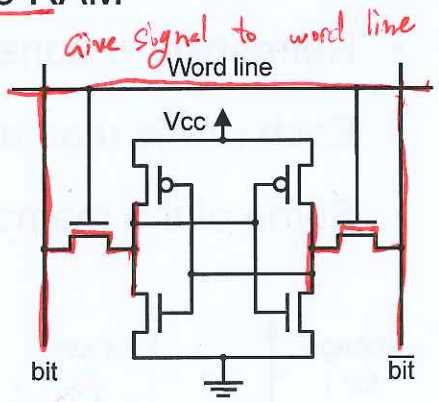
SRAM vs. DRAM

- Static RAM (SRAM) for Cache
  - Requires 6-8 transistors per bit
  - Requires low power to retain bit
- Dynamic RAM (DRAM) for Main Memory
  - One transistor + capacitor per bit
  - Must be re-written after being read
  - Must also be periodically refreshed
    - Each row can be refreshed simultaneously
  - Address lines are multiplexed
    - Upper half of address: Row Access Strobe (RAS)
    - Lower half of address: Column Access Strobe (CAS)

Recall: P-type transistor creates channel when path is 0
N-type transistor creates channel when path gets 1

Static RAM Storage Cell

- Static RAM (SRAM): fast but expensive RAM
- 6-Transistor cell with no static current
- Typically used for caches
- Provides fast access time
- Cell Implementation:
  - Cross-coupled inverters store bit
  - Two pass transistors enable the cell to be read and written

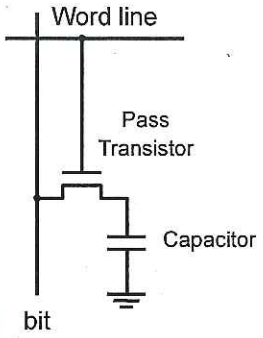


Typical SRAM cell

give signal to word line
2 gain access to data

Dynamic RAM Storage Cell

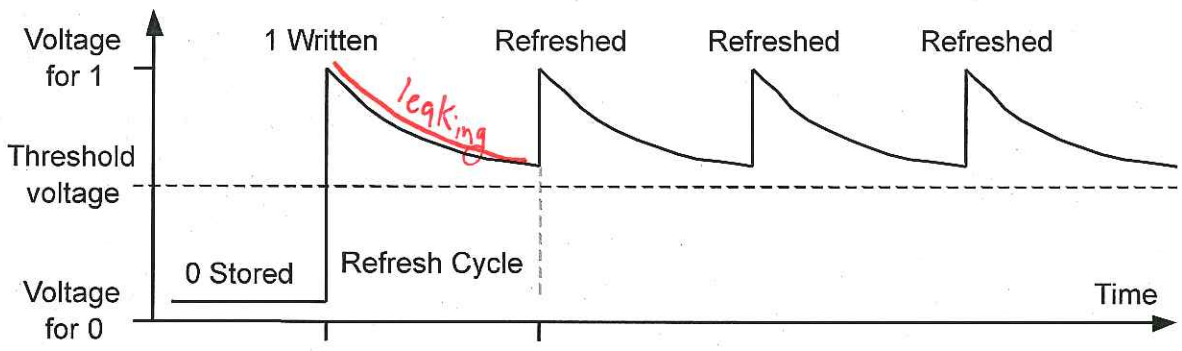
- Dynamic RAM (DRAM): slow, cheap, and dense memory
- Typical choice for main memory
- Cell Implementation:
  - 1-Transistor cell (pass transistor)
  - Trench capacitor (stores bit)
- Bit is stored as a charge on capacitor
- Must be refreshed periodically (it leaks over time)
  - Because of leakage of charge from tiny capacitor
- Refreshing for all memory rows
  - Reading each row and writing it back to restore the charge
    (every time you read you have to write back again)



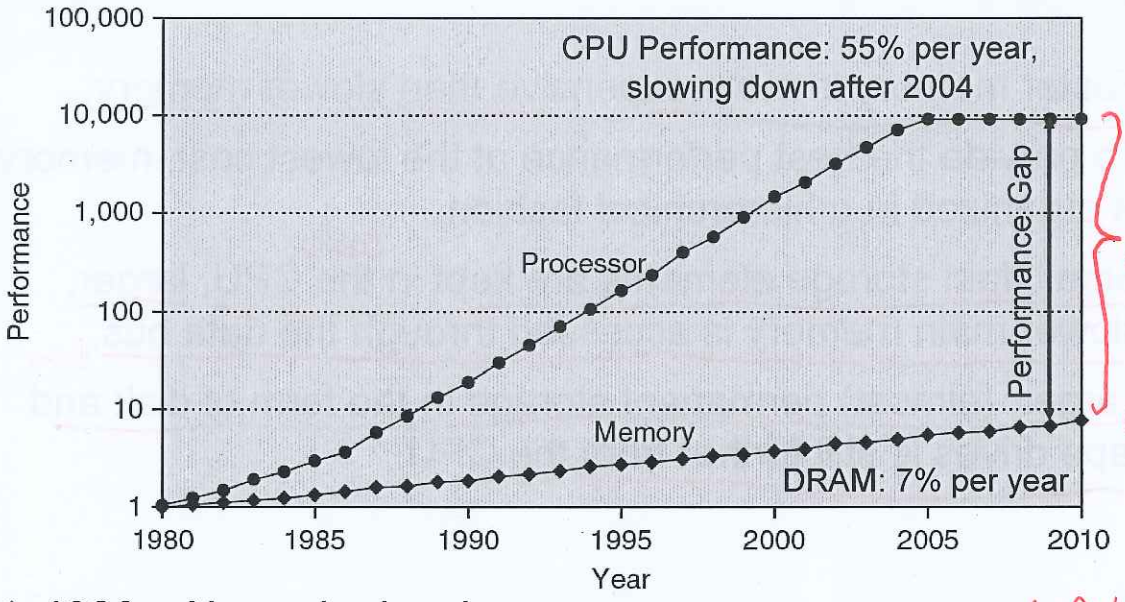
Typical DRAM cell

DRAM Refresh Cycles

- Refresh cycle is about tens of milliseconds
- Refreshing is done for the entire memory
- Each row is read and written back to restore the charge
- Some of the memory bandwidth is lost to refresh cycles



Processor-Memory Performance Gap



Gap b/w memory & processor

- 1980 – No cache in microprocessor
- 1995 – Two-level cache on microprocessor

need faster memory

The Need for Cache Memory

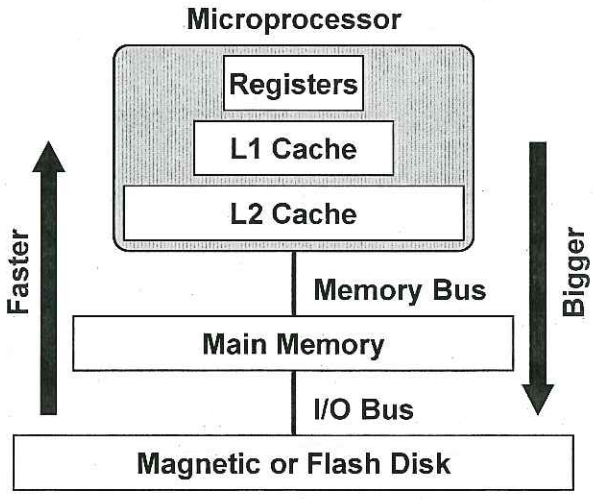
- Widening speed gap between CPU and main memory
  - Processor operation takes less than 1 ns
  - Main memory requires more than 50 ns to access
- Each instruction involves at least one memory access
  - One memory access to fetch the instruction
  - A second memory access for load and store instructions
- Memory bandwidth limits the instruction execution rate
- Cache memory can help bridge the CPU-memory gap
- Cache memory is small in size but fast

The Memory Hierarchy

- Faster memory is more expensive than slower memory.
- To provide the best performance at the lowest cost, memory is organized in a hierarchical fashion.
- Small, fast storage elements are kept in the CPU, larger, slower main memory is accessed through the data bus.
- Larger, (almost) permanent storage in the form of disk and tape drives is still further from the CPU.

Typical Memory Hierarchy

- Registers are at the top of the hierarchy
  - Typical size < 1 KB
  - Access time < 0.5 ns
- Level 1 Cache (8 – 64 KB)
  - Access time: 1 ns
- L2 Cache (512KB – 8MB)
  - Access time: 3 – 10 ns
- Main Memory (4 – 16 GB)
  - Access time: 50 – 100 ns
- Disk Storage (> 200 GB)
  - Access time: 5 – 10 ms



6.3 The Memory Hierarchy

- Data are processed at CPU.
- To access a particular piece of data, the CPU first sends a request to its nearest memory, usually cache.
- If the data is not in cache, then main memory is queried. If the data is not in main memory, then the request goes to disk.
- Once the data is located, then the data, and a number of its nearby data elements are fetched into cache memory.

Principle of Locality of Reference

- Programs access small portion of their address space
  - At any time, only a small set of instructions & data is needed
- Temporal Locality (in time) — if data is used its very likely it will be used again so keep it in cache
  - If an item is accessed, probably it will be accessed again soon
  - Same loop instructions are fetched each iteration
  - Same procedure may be called and executed many times
- Spatial Locality (in space) — using from address 101 means its very likely ill be using address 102 as well, bring it to cache
  - Tendency to access contiguous instructions/data in memory
  - Sequential execution of Instructions
  - Traversing arrays element by element

loops

Arrays

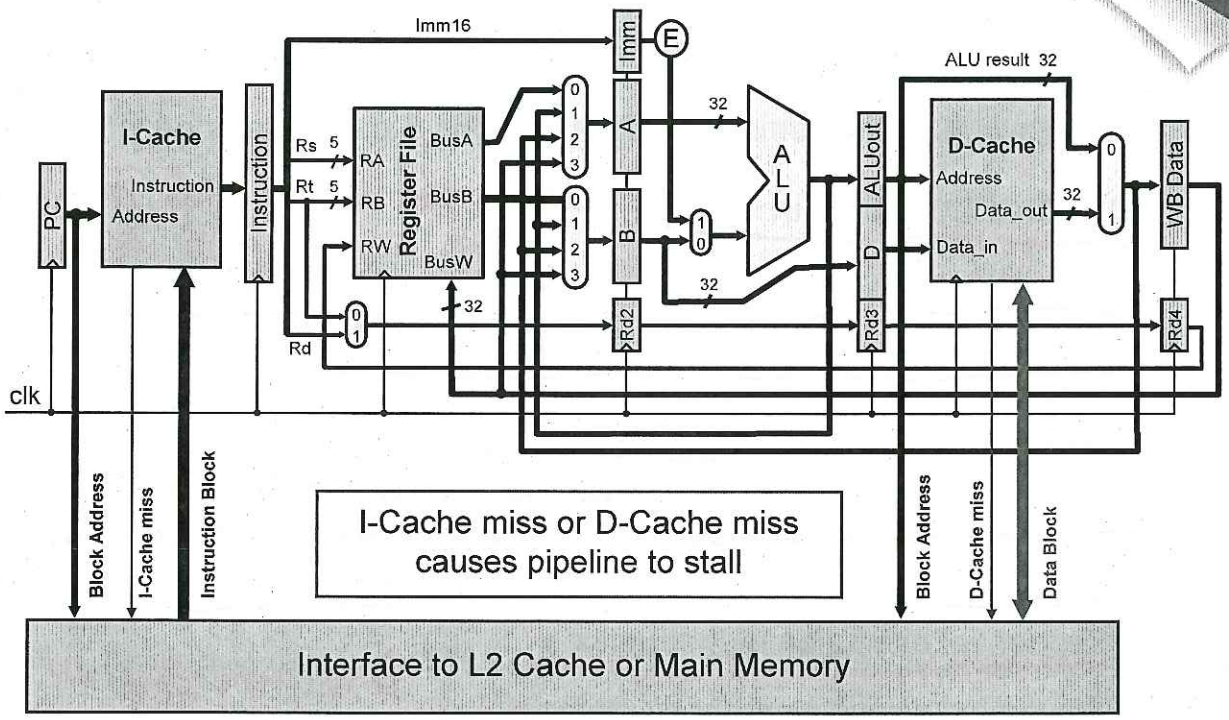
fetch more than we need just in case

What is a Cache Memory ?

- Small and fast (SRAM) memory technology
  - Stores the subset of instructions & data currently being accessed
- Used to reduce average access time to memory
- Caches exploit temporal locality by …
  - Keeping recently accessed data closer to the processor
- Caches exploit spatial locality by …
  - Moving blocks consisting of multiple contiguous words
- Goal is to achieve
  - Fast speed of cache memory access
  - Balance the cost of the memory system

Cache is like the items kept at the checkout line at supermarket registers. Small amount of items but fast access to them. Everything else is in the store.

Cache Memories in the Datapath



Almost Everything is a Cache !

- In computer architecture, almost everything is a cache!
- Registers: a cache on variables – software managed
- First-level cache: a cache on second-level cache
- Second-level cache: a cache on memory
- Memory: a cache on hard disk
  - Stores recent programs and their data
  - Hard disk can be viewed as an extension to main memory
- Branch target and prediction buffer
  - Cache on branch target and prediction information

Definitions

- This leads us to some definitions.
  - A *hit* is when data is found at a given memory level.
  - A *miss* is when it is not found.
  - The *hit rate* is the percentage of time data is found at a given memory level. 90% hit rate = 10% miss rate
  - The *miss rate* is the percentage of time it is not.
  - Miss rate = 1 - hit rate.
  - The *hit time* is the time required to access data at a given memory level.
  - The *miss penalty* is the time required to process a miss, including the time that it takes to replace a block of memory plus the time it takes to deliver the data to the processor. time taken to fetch data from somewhere else

Four Basic Questions on Caches

- Q1: Where can a block be placed in a cache?
  - Block placement
  - Direct Mapped, Set Associative, Fully Associative
- Q2: How is a block found in a cache?
  - Block identification
  - Block address, tag, index
- Q3: Which block should be replaced on a miss?
  - Block replacement
  - FIFO, Random, LRU
- Q4: What happens on a write?
  - Write strategy
  - Write Back or Write Through (with Write Buffer)

Lecture 20

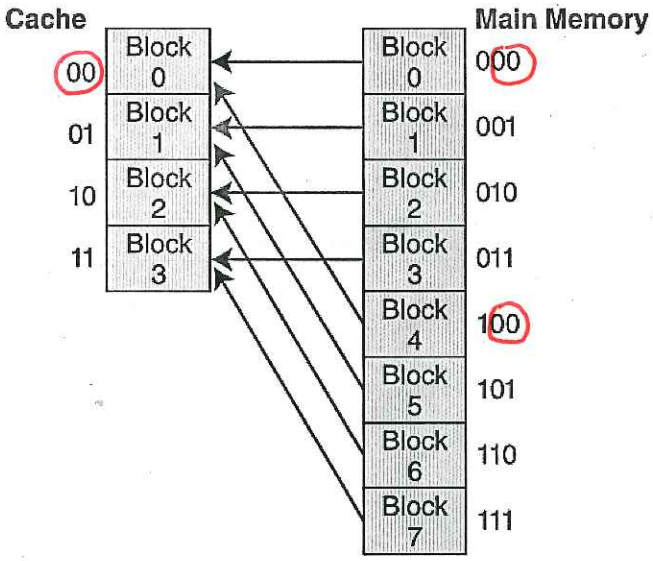
- Last lecture:
  - Types of memory
  - Memory hierarchy
- Today's lecture:
  - Cache mapping schemes
  - Block replacement
  - Block replacement
  - Write strategy

Four Basic Questions on Caches

- Q1: Where can a block be placed in a cache?
  - Block placement
  - Direct Mapped, Set Associative, Fully Associative
- Q2: How is a block found in a cache?
  - Block identification
  - Block address, tag, offset
- Q3: Which block should be replaced on a miss?
  - Block replacement
  - FIFO, Random, LRU
- Q4: What happens on a write?
  - Write strategy
  - Write Back or Write Through (with Write Buffer)

Block Placement: Direct Mapped

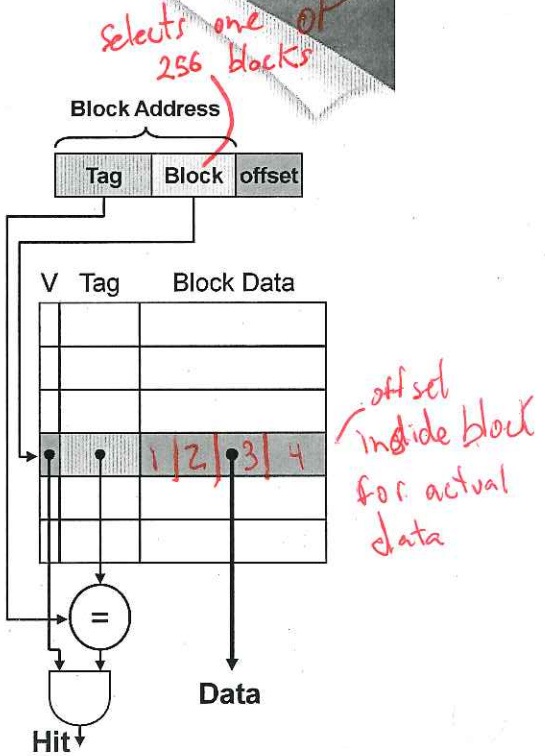
- Block: unit of data transfer between cache and memory
- Direct Mapped Cache:
  - A block can be placed in exactly one location in the cache
  - (Block address) modulo (#Blocks in cache)



* Direct-Mapped Cache

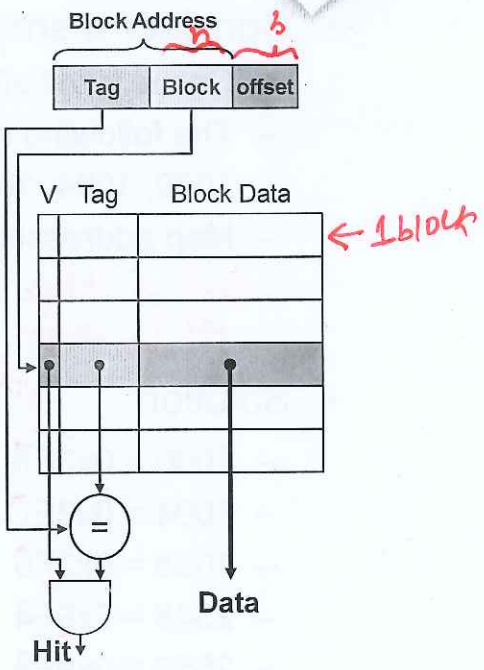
- A memory address is divided into
  - Block address: identifies block in memory
  - Block offset: to access bytes within a block
- A block address is further divided into
  - Block: used for direct cache access
  - Tag: most-significant bits of block address

  Index = Block Address **mod** *Cache Blocks*
- Tag must be stored also inside cache
  - For block identification
- A valid bit is also required to indicate
  - Whether a cache block is valid or not



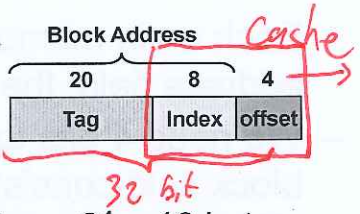
Direct Mapped Cache – cont'd

- Cache hit: block is stored inside cache
  - Index is used to access cache block
  - Address tag is compared against stored tag
  - If equal and cache block is valid then **hit**
  - Otherwise: **cache miss**
- If number of cache blocks is 2^n
  - n bits are used for the cache index
- If number of bytes in a block is 2^b
  - b bits are used for the block offset
  - byte-addressable
- If 32 bits are used for an address
  - $32 - n - b$ bits are used for the tag
- Cache data size = 2^{n+b} bytes



Mapping an Address to a Cache Block

- Example
  - Consider a direct-mapped cache with 256 blocks
  - Block size = 16 bytes (byte-addressable)
  - Compute tag, index, and byte offset of address: 0x01FFF8AC
  - Physical address: 32 bits

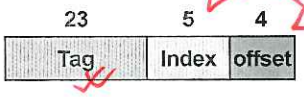


- Solution
  - 32-bit address is divided into:
    - 4-bit byte offset field, because block size = 2^4 = 16 bytes
    - 8-bit cache index, because there are 2^8 = 256 blocks in cache
    - 20-bit tag field
  - Byte offset = 0xC = 12 (least significant 4 bits of address)
  - Cache index = 0x8A = 138 (next lower 8 bits of address)
  - Tag = 0x01FFF (upper 20 bits of address)

Example on Cache Placement & Misses

- Consider a small direct-mapped cache with 32 blocks
  - Cache is initially empty, Block size = 16 bytes
  - The following memory addresses (in decimal) are referenced: 1000, 1004, 1008, 2548, 2552, 2556.
  - Map addresses to cache blocks and indicate whether hit or miss

block of 16 byte each will be brought

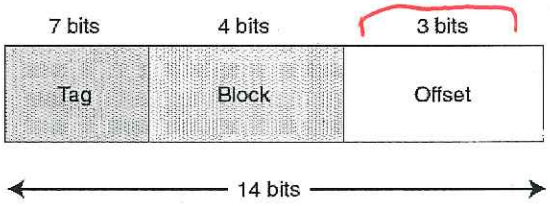


- Solution:

– 1000 = 0x3E8	cache index = 0x1E	Miss (first access)
– 1004 = 0x3EC	cache index = 0x1E	Hit
– 1008 = 0x3F0	cache index = 0x1F	Miss (first access) change in block, so miss
– 2548 = 0x9F4	cache index = 0x1F	Miss (different tag) change in tag, so miss
– 2552 = 0x9F8	cache index = 0x1F	Hit
– 2556 = 0x9FC	cache index = 0x1F	Hit

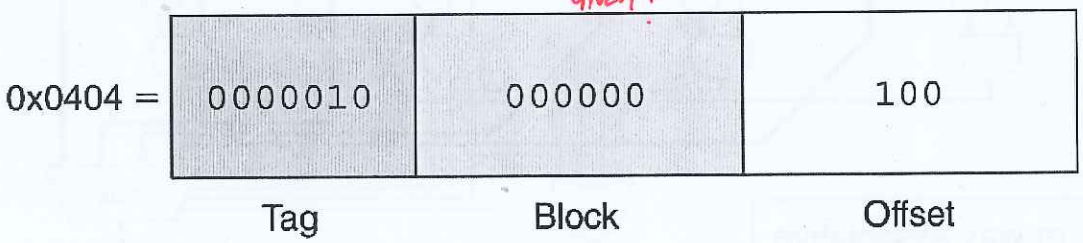
Exercise

- EXAMPLE 6.2 Assume a byte-addressable memory consists of 2^{14} bytes, cache has 16 blocks, and each block has 8 bytes. $2^3 = 8$ offset
  - The number of memory blocks are: $\frac{2^{14}}{2^3} = 2^{11}$
  - Each main memory address requires14 bits. Of this 14-bit address field, the rightmost 3 bits reflect the offset field
  - We need 4 bits to select a specific block in cache, so the block field consists of the middle 4 bits.
  - The remaining 7 bits make up the tag field.



Exercise

- EXAMPLE 6.4 Consider 16-bit memory addresses and 64 blocks of cache where each block contains 8 bytes. We have:
  - 3 bits for the offset
  - 6 bits for the block
  - 7 bits for the tag.
- A memory reference for 0x0404 maps as follows:



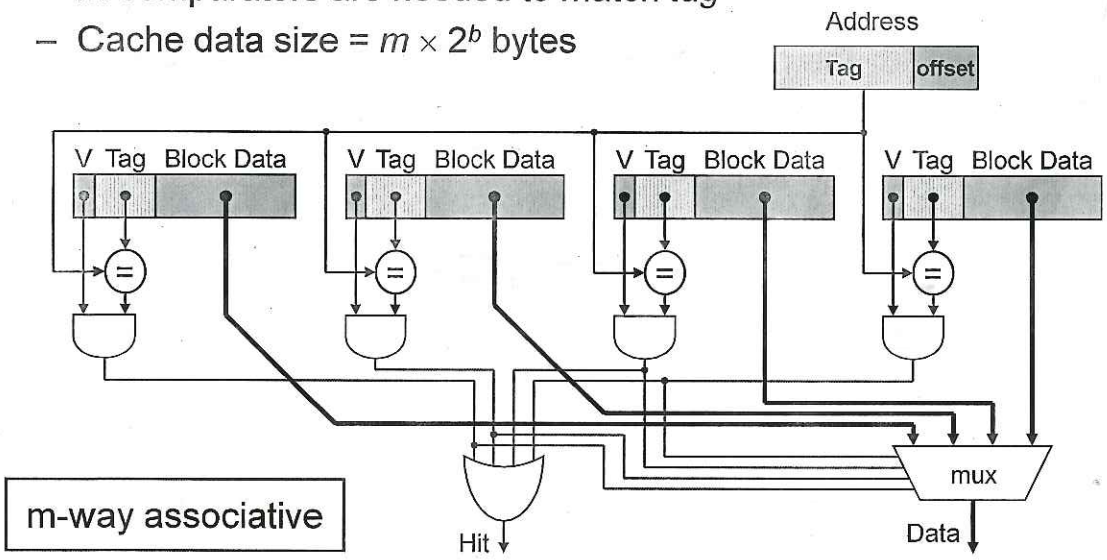
Direct Mapped Cache Summary

- In summary, direct mapped cache maps main memory blocks in a modular fashion to cache blocks.
- The mapping depends on:
  - The number of bits in the main memory address (how many addresses exist in main memory)
  - The number of blocks are in cache (which determines the size of the block field)
  - How many addresses (either bytes or words) are in a block (which determines the size of the offset field)

Fully Associative Cache

- A block can be placed anywhere in cache ⇒ no indexing
- If m blocks exist then
  - m comparators are needed to match *tag*
  - Cache data size = $m \times 2^b$ bytes

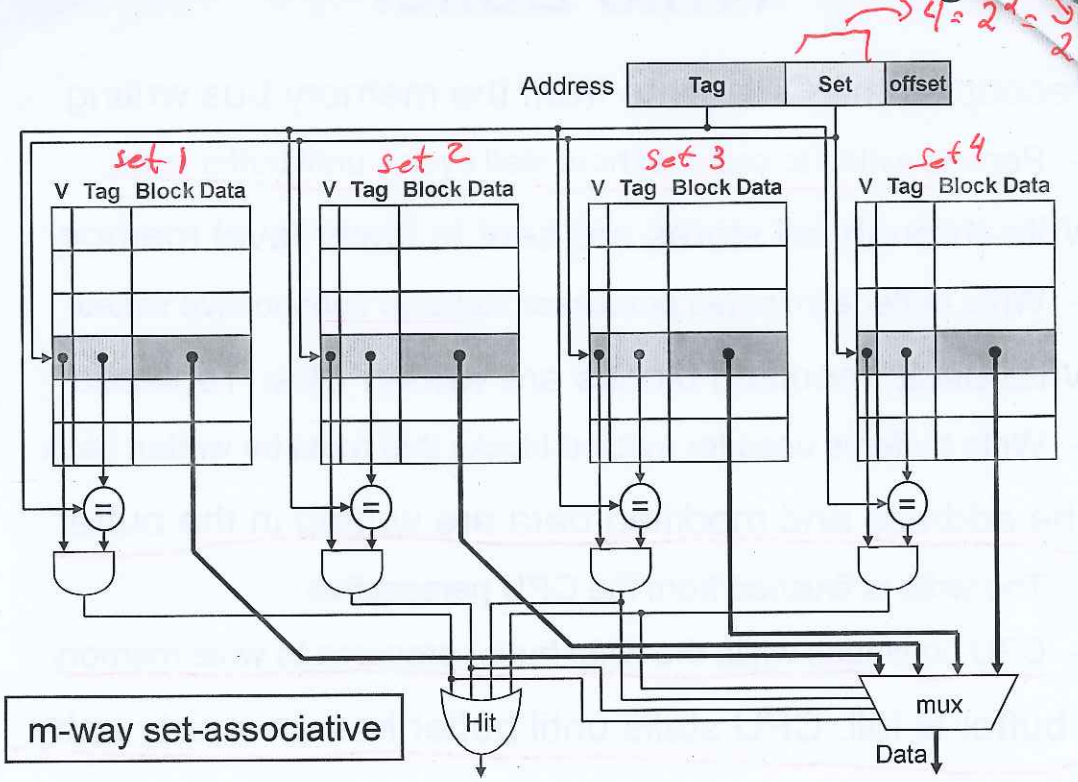
Longer tag = more flexible



Set-Associative Cache

- A set is a group of blocks that can be indexed
- A block is first mapped onto a set
  - Set index = Block address **mod** Number of sets in cache
- If there are m blocks in a set (m-way set associative) then
  - m tags are checked in parallel using m comparators
- If 2^n sets exist then set index consists of n bits
- Cache data size = $m \times 2^{n+b}$ bytes (with 2^b bytes per block)
  - Without counting tags and valid bits
- A direct-mapped cache has one block per set ($m = 1$)
- A fully-associative cache has one set ($2^n = 1$ or $n = 0$)

Set-Associative Cache Diagram



Handling Write - Write Policy

- Write Through: (instant)
  - Writes update cache and lower-level memory
  - Cache control bit: only a Valid bit is needed
  - Memory always has latest data, which simplifies data coherency
  - Can always discard cached data when a block is replaced

change in cache needs to also write to RAM
↳ change in cache & update lower level memory at same time

- Write Back: (delayed)
  - Writes update cache only
  - Cache control bits: Valid and Modified bits are required
  - Modified cached data is written back to memory when replaced
  - Multiple writes to a cache block require only one write to memory
  - Uses less memory bandwidth than write-through and less power
  - However, more complex to implement than write through

wait until a block in cache needs to be replaced. Then write to memory

modified bit changes from 0 → 1

Write Buffer

- Decouples the CPU write from the memory bus writing
  - Permits writes to occur without stall cycles until buffer is full
- Write-through: all stores are sent to lower level memory
  - Write buffer eliminates processor stalls on consecutive writes
- Write-back: modified blocks are written when replaced
  - Write buffer is used for evicted blocks that must be written back
- The address and modified data are written in the buffer
  - The write is finished from the CPU perspective
  - CPU continues while the write buffer prepares to write memory
- If buffer is full, CPU stalls until buffer has an empty entry

Replacement Policy

for Full & Set mapping only

- Which block to be replaced on a cache miss?
- No selection alternatives for direct-mapped caches
- *m* blocks per set to choose from for associative caches
- Random replacement
  - Candidate blocks are randomly selected
  - One counter for all sets (0 to *m* – 1): incremented on every cycle
  - On a cache miss replace block specified by counter
- First In First Out (FIFO) replacement
  - Replace oldest block in set
  - One counter per set (0 to *m* – 1): specifies oldest block to replace
  - Counter is incremented on a cache miss

Replacement Policy – cont'd

- Least Recently Used (LRU)
  - Replace block that has been unused for the longest time
  - Order blocks within a set from least to most recently used
  - Update ordering of blocks on each cache hit
  - With m blocks per set, there are $m!$ possible permutations
- Pure LRU is too costly to implement when $m > 2$
  - $m = 2$, there are 2 permutations only (a single bit is needed)
  - m = 4, there are 4! = 24 possible permutations
  - LRU approximation is used in practice
- For large $m > 4$,

  Random replacement can be as effective as LRU