Systems and Devices 1
Lec 5b : The Computer

Before we get started ...

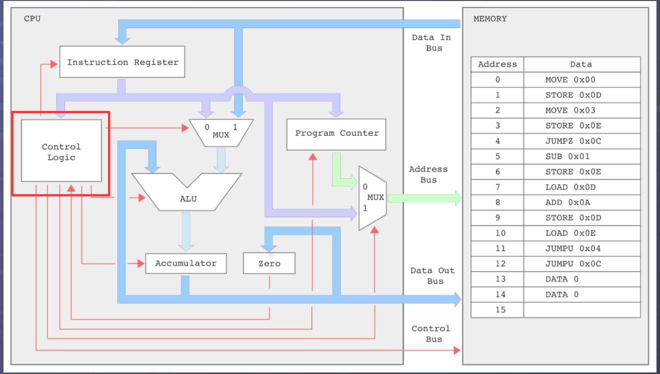
- We have seen how the processor will process an instruction (FDE). We know how instructions and data are stored in memory, BUT, how do we implement these instruction phases in hardware?
  - The missing link : what controls the hardware components in our processor i.e. registers, counters, ALU and memory?
    - For each machine-level instruction what are the required sequence of micro-instructions.
    - Need to identify what hardware is used, how is it controlled?

Instruction set

RTL	ENCODING	ASSEMBLER
Move KK : ACC <- KK	0000 XXXX KKKKKKKK	MOVE 0x01
Add KK : ACC <- ACC + KK	0001 XXXX KKKKKKKK	ADD 0x23
Sub KK : ACC <- ACC - KK	0010 XXXX KKKKKKKK	SUB 0x45
And KK : ACC <- ACC & KK	0011 XXXX KKKKKKKK	AND 0x67
Load AA : ACC <- M[AA]	0100 XXXX AAAAAAAA	LOAD 0x89
Store AA : M[AA] <- ACC	0101 XXXX AAAAAAAA	STORE 0x89
AddM AA : ACC <- ACC + M[AA]	0110 XXXX AAAAAAAA	ADDM 0xAB
SubM AA : ACC <- ACC - M[AA]	0111 XXXX AAAAAAAA	SUBM 0xAB
JumpU AA : PC <- AA	1000 XXXX AAAAAAAA	JUMPU 0xCD
JumpZ AA : IF Z=1 PC <- AA ELSE PC <- PC + 1	1001 XXXX AAAAAAAA	JUMPZ 0xEF
JumpNZ AA : IF Z=0 PC <- AA ELSE PC <- PC + 1	1010 XXXX AAAAAAAA	JUMPNZ 0xF0

- SimpleCPU machine-level instructions
  - Q : what are micro-instructions?

SimpleCPU_v1a



Address	Data
0	MOVE 0x00
1	STORE 0x0D
2	MOVE 0x03
3	STORE 0x0E
4	JUMPZ 0x0C
5	SUB 0x01
6	STORE 0x0E
7	LOAD 0x0D
8	ADD 0x0A
9	STORE 0x0D
10	LOAD 0x0E
11	JUMPU 0x04
12	JUMPU 0x0C
13	DATA 0
14	DATA 0
15	

- A : a micro-instruction defines the state of the control signals within the processor for a particular phase of a machine-level instructions execution.
  - Quick Quizzz : How many control signals are needed?

Slide 5

Control logic

INSTRUCTION	FETCH	DECODE	EXECUTE	PC_EN	PC_LD	IR_EN	ACC_EN	ACC_CTL2	ACC_CTL1	ACC_CTL0	ADDR_SEL	DATA_SEL	RAM_WR	RAM_EN	ROM_EN
MOVE	1	0	0	0	0	1	0	1	0	0	0	0	0	0	1
	0	1	0	1	0	0	0	1	0	0	0	0	0	0	0
	0	0	1	0	0	0	1	1	0	0	0	0	0	0	0
ADD	1	0	0	0	0	1	0	0	0	0	0	0	0	0	1
	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0
	0	0	1	0	0	0	1	0	0	0	0	0	0	0	0
SUB	1	0	0	0	0	1	0	0	0	1	0	0	0	0	1
	0	1	0	1	0	0	0	0	0	1	0	0	0	0	0
	0	0	1	0	0	0	1	0	0	1	0	0	0	0	0
AND	1	0	0	0	0	1	0	0	1	0	0	0	0	0	1
	0	1	0	1	0	0	0	0	1	0	0	0	0	0	0
	0	0	1	0	0	0	1	0	1	0	0	0	0	0	0
LOAD	1	0	0	0	0	1	0	1	0	0	0	1	0	0	1
	0	1	0	1	0	0	0	1	0	0	1	1	0	1	0
	0	0	1	0	0	0	1	1	0	0	1	1	0	1	0
STORE	1	0	0	0	0	1	0	0	0	0	0	0	0	0	1
	0	1	0	1	0	0	0	0	0	0	1	0	1	1	0
	0	0	1	0	0	0	0	0	0	0	1	0	0	1	0
ADDM	1	0	0	0	0	1	0	0	0	0	0	1	0	0	1
	0	1	0	1	0	0	0	0	0	0	1	1	0	1	0
	0	0	1	0	0	0	1	0	0	0	1	1	0	1	0
SUBM	1	0	0	0	0	1	0	0	0	1	0	1	0	0	1
	0	1	0	1	0	0	0	0	0	1	1	1	0	1	0
	0	0	1	0	0	0	1	0	0	1	1	1	0	1	0
JUMPU	1	0	0	0	0	1	0	0	0	0	0	0	0	0	1
	0	1	0	!J	0	0	0	0	0	0	0	0	0	0	0
	0	0	1	J	1	0	0	0	0	0	0	0	0	0	0
JUMPZ	1	0	0	0	0	1	0	0	0	0	0	0	0	0	1
	0	1	0	!J	0	0	0	0	0	0	0	0	0	0	0
	0	0	1	J	Z	0	0	0	0	0	0	0	0	0	0
JUMPNZ	1	0	0	0	0	1	0	0	0	0	0	0	0	0	1
	0	1	0	!J		0	0	0	0	0	0	0	0	0	0
	0	0	1	J	!Z	0	0	0	0	0	0	0	0	0	0

NOTE: J = (JUMPU # (JUMPZ & Z) & (JUMPNZ & !Z)), Z = ZERO   LOGIC SYMBOLS: AND = &, OR = #, NOT = !

- Step 1 : for each instruction identify the signals to control and their state at each phase (FDE).

Slide 6

Control logic

CTL SIGNAL	LOGIC
ROM_EN	FETCH
RAM_EN	(DECODE # EXECUTE) & (LOAD # STORE # ADDM # SUBM)
RAM_WR	STORE & EXECUTE
ADDR_SEL	(DECODE # EXECUTE) & (LOAD # STORE # ADDM # SUBM)
DATA_SEL	LOAD # ADDM # SUBM
ACC_CTL0	SUB # SUBM
ACC_CTL1	AND
ACC_CTL2	MOVE # LOAD
ACC_EN	(MOVE # ADD # SUB # AND # LOAD # ADDM # SUBM) & EXECUTE
IR_EN	FETCH
PC_LD	EXECUTE & J
PC_EN	(DECODE & !J) # (EXECUTE & J)

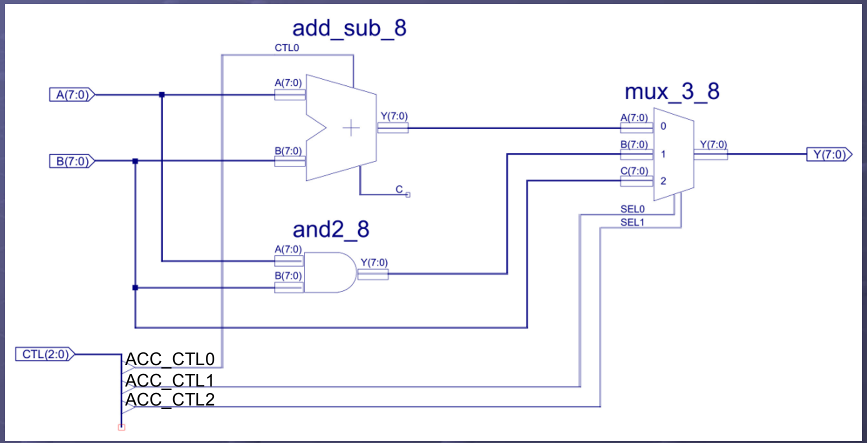
NOTE: J = (JUMPU # (JUMPZ & Z) & (JUMPNZ & !Z)), Z = ZERO

LOGIC SYMBOLS: AND = &, OR = #, NOT = !

- Step 2 : convert truth table into combinatorial logic circuit (Boolean equations, ABEL, VHDL).

Slide 7

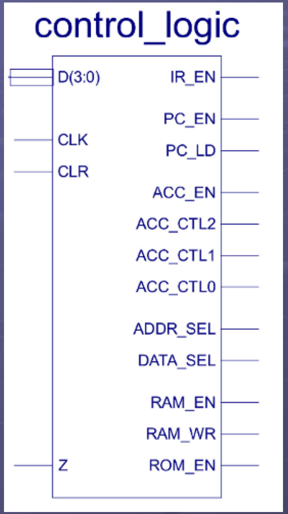
Control logic : ALU

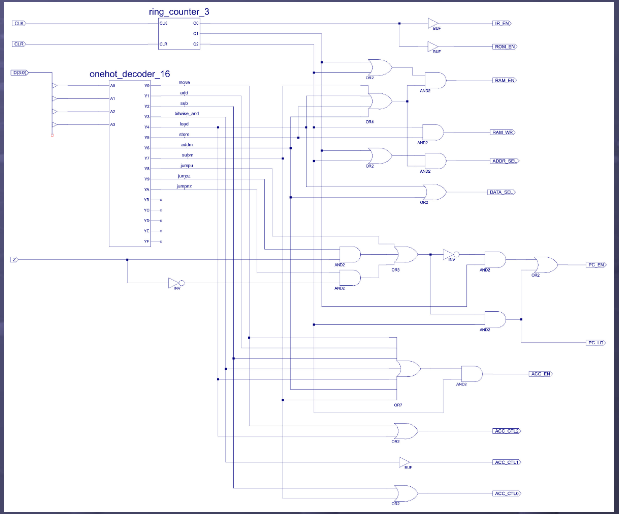


- ADD / ADDM = “000”   SUB / SUBM = “001”
- BITWISE_AND = “01X”   MOVE / LOAD = “10X”

Slide 8

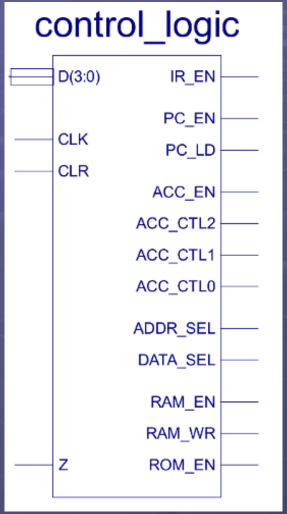
Control logic

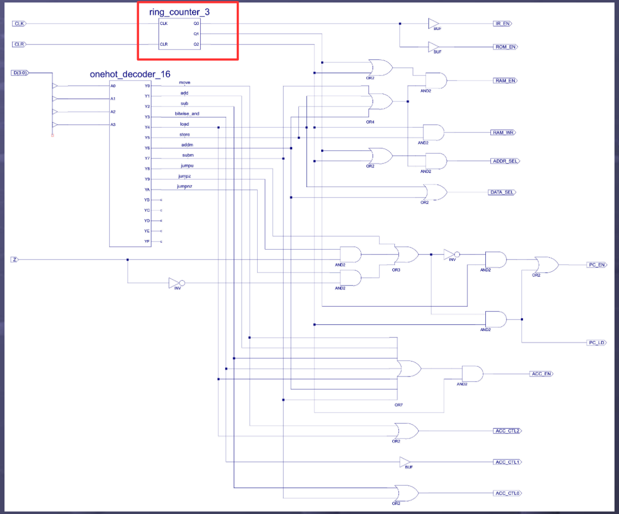




- Step 3 : implementation – ring counter, encoder, logic

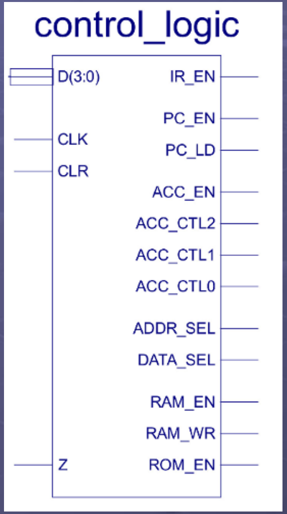
Control logic : FDE

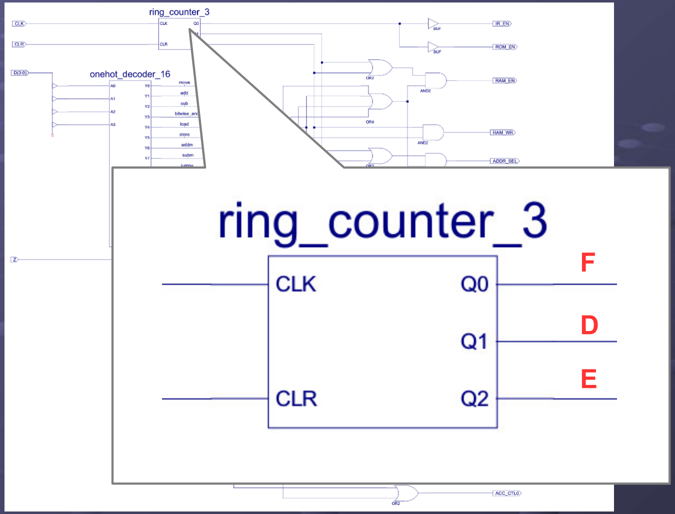




- 3bit ring counter : representing instruction phases (FDE)

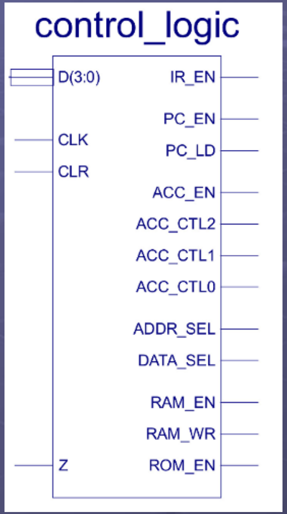
Control logic : FDE

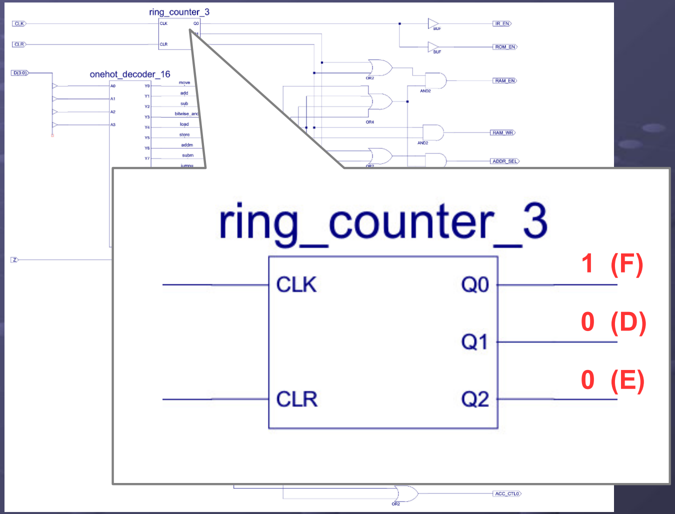




- 3bit ring counter : representing instruction phases (FDE)

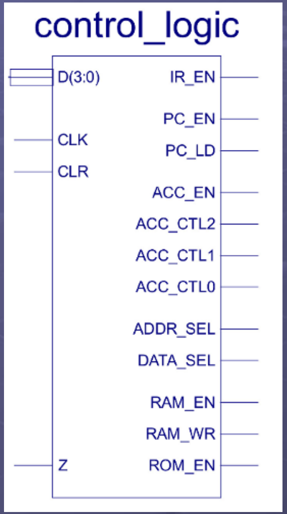
Control logic : FDE

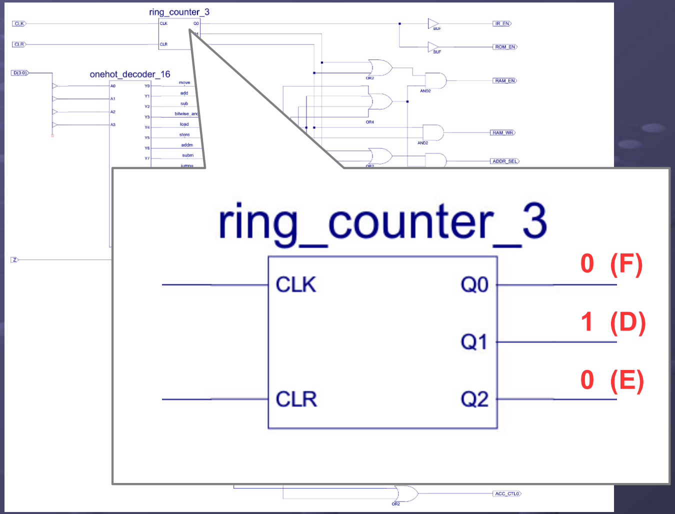




- 3bit ring counter : representing instruction phases (FDE)

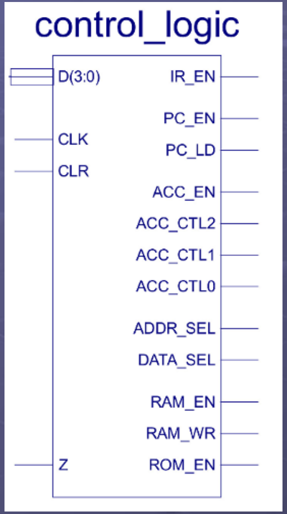
Control logic : FDE

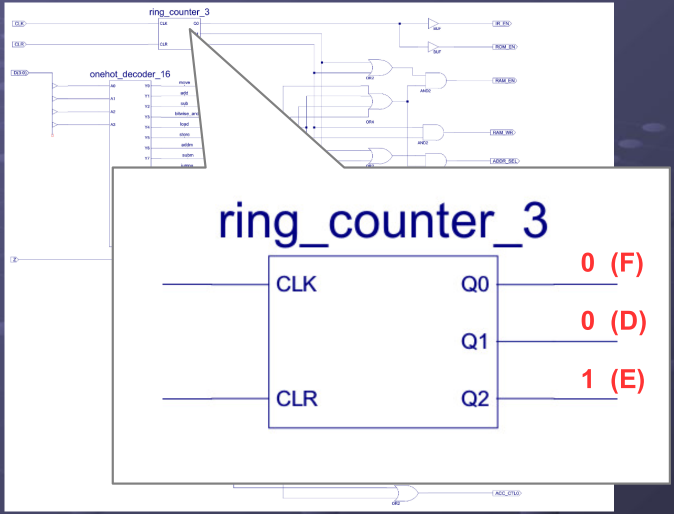




- 3bit ring counter : representing instruction phases (FDE)

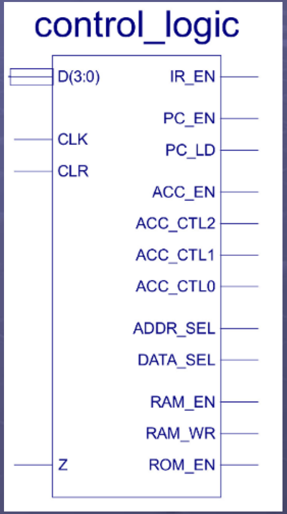
Control logic : FDE

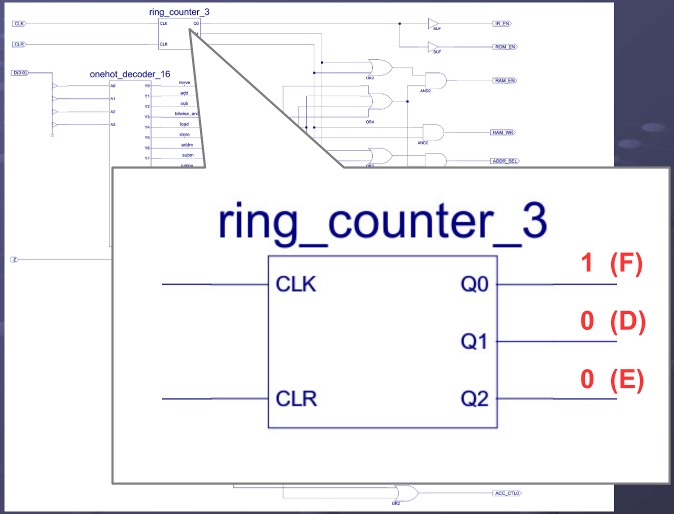




- 3bit ring counter : representing instruction phases (FDE)

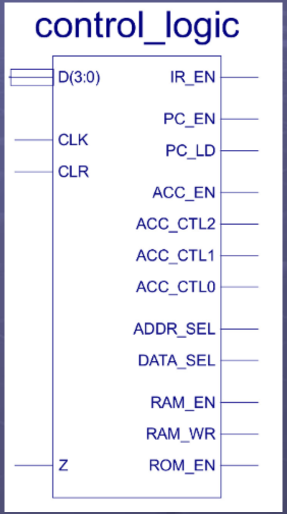
Control logic : FDE

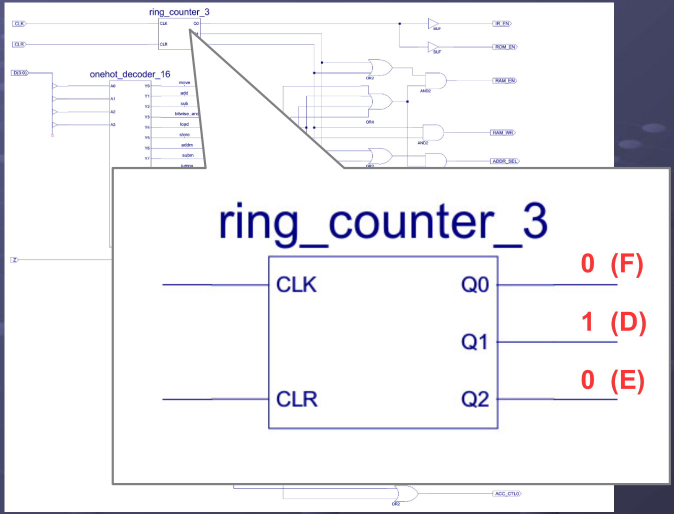




- 3bit ring counter : representing instruction phases (FDE)

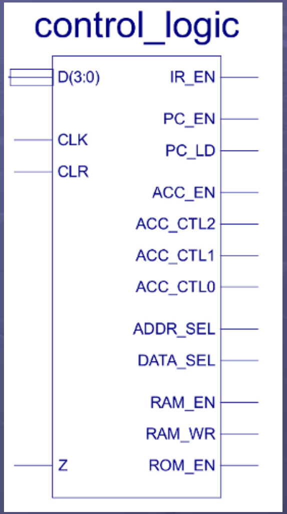
Control logic : FDE

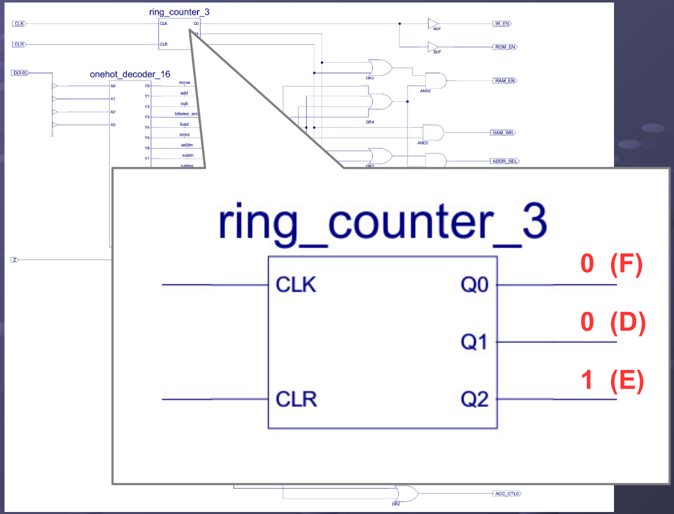




- 3bit ring counter : representing instruction phases (FDE)

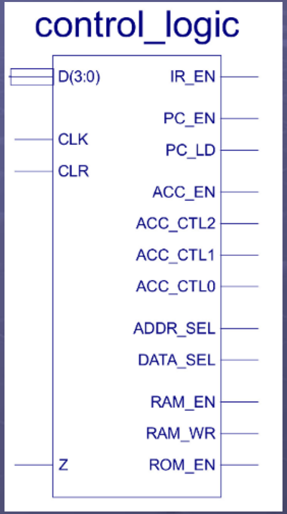
Control logic : FDE

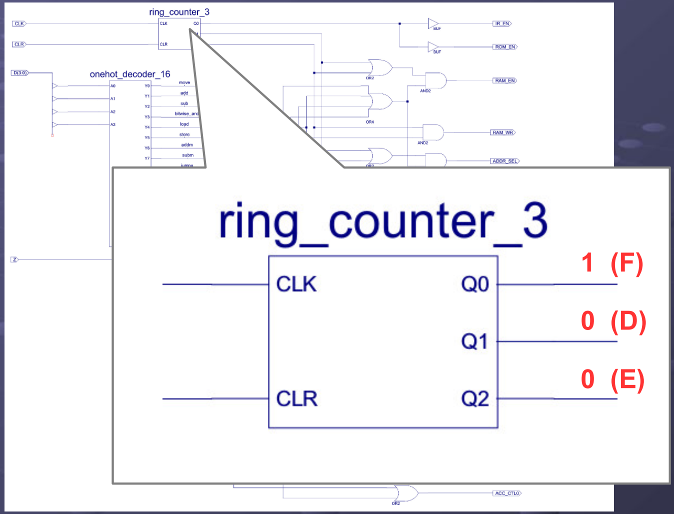




- 3bit ring counter : representing instruction phases (FDE)

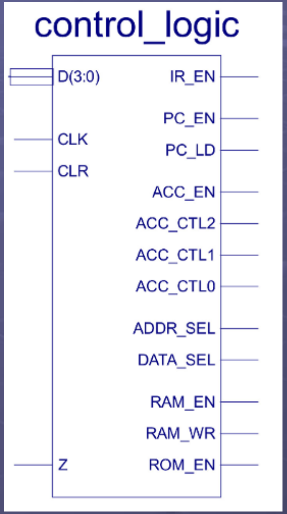
Control logic : FDE

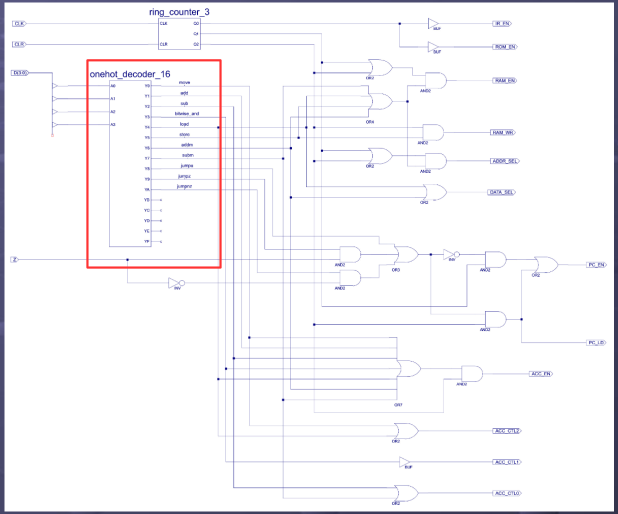




- 3bit ring counter : representing instruction phases (FDE)

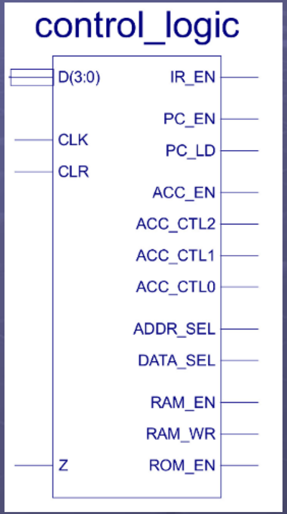
Control logic : Decoder

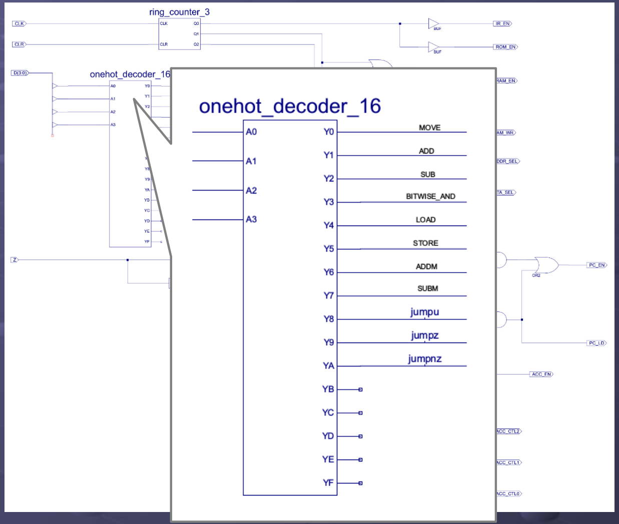




- One-hot decoder : process 4 bit opcode field

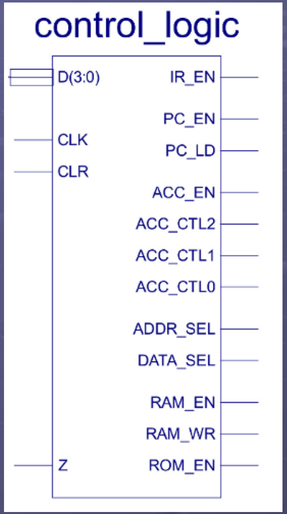
Control logic : Decoder

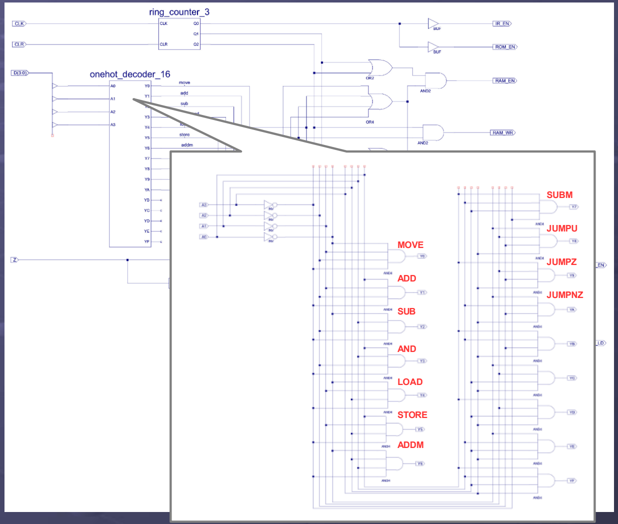




- One-hot decoder : process 4 bit opcode field

Control logic : Decoder

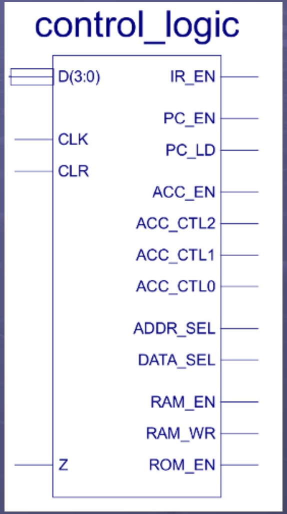


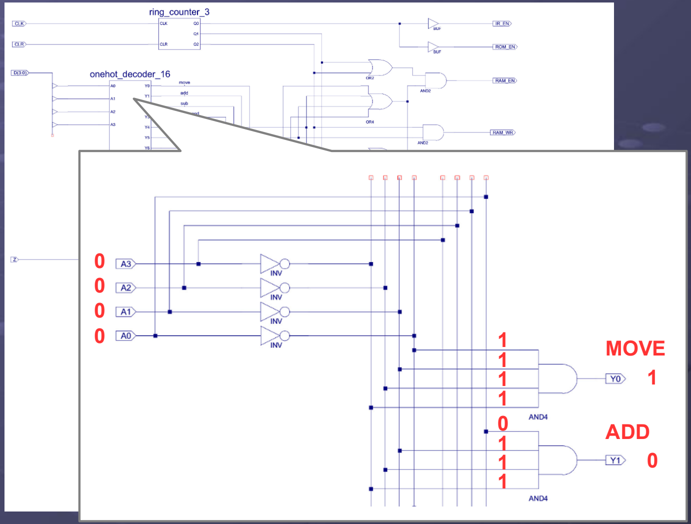


- One-hot decoder : process 4 bit opcode field

Slide 21

Control logic : Decoder

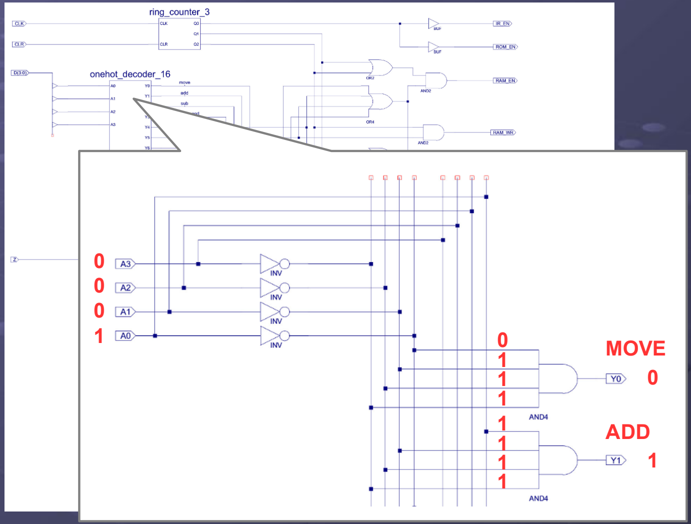




- One-hot decoder : process 4 bit opcode field

Slide 22

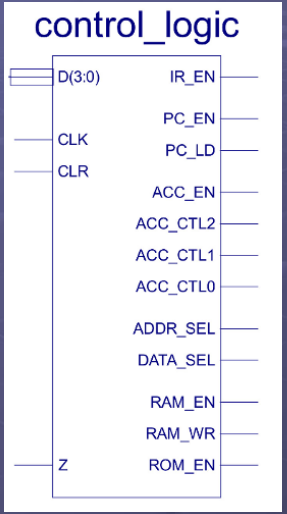
Control logic : Decoder

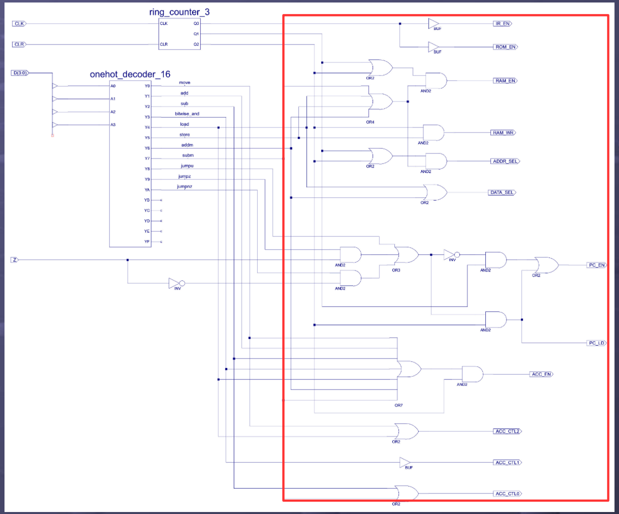


- One-hot decoder : process 4 bit opcode field

Slide 23

Control logic

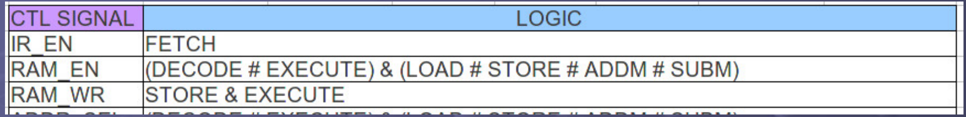




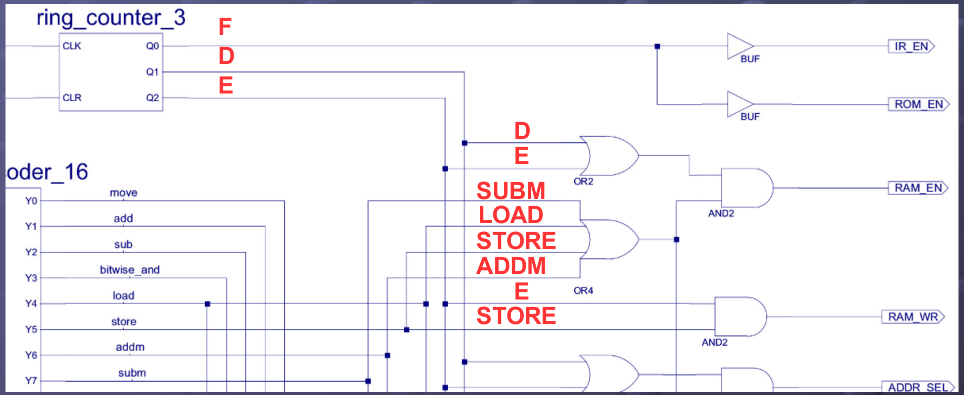
- Decode logic : generate control signals

Slide 24

Control logic

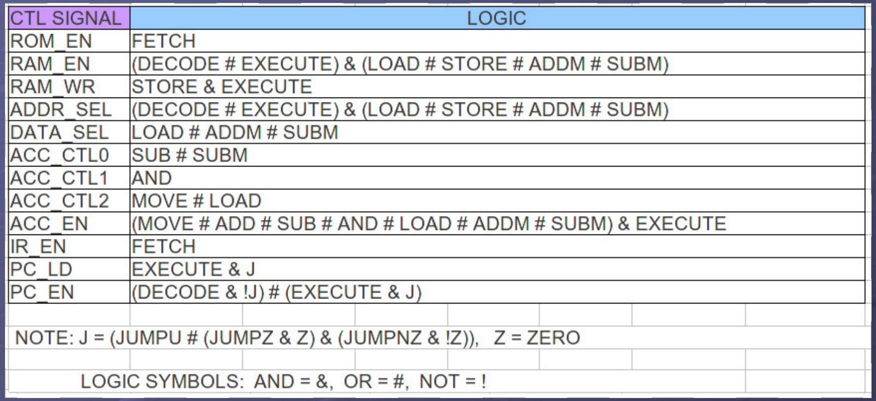


CTL SIGNAL	LOGIC
IR_EN	FETCH
RAM_EN	(DECODE # EXECUTE) & (LOAD # STORE # ADDM # SUBM)
RAM_WR	STORE & EXECUTE



- Decode logic implementation

Control logic



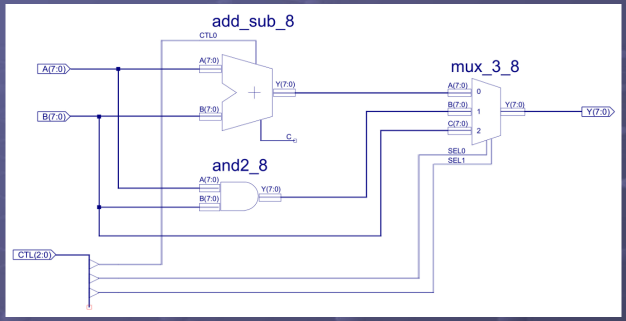
CTL SIGNAL	LOGIC
ROM_EN	FETCH
RAM_EN	(DECODE # EXECUTE) & (LOAD # STORE # ADDM # SUBM)
RAM_WR	STORE & EXECUTE
ADDR_SEL	(DECODE # EXECUTE) & (LOAD # STORE # ADDM # SUBM)
DATA_SEL	LOAD # ADDM # SUBM
ACC_CTL0	SUB # SUBM
ACC_CTL1	AND
ACC_CTL2	MOVE # LOAD
ACC_EN	(MOVE # ADD # SUB # AND # LOAD # ADDM # SUBM) & EXECUTE
IR_EN	FETCH
PC_LD	EXECUTE & J
PC_EN	(DECODE & !J) # (EXECUTE & J)

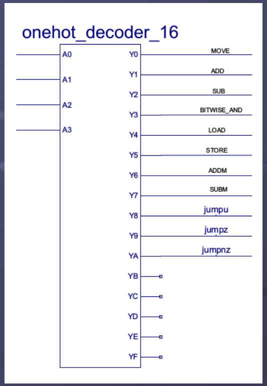
NOTE: J = (JUMPU # (JUMPZ & Z) & (JUMPNZ & !Z)), Z = ZERO

LOGIC SYMBOLS: AND = &, OR = #, NOT = !

- Quick Quizzz
  - Draw the control logic to implement the ALU control lines: ACC_CTL0/1/2. What is the input source for this circuit?

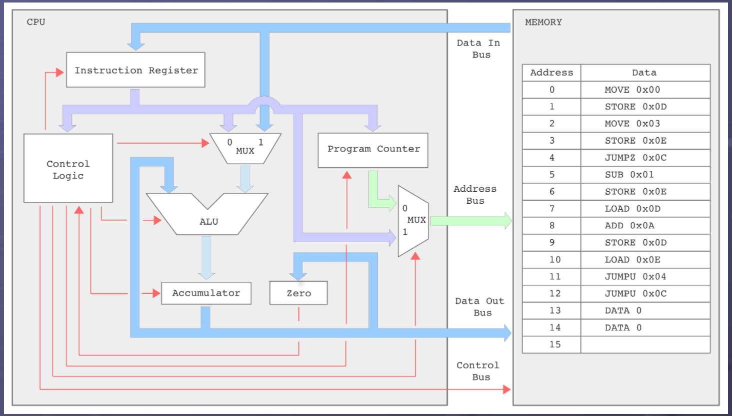
New instructions?





- Quick Quizzz
  - What would you need to do to add an immediate multiply instruction: `ACC <- ACC × KK`?
  - What problem could occur when storing the result?

Memory



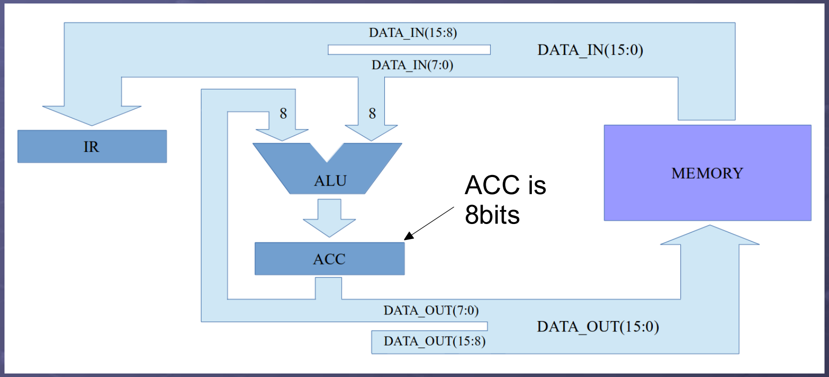
Address	Data
0	MOVE 0x00
1	STORE 0x0D
2	MOVE 0x03
3	STORE 0x0E
4	JUMPZ 0x0C
5	SUB 0x01
6	STORE 0x0E
7	LOAD 0x0D
8	ADD 0x0A
9	STORE 0x0D
10	LOAD 0x0E
11	JUMPU 0x04
12	JUMPU 0x0C
13	DATA 0
14	DATA 0
15	

- Q : if the ACC is 8bits wide and memory stores 16bit values e.g. an instruction. What happens during LOAD or STORE instructions?

Memory

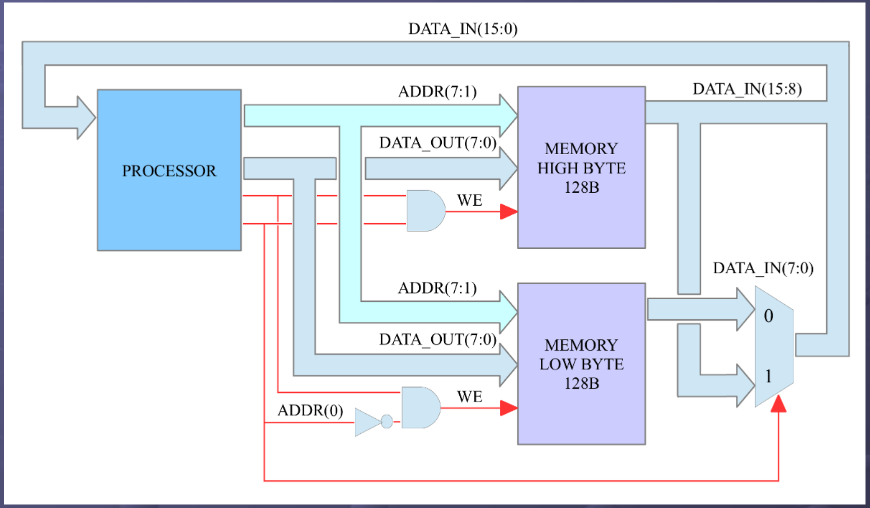
- A : it depends on the architectural choices we make.
  - 1) Make everything 16-bits : upgrade to an 16-bit ACC and ALU to, downside wastes hardware if we do not need to process 16-bit data types.
  - 2) Only read and write to lower 8-bits of a memory locations, downside wastes memory i.e. each time you declare a variable we will waste 8-bits.
  - 3) Make memory 8-bits wide, processor can use any memory location to store variables, downside instructions have to be stored in two memory locations.
    - Quick Quizz : how will this affect the processor's FETCH phase i.e. address bus, PC, IR, data bus?

Memory



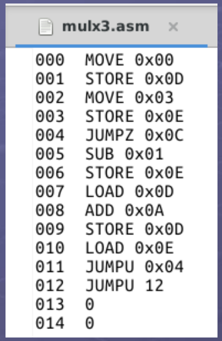
- For the simpleCPU_v1a we take the simple solution
  - Only read and write to lower 8-bits of a memory locations, downside wastes memory i.e. each time you declare a variable we will waste 8-bits.

Memory



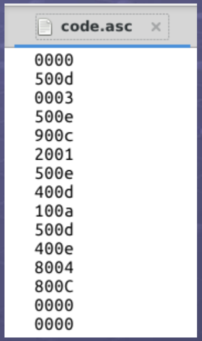
- Alternatively we could use byte addressable memory

Memory



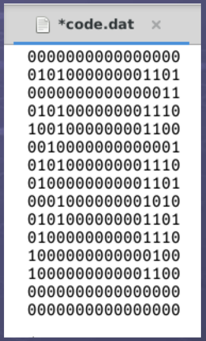
mulx3.asm

```
000  MOVE 0x00
001  STORE 0x0D
002  MOVE 0x03
003  STORE 0x0E
004  JUMPZ 0x0C
005  SUB 0x01
006  STORE 0x0E
007  LOAD 0x0D
008  ADD 0x0A
009  STORE 0x0D
010  LOAD 0x0E
011  JUMPU 0x04
012  JUMPU 12
013  0
014  0
```



code.asc

```
0000
500d
0003
500e
900c
2001
500e
400d
100a
500d
400e
8004
800C
0000
0000
```



*code.dat

```
0000000000000000
0101000000001101
0000000000000011
0101000000001110
1001000000001100
0010000000000001
0101000000001110
0100000000001101
0001000000001010
0101000000001101
0100000000001110
1000000000000100
1000000000001100
0000000000000000
0000000000000000
```

00	00
01	00
02	0D
03	50
04	03
05	00
06	0E
07	50
08	??
09	??
0A	??
0B	??
...	...

00	00	00
02	50	1A
04	00	03
06	50	1B
08	90	18
0A	20	01
0C	50	1B
0E	40	1A
10	10	0A
12	50	1A
14	40	1B
16	80	08
18	80	18
1A	??	??

- Quick Quizz
  - How will the MULx3 program be stored in byte addressable memory? Will it change? What byte do you store “first”?
  - When fetching an instruction what will be the value of ADDR(0)?

Summary

- Key concepts
  - Control logic
    - Representing processor state
      - Ring counter
    - Generating control signals
      - One-hot decoder + logic
  - Memory architecture
    - Word or Byte addressable memory
    - Endianness : little=LSD or Big=MSD in first address.
  - Self-modifying code
    - Treating instructions as data, overwriting instructions as the program is executed.
      - Not a recommended programming technique :)