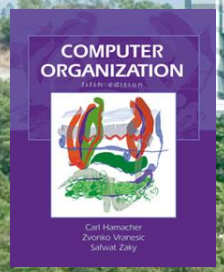
香港中文大學
The Chinese University of Hong Kong

CSCI2510 Computer Organization

Lecture 10: Control Unit and Instruction Encoding

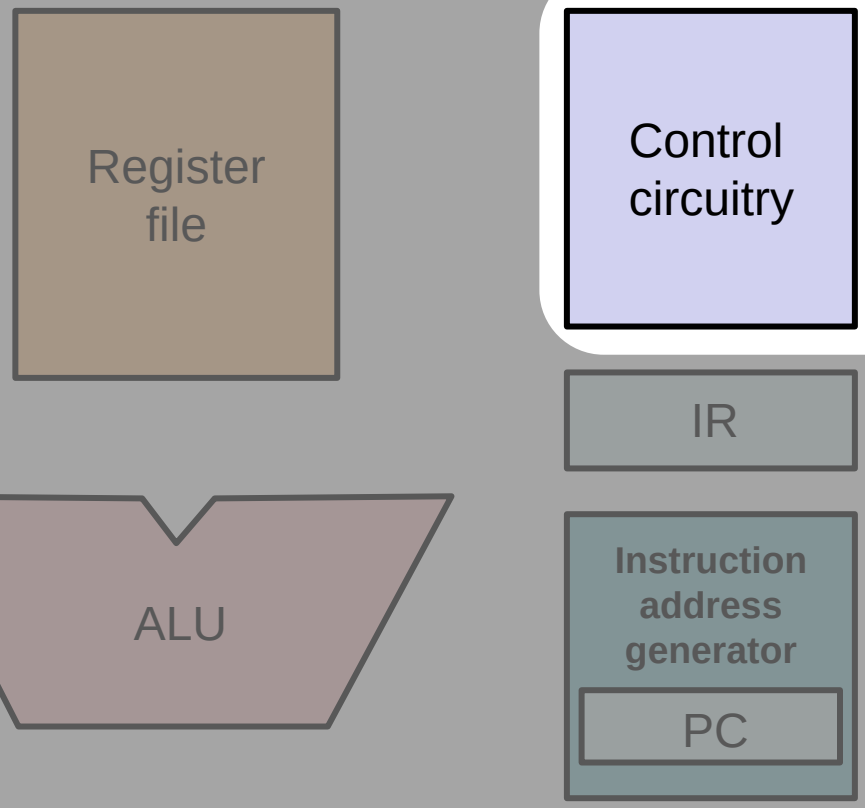
Ming-Chang YANG

mcyang@cse.cuhk.edu.hk



Reading: Chap. 7.4~7.5 (5th Ed.)

Recall: Components of a Processor



Register file: a memory unit for the processor's **general-purpose registers (GPRs)**

Control circuitry: Interpret or decode the fetched instruction

IR: Hold the instruction until its execution is completed (*special purpose register*)

Arithmetic and Logic Unit (ALU): Perform an arithmetic or logic operation

PC: Keep track of the address of the next instruction to be fetched and executed (*special purpose register*)

Processor-memory interface: Allow the communication between processor and memory

Outline

- Control Signal Generation
  1) Hard-wired Control
  2) Micro-programmed Control

- Machine Instruction Encoding

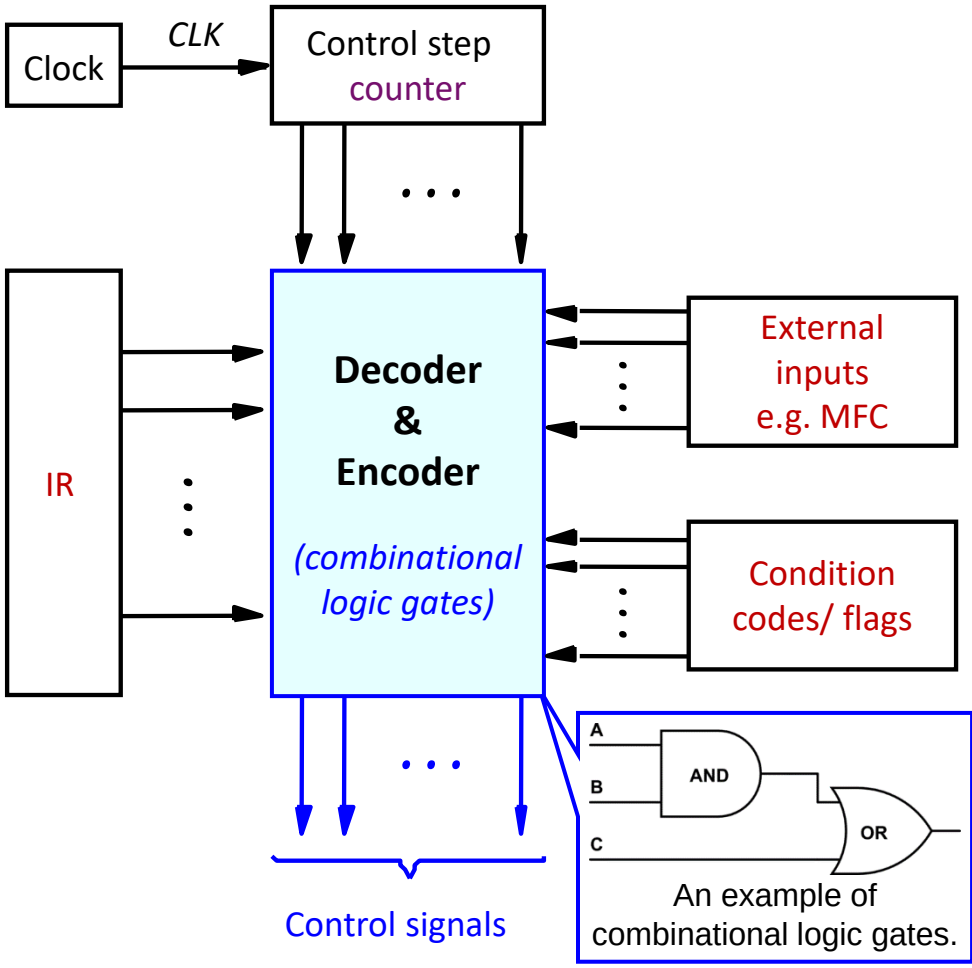
Control Signal Generation

- The processor must have some means to generate the control signals for instruction execution:
  1) Hard-wired control
  2) Micro-programmed control

- Every control signals (e.g., PC-out, MDR-in, ADD, SUB, …) are switched **on** (active) and **off** (inactive) at suitable time.
  - The time duration is determined by the clock.

1) Hard-wired Control

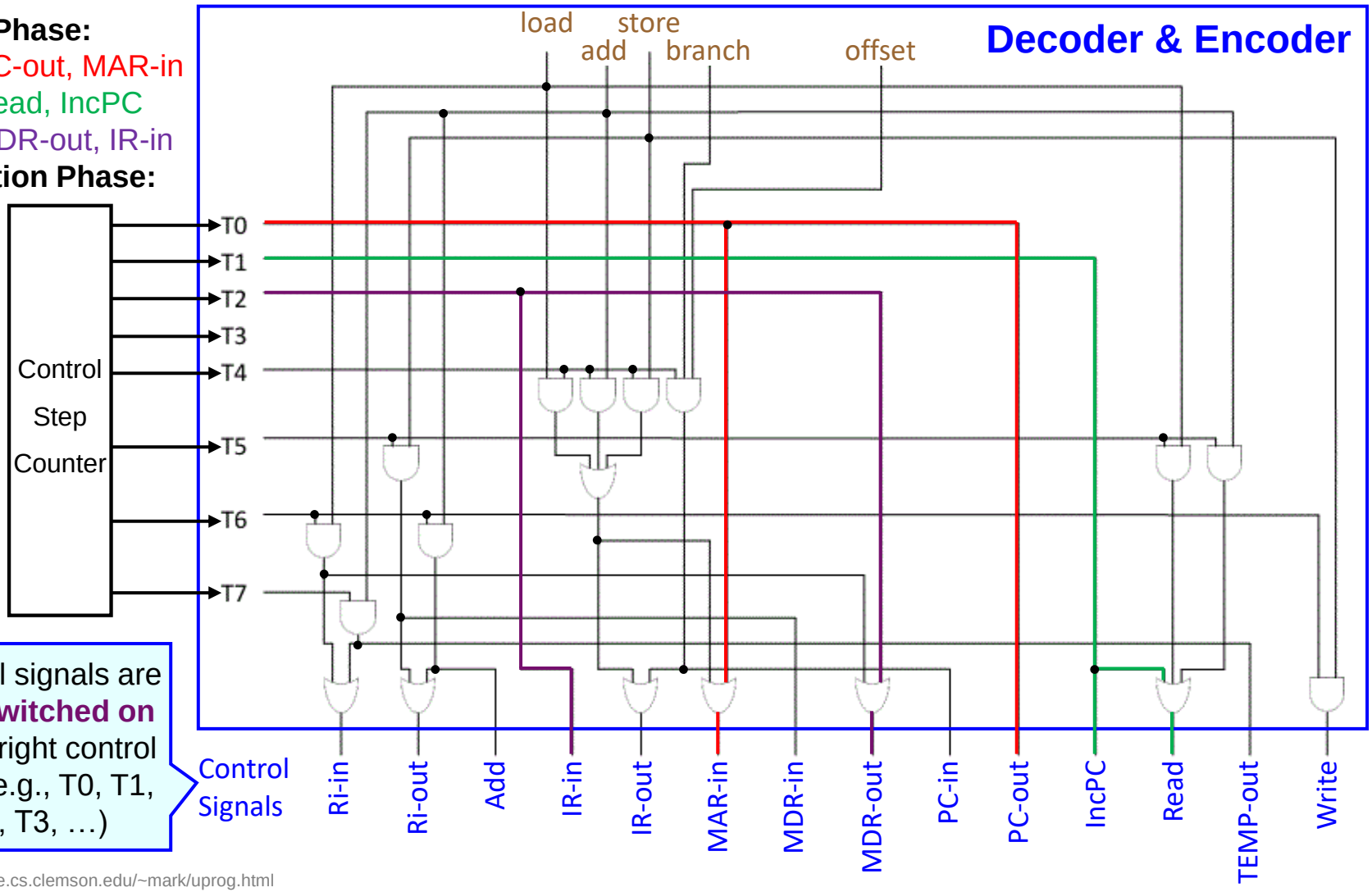
- **Hard-wired Control**: The combinational logic gates are used to determine the sequence of control signals:
  - A counter is used keep track of the control steps.
  - Control signals are functions of the IR, external inputs and condition codes
  - The control signals are generated at the right time (i.e., control step).



1) Hard-wired Control (Cont'd)

- A simplified example:

Fetch Phase:
T0: PC-out, MAR-in
T1: Read, IncPC
T2: MDR-out, IR-in
Execution Phase:
…

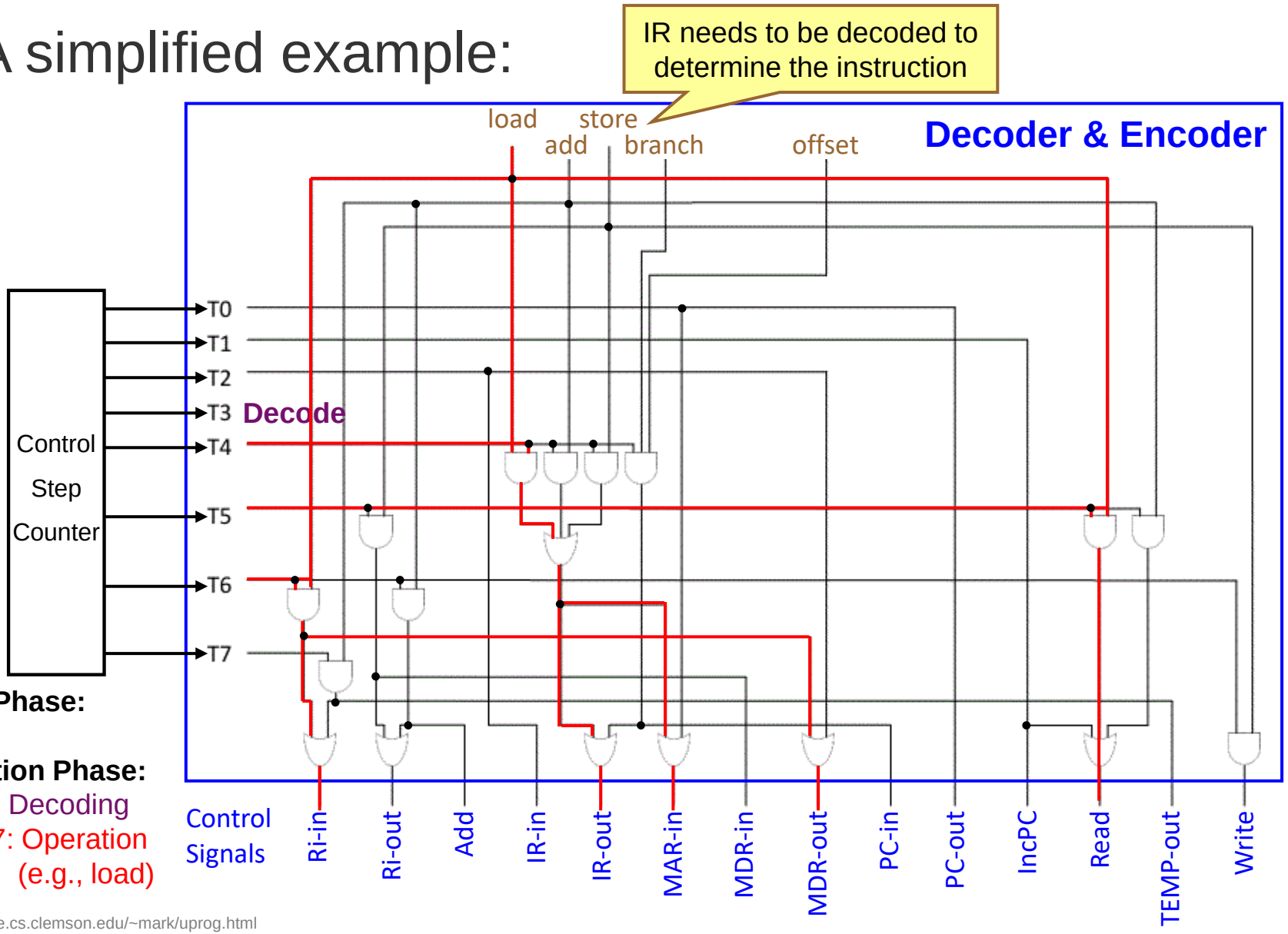


Control signals are only **switched on** at the right control step (e.g., T0, T1, T2, T3, …)

https://people.cs.clemson.edu/~mark/uprog.html

1) Hard-wired Control (Cont'd)

- A simplified example:



Fetch Phase:

…

Execution Phase:

T3: IR Decoding

T4~T7: Operation (e.g., load)

https://people.cs.clemson.edu/~mark/uprog.html

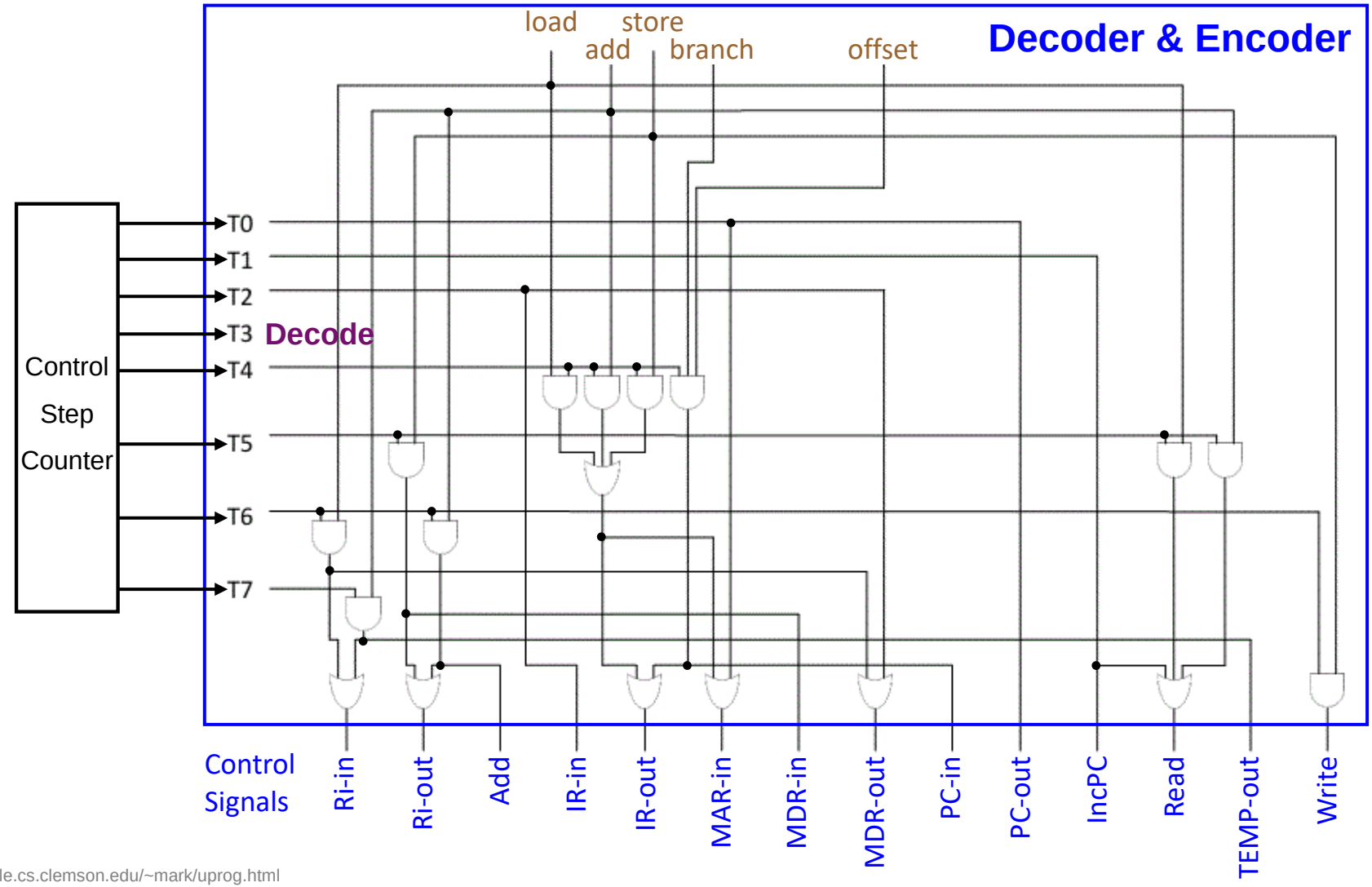
Class Exercise 10.1

Student ID: _____ Date:
Name: _____ _____

- The control sequences of different instructions may consist of a different number of steps.
  - For example, the load instruction is composed of 6 steps (3 for the fetch, 1 for the decode, and 3 for the execution).
- Can you tell how many control steps are required for the other three instructions (i.e., add, store, and branch) in the given simplified hard-wired control?

Class Exercise 10.1

- A simplified example:



https://people.cs.clemson.edu/~mark/uprog.html

1) Hard-wired Control (Cont'd)

- The wiring of the logic gates for control signal generation is fixed.
  - Simple signal:
    - PC-out = T0
  - Complicated signal :
    - MDR-inE = ((IR == ADD) and ((T2) or (T5))) or
      ((IR == SUB) and ((T2) or (T5))) or
      … CarryFlag or … and … or … and … and …
- The hard-wired control can operate at high speed.
- However, the hard-wired control has little flexibility.
  - It can only implement instruction set of limited complexity.

2) Micro-programmed Control

- The control signals are generated by a micro-program.
- Every line is a control word.
- Micro-programs are stored in a special memory (control store).

Step	Action — Ex: ADD R1, (R3)
1	PC_{out} , MAR_{in} , Read, Select4, Add, Z_{in}, B_{in}
2	Z_{out} , PC_{in} , Y_{in} , WMFC, MDR_{inE}
3	MDR_{out} , IR_{in}
4	$R3_{out}$, MAR_{in} , Read
5	$R1_{out}$, Y_{in} , WMFC, MDR_{inE}
6	MDR_{out} , SelectY, Add, Z_{in} , B_{in}
7	Z_{out} , $R1_{in}$, End

Micro-Program

Micro-instruction	..	PC*in*	PC*out*	MAR *in*	Read	MDR*out*	IR*in*	Y*in*	Select	Add	Z*in*	Z*out*	R1*out*	R1*in*	R3*out*	WMFC	End	B_{in}	MDR_{inE}
1		0	1	1	1	0	0	0	1	1	1	0	0	0	0	0	0	1	0
2	Control word	1	0	0	0	0	0	1	0	0	0	1	0	0	0	1	0	0	1
3		0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0
4		0	0	1	1	0	0	0	0	0	0	0	0	0	1	0	0	0	0
5																			
6		0	0	0	0	1	0	0	0	1	1	0	0	0	0	0	0	1	0
7		0	0	0	0	0	0	0	0	0	0	1	0	1	0	0	1	0	0

Class Exercise 10.2

- Please fill in the missing control word in the below micro-program for the instruction ADD R1, (R3):

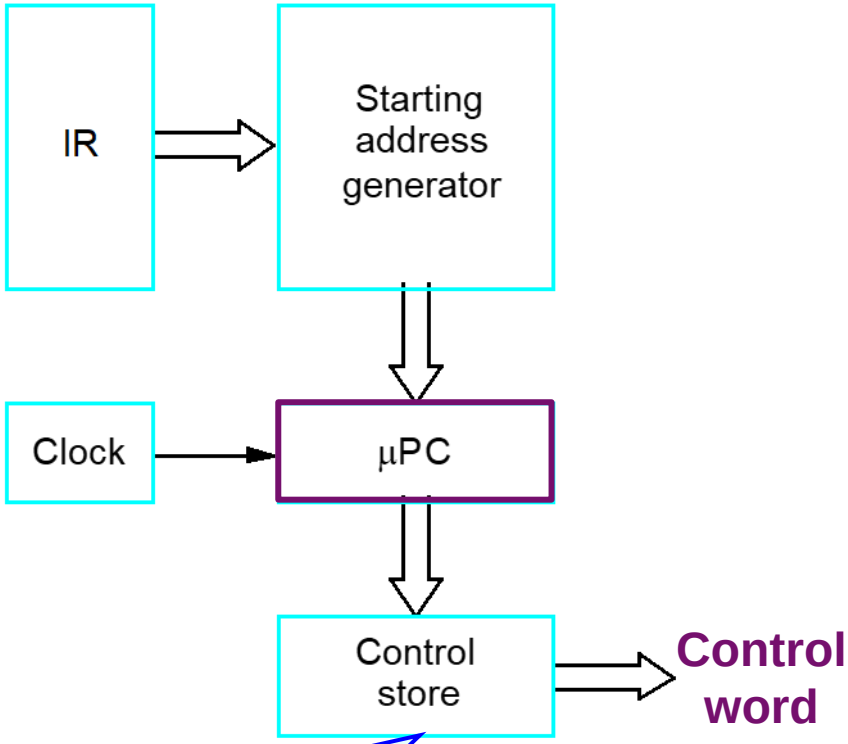
Ex: ADD R1, (R3)

Step	Action
1	PC_{out} , MAR_{in} , Read, Select4, Add, Z_{in}, B_{in}
2	Z_{out} , PC_{in} , Y_{in} , WMF C, MDR_{inE}
3	MDR_{out} , IR_{in}
4	$R3_{out}$, MAR_{in} , Read
5	$R1_{out}$, Y_{in} , WMF C, MDR_{inE}
6	MDR_{out} , SelectY, Add, Z_{in} , B_{in}
7	Z_{out} , $R1_{in}$, End

Micro-instruction	..	PC*in*	PC*out*	MAR *in*	Read	MDR *out*	IR *in*	Y *in*	Select	Add	Z*in*	Z *out*	R1*out*	R1 *in*	R3*out*	WMFC	End	B_{in}	MDR_{inE}
1		0	1	1	1	0	0	0	1	1	1	0	0	0	0	0	0	1	0
2		1	0	0	0	0	0	1	0	0	0	1	0	0	0	1	0	0	1
3		0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0
4		0	0	1	1	0	0	0	0	0	0	0	0	0	1	0	0	0	0
Control word 5																			
6		0	0	0	0	1	0	0	0	1	1	0	0	0	0	0	0	1	0
7		0	0	0	0	0	0	0	0	0	0	1	0	1	0	0	1	0	0

2) Micro-programmed Control (Cont'd)

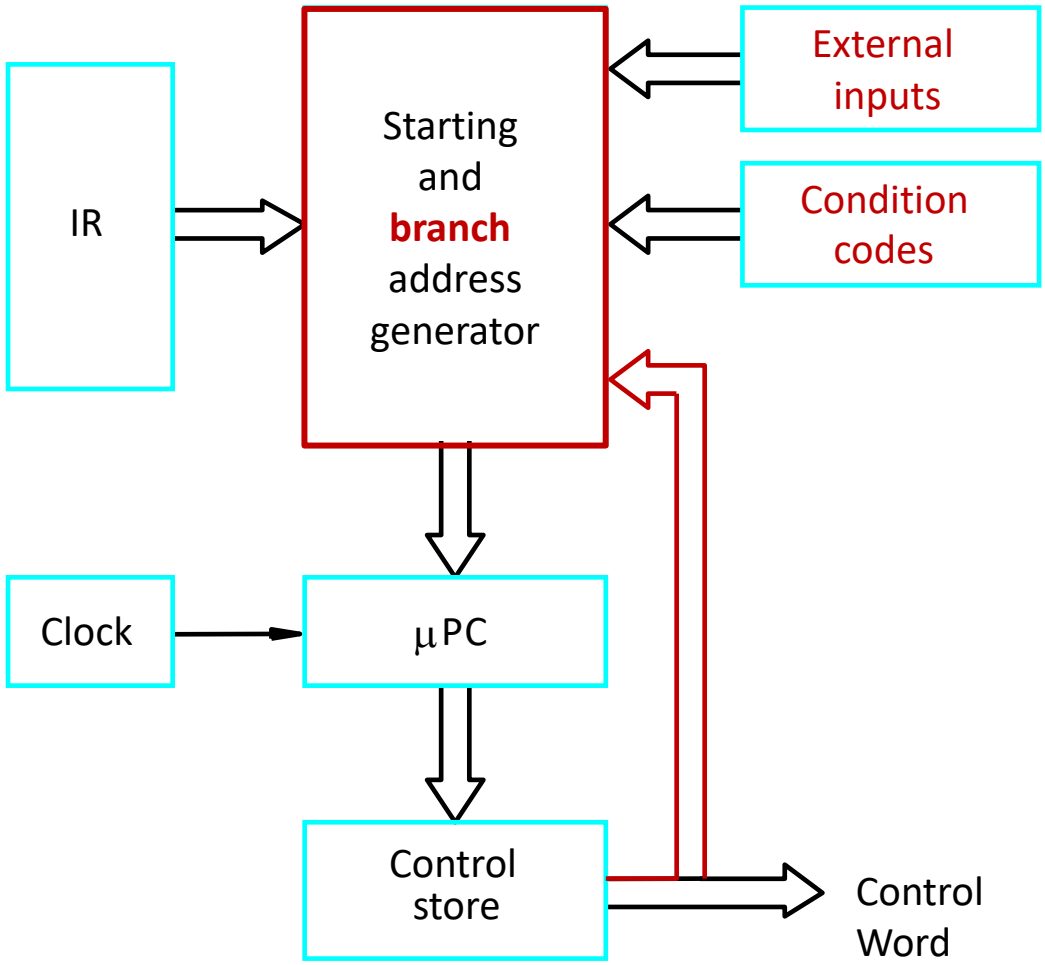
- A micro-program counter (uPC) is used to read control words sequentially from control store.
  - ① Whenever a new instruction is loaded into IR, "Starting Address Generator" loads the starting address into uPC.
  - ② uPC increments by clock, causing successive micro-instructions to be read out from the control store.
  - ③ Control signals are generated in the correct sequence defined by a micro-program.



Micro - instruction	..	PC_{in}	PC_{out}	MAR_{in}	Read	MDR_{out}	IR_{in}	Y_{in}	Select	Add	Z_{in}	Z_{out}	$R1_{out}$	$R1_{in}$	$R3_{out}$	WMFC	End	B_{in}	MDR_{inE}
1		0	1	1	1	0	0	0	1	1	1	0	0	0	0	0	0	1	0
2		1	0	0	0	0	0	1	0	0	0	1	0	0	0	1	0	0	1
3		0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0
4		0	0	1	1	0	0	0	0	0	0	0	0	0	1	0	0	0	0
5		0	0	0	0	0	0	1	0	0	0	0	1	0	0	1	0	0	1
6		0	0	0	0	1	0	0	0	1	1	0	0	0	0	0	0	1	0
7		0	0	0	0	0	0	0	0	0	0	1	0	1	0	0	1	0	0

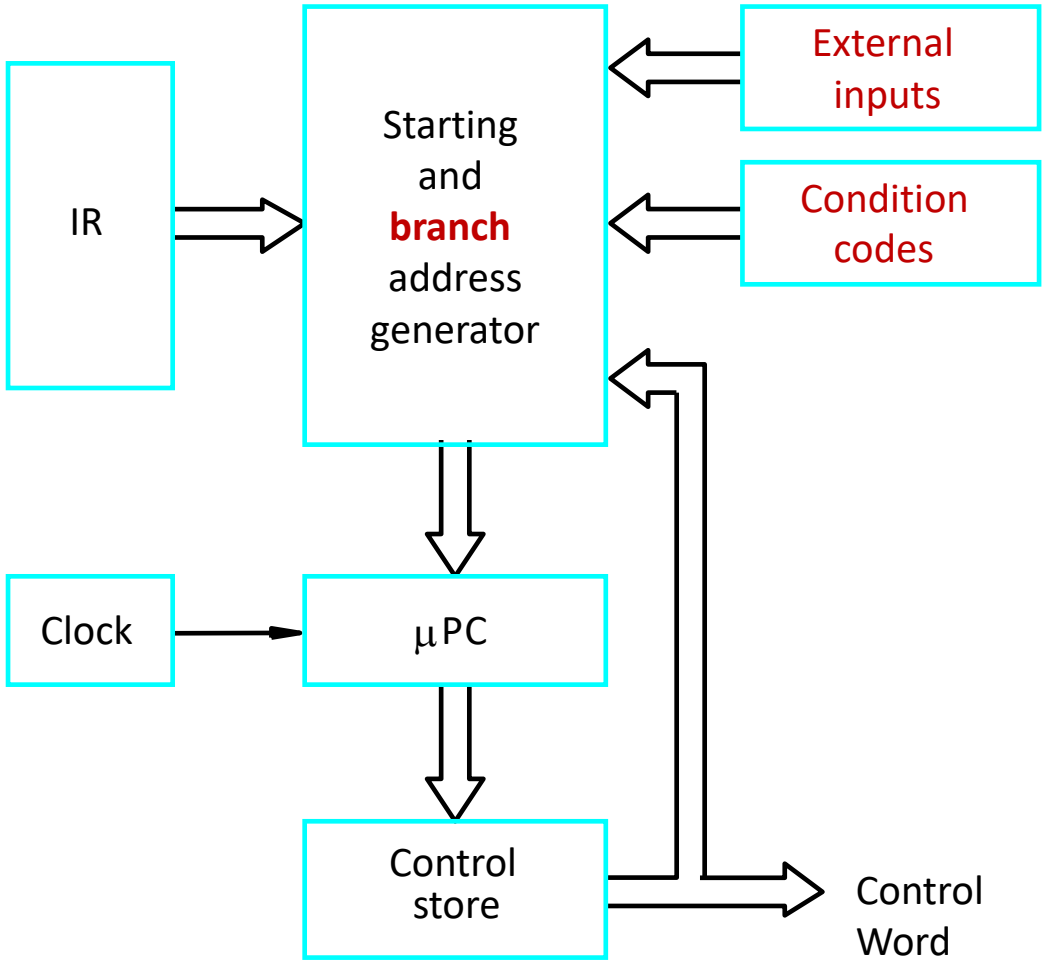
2) Micro-programmed Control (Cont'd)

- The previous scheme is not able to change the control sequence by other inputs.
  - It cannot support branch on condition code (e.g. Jump if < 0)
- Starting and **branch** address generator:
  - Load new address into uPC when needed.
    - By checking condition codes or external inputs that can affect uPC.



2) Micro-programmed Control (Cont'd)

- uPC is incremented every cycle except:
  - ① When a new instruction loaded into IR, uPC is loaded with starting address of the micro-program.
  - ② When taken branches, uPC is updated with the branch address.
  - ③ When taken END micro-instruction, uPC is reset.



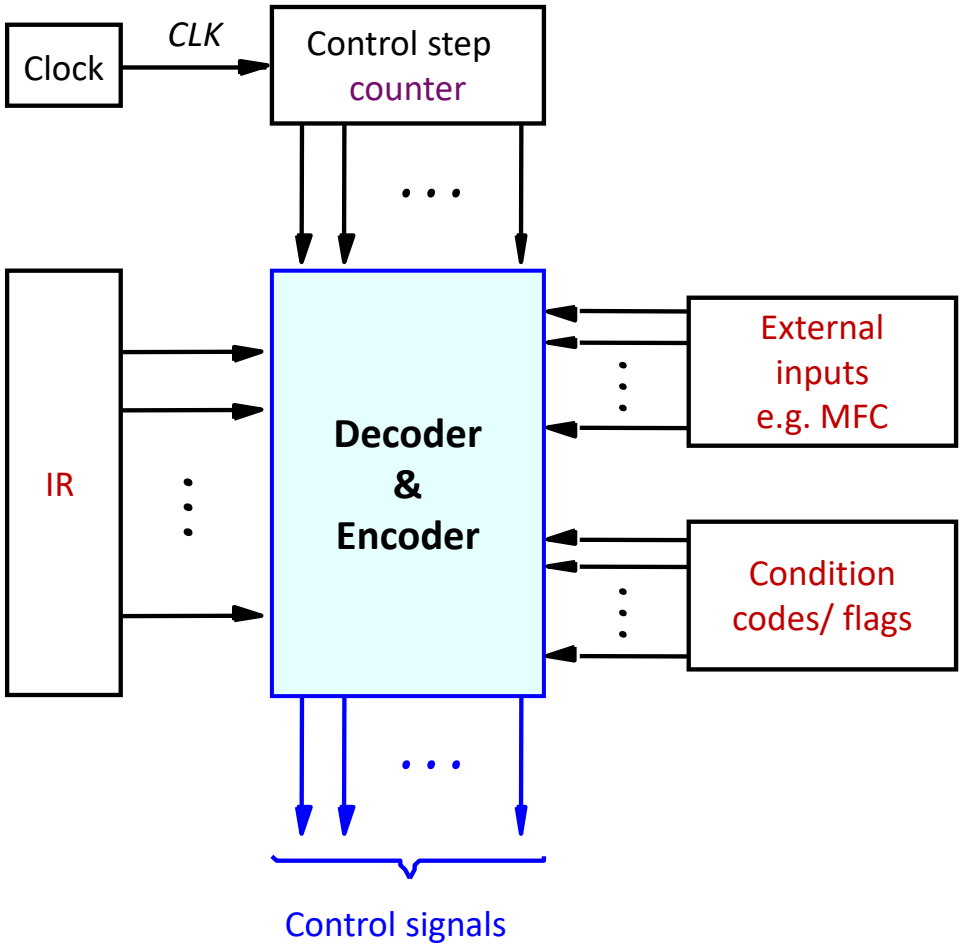
Outline

- Control Signal Generation
  1) Hard-wired Control
  2) Micro-programmed Control

- Machine Instruction Encoding

Machine Instruction Encoding

- An instruction must be encoded in a compact binary pattern.
- The decoder must interpret (or decode) the instruction, and generate the control signals correctly.



Why Machine Instruction Encoding?

- We have a bunch of instructions:
  - Such as add, subtract, move, shift, rotate, branch, etc.
- Instructions may use operands of different sizes.
  - Such as 32-bit and 8-bit number, or 8-bit ASCII characters.
- Both the type of operation and the type of operands need to be specified in encoded binary patterns.
  - **Type of Operation**: Often referred to as the OP code.
    - E.g., 8 bits can represent 256 different OP codes.
  - **Type of Operands**: Addressing modes.
    - An operand is the part of an instruction that specifies data to be operating on or manipulated.

Example: 8051/8052 OP Code Map

OpCode (Upper Nibble \ Lower Nibble)	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	NOP	AJMP page0	LJMP addr16	RR A	INC A	INC iram	INC @R0	INC @R1	INC R0	INC R1	INC R2	INC R3	INC R4	INC R5	INC R6	INC R7
1	JBC bit, reladdr	ACALL page0	LCALL addr16	RRC A	DEC A	DEC iram	DEC @R0	DEC @R1	DEC R0	DEC R1	DEC R2	DEC R3	DEC R4	DEC R5	DEC R6	DEC R7
2	JB bit, reladdr	AJMP page1	RET	RL A	ADD A, #data	ADD A, iram	ADD A, @R0	ADD A, @R1	ADD A, R0	ADD A, R1	ADD A, R2	ADD A, R3	ADD A, R4	ADD A, R5	ADD A, R6	ADD A, R7
3	JNB bit, reladdr	ACALL page1	RETI	RLC A	ADDC A, #data	ADDC A, iram	ADDC A, @R0	ADDC A, @R1	ADDC A, R0	ADDC A, R1	ADDC A, R2	ADDC A, R3	ADDC A, R4	ADDC A, R5	ADDC A, R6	ADDC A, R7
4	JC reladdr	AJMP page2	ORL iram, A	ORL iram, #data	ORL A, #data	ORL A, iram	ORL A, @R0	ORL A, @R1	ORL A, R0	ORL A, R1	ORL A, R2	ORL A, R3	ORL A, R4	ORL A, R5	ORL A, R6	ORL A, R7
5	JNC reladdr	ACALL page2	ANL iram, A	ANL iram, #data	ANL A, #data	ANL A, iram	ANL A, @R0	ANL A, @R1	ANL A, R0	ANL A, R1	ANL A, R2	ANL A, R3	ANL A, R4	ANL A, R5	ANL A, R6	ANL A, R7
6	JZ reladdr	AJMP page3	XRL iram, A	XRL iram, #data	XRL A, #data	XRL A, iram	XRL A, @R0	XRL A, @R1	XRL A, R0	XRL A, R1	XRL A, R2	XRL A, R3	XRL A, R4	XRL A, R5	XRL A, R6	XRL A, R7
7	JNZ reladdr	ACALL page3	ORL C, bit	JMP @A+DPTR	MOV A, #data	MOV iram, #data	MOV @R0, #data	MOV @R1, #data	MOV R0, #data	MOV R1, #data	MOV R2, #data	MOV R3, #data	MOV R4, #data	MOV R5, #data	MOV R6, #data	MOV R7, #data
8	SJMP reladdr	AJMP page4	ANL C, bit	MOVC A, @A+PC	DIV AB	MOV iram, iram	MOV iram, @R0	MOV iram, @R1	MOV iram, R0	MOV iram, R1	MOV iram, R2	MOV iram, R3	MOV iram, R4	MOV iram, R5	MOV iram, R6	MOV iram, R7
9	MOV DPTR, #data16	ACALL page4	MOV bit, C	MOVC A, @A+DPTR	SUBB A, #data	SUBB A, iram	SUBB A, @R0	SUBB A, @R1	SUBB A, R0	SUBB A, R1	SUBB A, R2	SUBB A, R3	SUBB A, R4	SUBB A, R5	SUBB A, R6	SUBB A, R7
A	ORL C, /bit	AJMP page5	MOV C, bit	INC DPTR	MUL AB	???	MOV @R0, iram	MOV @R1, iram	MOV R0, iram	MOV R1, iram	MOV R2, iram	MOV R3, iram	MOV R4, iram	MOV R5, iram	MOV R6, iram	MOV R7, iram
B	ANL C, /bit	ACALL page5	CPL bit	CPL C	CJNE A, #data,	CJNE A, iram,	CJNE @R0, #data,	CJNE @R1 #data,	CJNE R0, #data,	CJNE R1, #data,	CJNE R2, #data,	CJNE R3, #data,	CJNE R4, #data,	CJNE R5, #data,	CJNE R6, #data,	CJNE R7, #data,
C	PUSH iram	AJMP page6	CLR bit	CLR C	SWAP A	XCH A, iram	XCH A, @R0	XCH A, @R1	XCH A, R0	XCH A, R1	XCH A, R2	XCH A, R3	XCH A, R4	XCH A, R5	XCH A, R6	XCH A, R7
D	POP iram	ACALL page6	SETB bit	SETB C	DA A	DJNZ iram, reladdr	XCHD A, @R0	XCHD A, @R1	DJNZ R0, reladdr	DJNZ R1, reladdr	DJNZ R2, reladdr	DJNZ R3, reladdr	DJNZ R4, reladdr	DJNZ R5, reladdr	DJNZ R6, reladdr	DJNZ R7, reladdr
E	MOVX A, @DPTR	AJMP page7	MOVX A, @R0	MOVX A, @R1	CLR A	MOV A, iram	MOV A, @R0	MOV A, @R1	MOV A, R0	MOV A, R1	MOV A, R2	MOV A, R3	MOV A, R4	MOV A, R5	MOV A, R6	MOV A, R7
F	MOVX @DPTR, A	ACALL page7	MOVX @R0, A	MOVX @R1, A	CPL A	MOV iram, A	MOV @R0, A	MOV @R1, A	MOV R0, A	MOV R1, A	MOV R2, A	MOV R3, A	MOV R4, A	MOV R5, A	MOV R6, A	MOV R7, A

Recall: Type of Operands

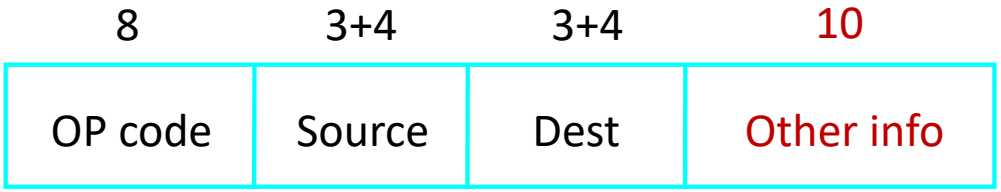
- **Addressing Modes**: the ways for specifying the locations of instruction operands.

Address Mode	Assembler Syntax	Addressing Function
1) Immediate	$\#Value$	$Operand = Value$
2) Register	Ri	$EA = Ri$
3) Absolute	LOC	$EA = LOC$
4) Register indirect	(Ri)	$EA = [Ri]$
5) Index	$X(Ri)$	$EA = [Ri] + X$
6) Base with index	(Ri, Rj)	$EA = [Ri] + [Rj]$

$Value$: a signed number
EA: the effective address of a register or a memory location
X: an index value

One-word Instruction (1/2)

- Some instructions can be encoded in one 32-bit word:



  - **OP code**: 8 bits
  - **Src** and **Dest**: 3 bits (addressing mode) + 4 bits (register #)
  - **Other info**: 10 bits (such as index value)

- **`ADD R1, R2`**
  - Needs to specify OP code, SRC and DEST registers.
    - 8 bits for OP code.
    - 3 bits are needed for addressing modes.
    - 4 bits are required to distinguish 16 registers.

- **`MOV R5, 24(R0)`**
  - Needs to specify OP code, two registers and an index value of 24.
    - 10 bits of other info can be used for the index value.

One-word Instruction (2/2)

- Some instructions can be encoded in one 32-bit word:

  - **OP code**: 8 bits
  - **Branch address**: 24 bits

- **`Branch>0  Offset`**
  - 8 bits for OP code
  - 24 bits are left for the branch address.

Question: How can we branch farther away?

Two-word Instruction

- What if we want to specify a memory operand using the absolute addressing mode?

`MOV R2, LOC`

  - 8 bits for OP code, 3+4 bits for addressing mode and register number for R2, 3 bits for addressing mode for LOC.
  - Only 14 bits left for specifying the memory address.

- Some instructions need an additional word to contain the absolute memory address or an immediate value:

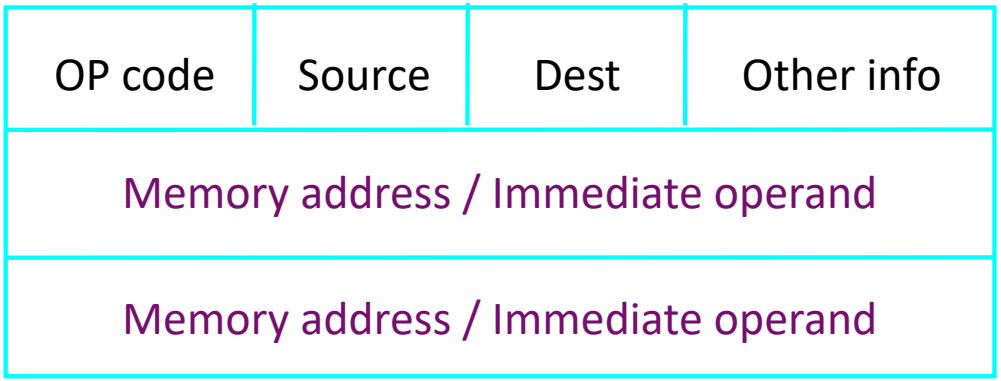
  - E.g., **`Add R2,`** **`FF000000`**$_h$ (immediate operand)

Multi-word Instruction (1/2)

- What if we want to allow an instruction in which both two operands can be specified using the absolute addressing mode?

```
MOV   LOC1, LOC2
```

- It becomes necessary to use two additional words for the 32-bit addresses of the two operands ...



Multi-word Instruction (2/2)

- If we allow instructions using two 32-bit direct address operands, we need three words in total for the instruction encoding scheme.
  - E.g., **`MOV  LOC1, LOC2`**

- Multiple length instructions are difficult to implement with high clock rate.
  - The design of the Instruction Register (IR) and the Instruction Decoder will be complex.
  - The Control Unit will be difficult to design.

- Shall we go for simple or complex?

Recall: RISC vs. CISC Styles

RISC	CISC
Simple addressing modes	More complex addressing modes
All instructions fitting in a single word	More complex instructions, where an instruction may span multiple words
Fewer instructions in the instruction set, and simpler addressing modes	Many instructions that implement complex tasks, and complicated addressing modes
Arithmetic and logic operations that can be performed only on operands in processor registers	Arithmetic and logic operations that can be performed on memory and register operands
Don't allow direct transfers from one memory location to another Note: Such transfers must take place via a processor register.	Possible to transfer from one memory location to another by using a single Move instruction
Programs that tend to be larger in size, because more but simpler instructions are needed to perform complex tasks	Programs that tend to be smaller in size, because fewer but more complex instructions are needed to perform complex tasks
Simple instructions that are conducive to fast execution by the processing unit using techniques such as pipelining	

CISC vs RISC

- CISC OR RISC?
  - CISC machines usually require less instructions to do something but have a lower clock rate …
  - RISC machines require more instructions to do something but have a higher clock rate…
- The Best of Both World: CISC WITH RISC
  - Modern processors usually combine the strengths of both CISC and RISC.
  - E.g., a CISC design with a RISC core:
    - Design a RISC-style core instruction decoder with high clock rates.
    - Provide a rich set of CISC-style instructions and addressing modes to assembly programmers.

Summary

- Control Signal Generation
  1) Hard-wired Control
  2) Micro-programmed Control

- Machine Instruction Encoding