微算機系統 第五章
The Processor: Datapath and Control

陳伯寧 教授
電信工程學系
國立交通大學

The processor: Datapath and control

- *We're ready to look at an implementation of the MIPS*
- *The **target instruction set** is simplified to contain only:*
  - *memory-reference instructions:*
    `lw, sw`
  - *arithmetic-logical instructions:*
    `add, sub, and, or, slt`
  - *control flow instructions:*
    `beq, j`

The processor: Datapath and control

- *Generic Implementation:*
  - *use the program counter (PC) to supply instruction address*
  - *get the instruction from memory (stored program concept)*
  - *read registers*
  - *use the instruction to decide exactly what to do*

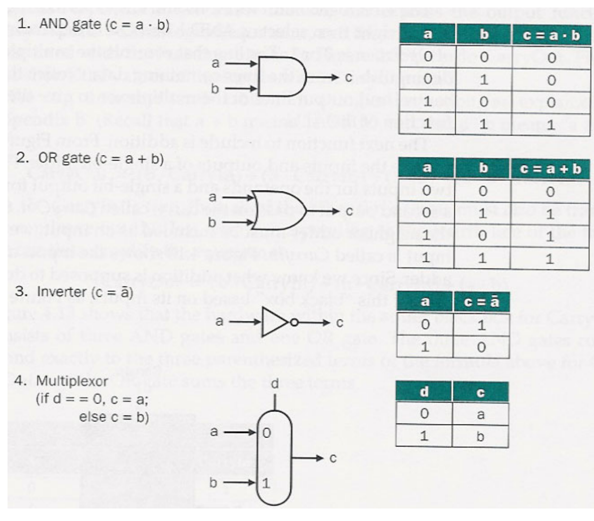
The processor: Datapath and control

- *All instructions (including* memory-reference, arithmetic, control flow) *use the ALU after reading the registers* Why?
  - *The memory-reference instructions use the ALU for an address calculation,*
  - *the arithmetic-logical instructions for the operation execution,*
  - *and branch instruction for comparison.*

Appendix B: Construction of an arithmetic logic unit

- *How to implement addition and subtraction, as well as multiplication and division?*
  - *Show the truth table for these functions.*
  - *Show the Boolean equations for these functions.*
  - *Show an implementation consisting of* ***Inverters****,* ***AND****, and* ***OR*** *gates, and* ***multiplexor (mux)****.*

Four basic hardware building blocks used to construct an ALU



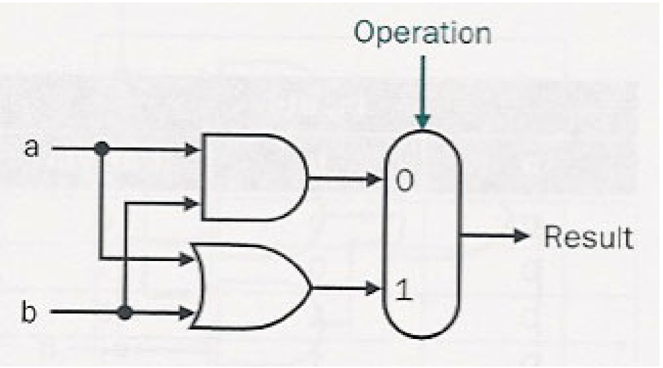
a	b	c = a · b
0	0	0
0	1	0
1	0	0
1	1	1

a	b	c = a + b
0	0	0
0	1	1
1	0	1
1	1	1

a	c = ā
0	1
1	0

d	c
0	a
1	b

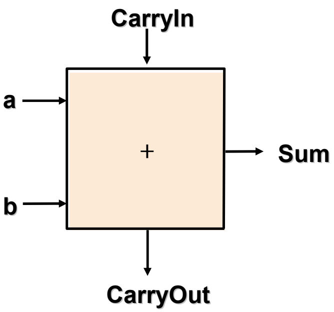
Example. 1-bit ALU that is capable of performing AND and OR



Truth table for 1-bit full adder or (3,2) adder, where 3 = # of inputs, and 2 = # of outputs

Inputs			Outputs		Comments
a	b	CarryIn	CarryOut	Sum	
0	0	0	0	0	0+0+0=00
0	0	1	0	1	0+1+0=01
0	1	0	0	1	0+1+0=01
0	1	1	1	0	0+1+1=10
1	0	0	0	1	1+0+0=01
1	0	1	1	0	1+0+1=10
1	1	0	1	0	1+1+0=10
1	1	1	1	1	1+1+1=11

Boolean equations for 1-bit full adder or (3,2) adder (sum-of-product representation)

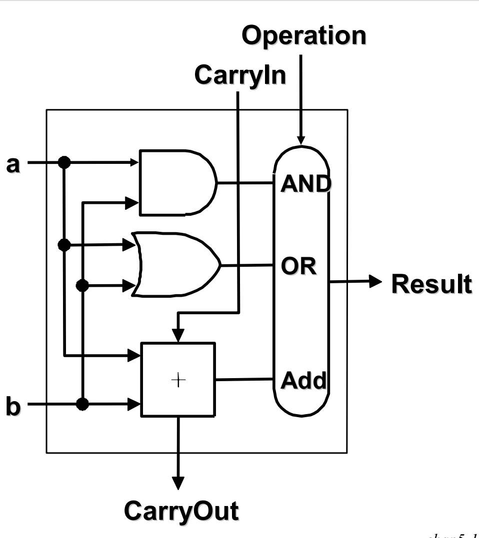


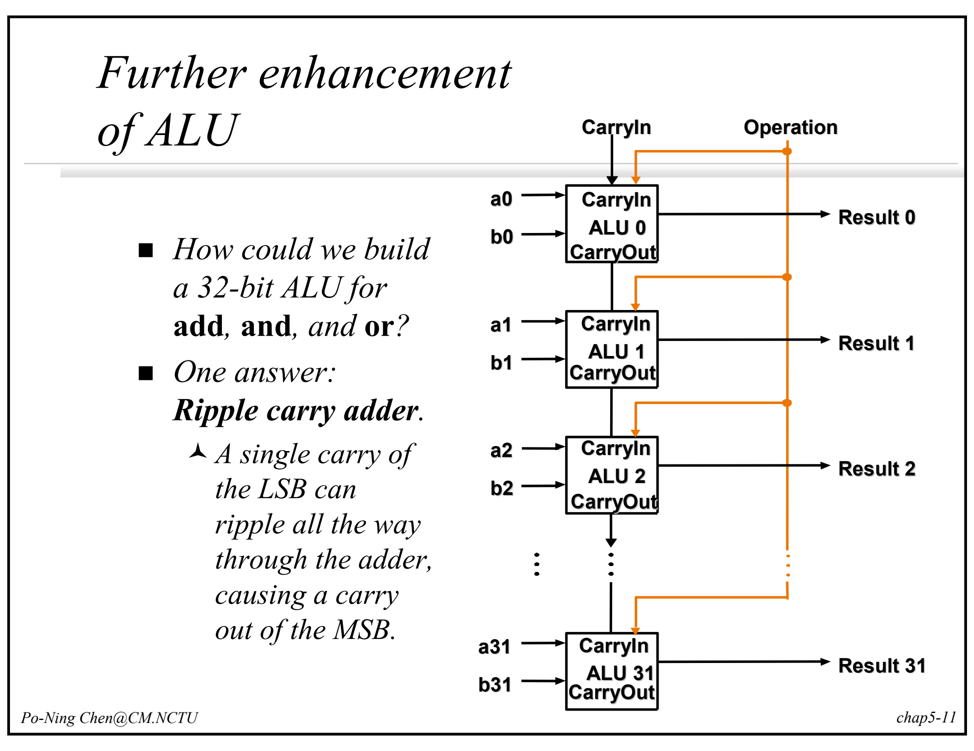
$$Sum = a \cdot \overline{b} \cdot \overline{CarryIn} + \overline{a} \cdot b \cdot \overline{CarryIn} + \overline{a} \cdot \overline{b} \cdot CarryIn + a \cdot b \cdot CarryIn$$

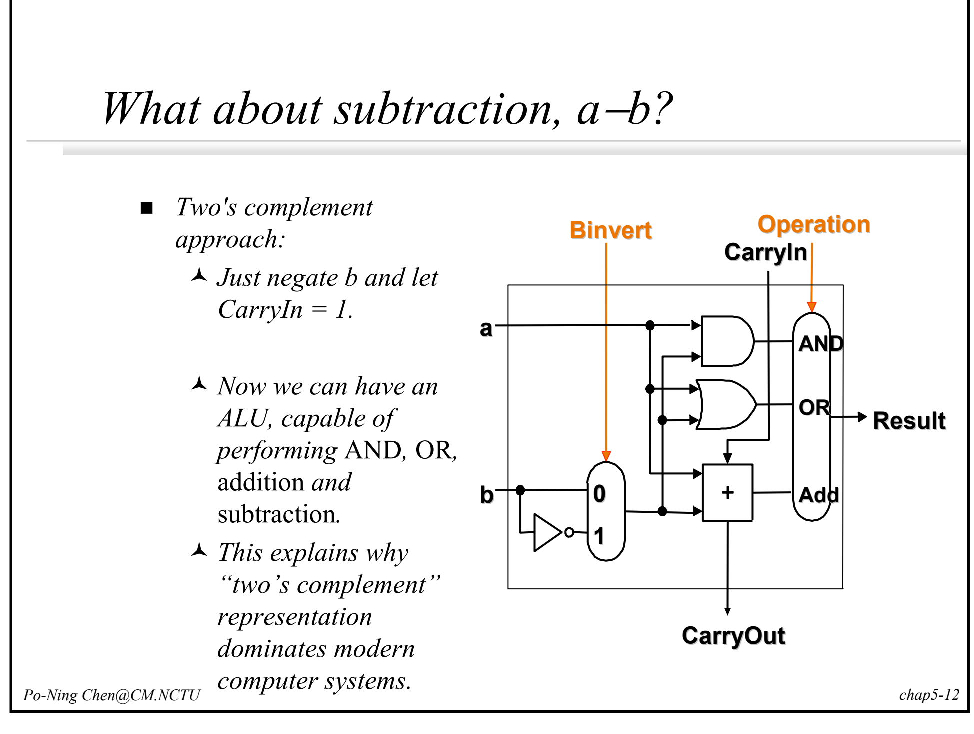
$$CarryOut = a \cdot CarryIn + b \cdot CarryIn + a \cdot b$$

Enhancement of ALU

- *How could we build a 1-bit ALU for* **add**, **and**, *and* **or**?




Further enhancement of ALU
How could we build a 32-bit ALU for add, and, and or?
One answer: Ripple carry adder.
A single carry of the LSB can ripple all the way through the adder, causing a carry out of the MSB.
CarryIn
Operation
a0
b0
CarryIn
ALU 0
CarryOut
Result 0
a1
b1
CarryIn
ALU 1
CarryOut
Result 1
a2
b2
CarryIn
ALU 2
CarryOut
Result 2
a31
b31
CarryIn
ALU 31
CarryOut
Result 31
Po-Ning Chen@CM.NCTU
chap5-11

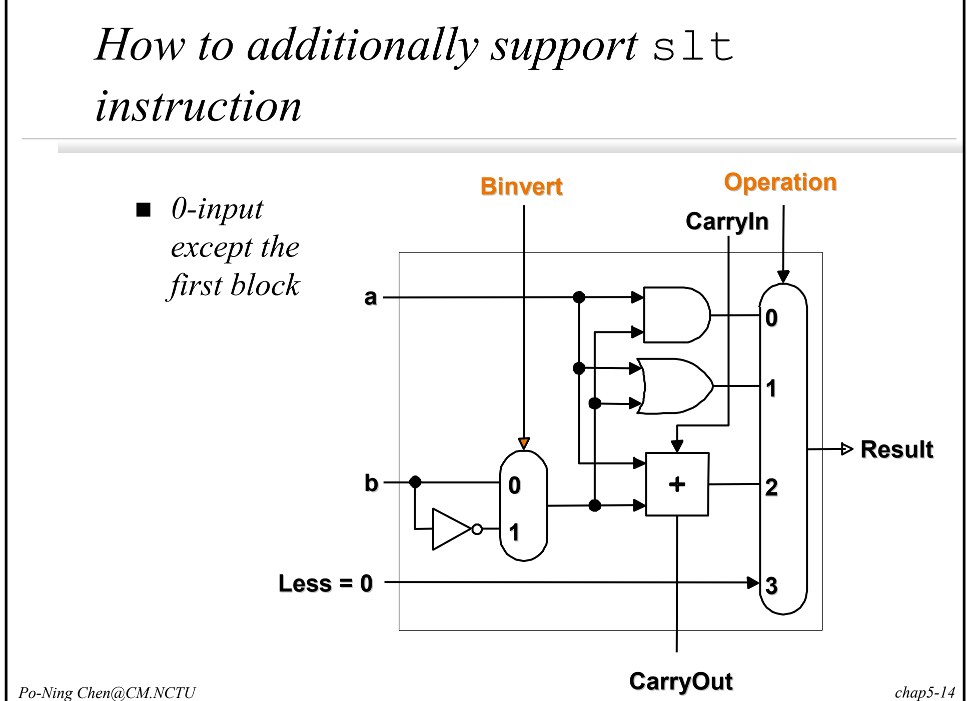

What about subtraction, a–b?
Two's complement approach:
Just negate b and let CarryIn = 1.
Now we can have an ALU, capable of performing AND, OR, addition and subtraction.
This explains why "two's complement" representation dominates modern computer systems.
Binvert
Operation
CarryIn
a
b
0
1
+
AND
OR
Add
Result
CarryOut
Po-Ning Chen@CM.NCTU
chap5-12

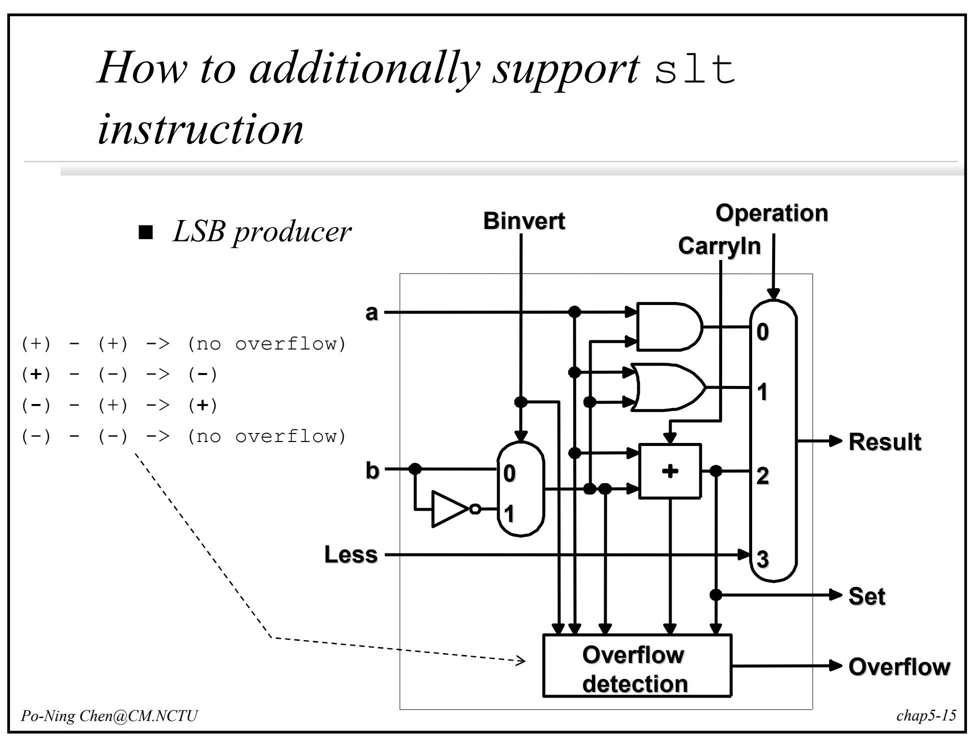
How to additionally support `slt` *instruction*

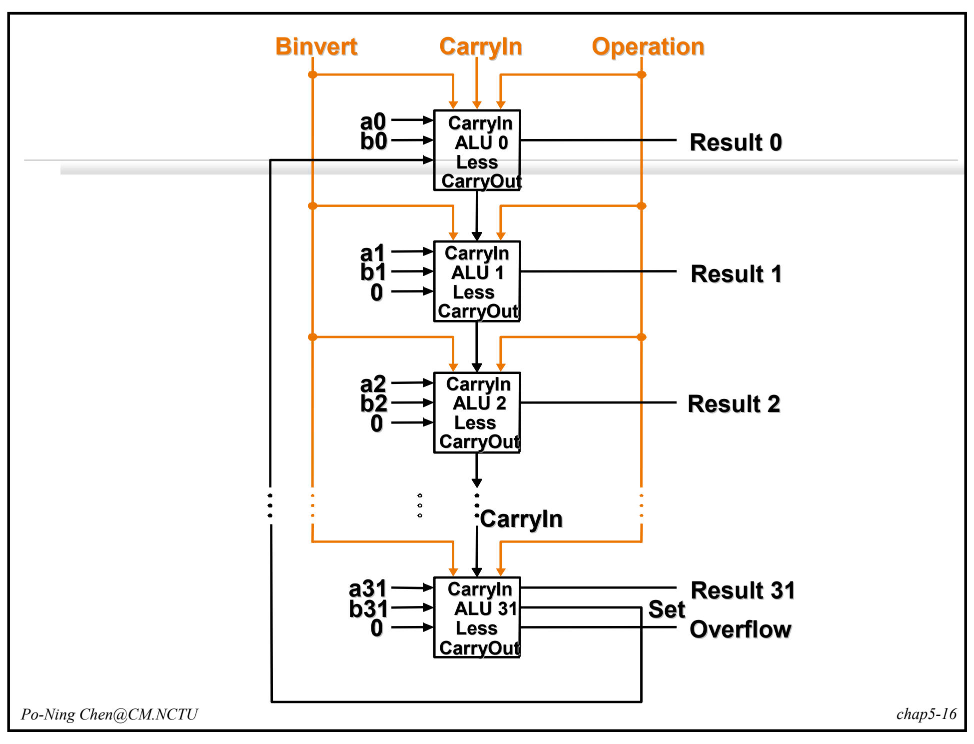
- *Remember: slt is an arithmetic instruction*
- *Use subtraction:* a - b < *0 implies* a < b.
  - *All the bits of the slt result equals 0, except possibly LSB*
    - *So we need 31 blocks with 0 input*
  - *LSB of slt result = 1, if* \$rs < \$rt, *and 0 otherwise*
    - *So we need 1 block of LSB producer*

How to additionally support `slt` *instruction*

- *0-input except the first block*




How to additionally support slt instruction
LSB producer
(+) - (+) -> (no overflow)
(+) - (-) -> (-)
(-) - (+) -> (+)
(-) - (-) -> (no overflow)
Binvert
Operation
CarryIn
a
b
0
1
2
3
Less
Result
Set
Overflow
Overflow detection
+
Po-Ning Chen@CM.NCTU
chap5-15


Binvert
CarryIn
Operation
a0
b0
CarryIn
ALU 0
Less
CarryOut
Result 0
a1
b1
0
ALU 1
Result 1
a2
b2
0
ALU 2
Result 2
CarryIn
a31
b31
0
ALU 31
Result 31
Set
Overflow
Po-Ning Chen@CM.NCTU
chap5-16

Relation between the ALU design and machine language

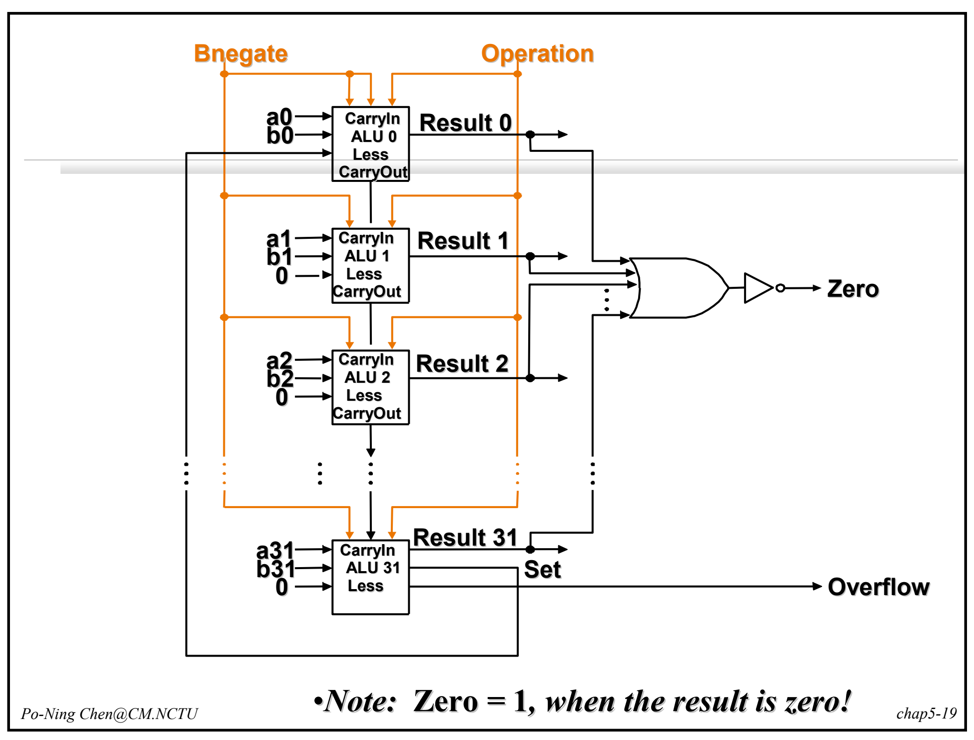
- *Control lines:*

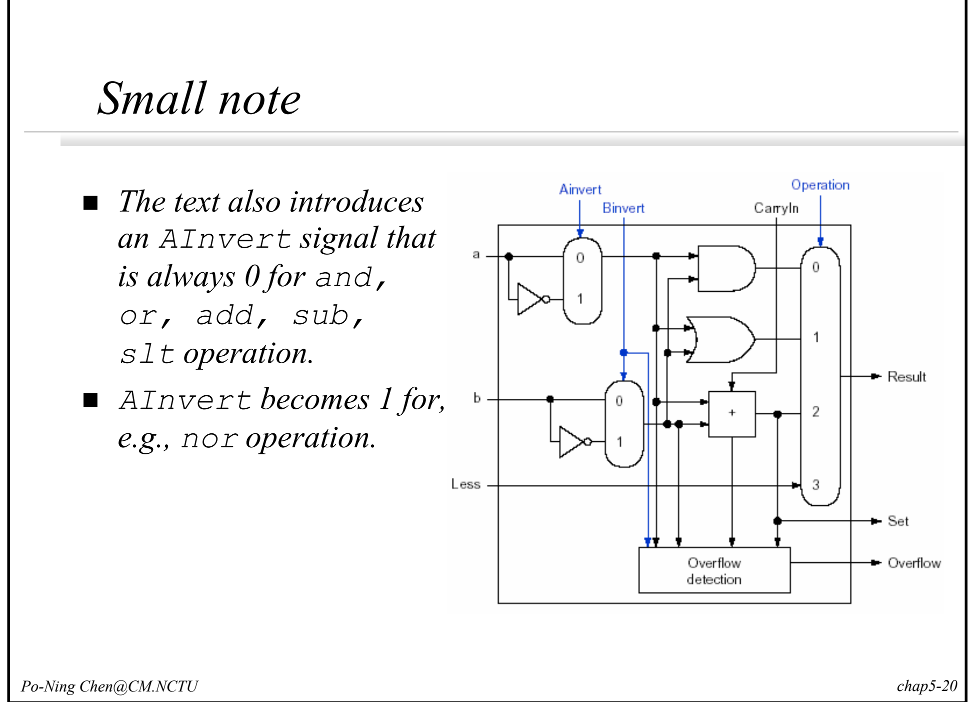
Instructions	*BInvert*	*CarryIn*	*Operation*
AND	*0*	*0*	*00*
OR	*0*	*0*	*01*
ADD	*0*	*0*	*10*
SUB	*1*	*1*	*10*
SLT	*1*	*1*	*11*

These two bits can be combined into one, named *Bnegate*.

Support of conditional branch instruction

- *At last, an ALU should support conditional branch instructions such as bne and beq.*
- *Hence, we need an additional "test" on equality of zero.*


Bnegate
Operation
a0
b0
CarryIn
ALU 0
Less
CarryOut
Result 0
a1
b1
0
CarryIn
ALU 1
Less
CarryOut
Result 1
a2
b2
0
CarryIn
ALU 2
Less
CarryOut
Result 2
Zero
a31
b31
0
CarryIn
ALU 31
Less
Result 31
Set
Overflow
•Note: Zero = 1, when the result is zero!
Po-Ning Chen@CM.NCTU
chap5-19


Small note
The text also introduces an AInvert signal that is always 0 for and, or, add, sub, slt operation.
AInvert becomes 1 for, e.g., nor operation.
Ainvert
Binvert
CarryIn
Operation
a
0
1
b
0
1
0
1
2
3
+
Less
Result
Set
Overflow
Overflow detection
Po-Ning Chen@CM.NCTU
chap5-20

Relation of the ALU design with machine language

- *Control lines:*

Instructions	*AInvert*	***BInvert***	***CarryIn***	*Operation*
AND	*0*	***0***	***0***	*00*
OR	*0*	***0***	***0***	*01*
ADD	*0*	***0***	***0***	*10*
SUB	*0*	***1***	***1***	*10*
SLT	*0*	***1***	***1***	*11*
NOR	*1*	***1***	***1***	*00*

These two bits can be combined into one, named *Bnegate*.

Partial Summary

- *We can build an ALU to support the MIPS instruction set*
  - *Key idea: Use **multiplexor** to select the output we want*
  - *We can efficiently perform subtraction using **two's complement***
  - *We can **replicate** a 1-bit ALU to produce a 32-bit ALU*

Important points about hardware

- *All of the gates are **always working***
- *The speed of a gate is affected by the **number of inputs** to the gate*
- *The speed of a circuit is affected by the number of gates **in series** (on the "critical path" or the "deepest level of logic")*
  - *Accordingly, the 32-bit ripple carry adder is slow since result31 shall wait for 31 carry ripples.*

Speed concern of ALU

- *Clever changes to organization can improve performance (similar to using better algorithms in software)*
- *Let's look at an example for addition.*

Speed concern of ALU

- *Can a 32-bit ALU be as fast as a 1-bit ALU?*
- *Answer: Yes, e.g., using a* ***sum-of-products adder****.*
  - *With the following equations, we obtain the equation on next slide.*

$$c_1 = b_0c_0 + a_0c_0 + a_0b_0$$
$$c_2 = b_1c_1 + a_1c_1 + a_1b_1$$
$$c_3 = b_2c_2 + a_2c_2 + a_2b_2$$
$$c_4 = b_3c_3 + a_3c_3 + a_3b_3$$
$$\vdots$$

where c_j = $CarryIn_j$ = $CarryOut_{j-1}$.

Speed concern of ALU

- *c2 = a0 a1 b0 + a1 b1 + a0 b0 b1 + a0 a1 c0 + a1 b0 c0 + a0 b1 c0 + b0 b1 c0*
- *c3 = a0 a1 a2 b0 + a1 a2 b1 + a0 a2 b0 b1 + a2 b2 + a0 a1 b0 b2 + a1 b1 b2 + a0 b0 b1 b2 + a0 a1 a2 c0 + a1 a2 b0 c0 + a0 a2 b1 c0 + a2 b0 b1 c0 + a0 a1 b2 c0 + a1 b0 b2 c0 + a0 b1 b2 c0 + b0 b1 b2 c0*
- *c4 = a0 a1 a2 a3 b0 + a1 a2 a3 b1 + a0 a2 a3 b0 b1 + a2 a3 b2 + a0 a1 a3 b0 b2 + a1 a3 b1 b2 + a0 a3 b0 b1 b2 + a3 b3 + a0 a1 a2 b0 b3 + a1 a2 b1 b3 + a0 a2 b0 b1 b3 + a2 b2 b3 + a0 a1 b0 b2 b3 + a1 b1 b2 b3 + a0 b0 b1 b2 b3 + a0 a1 a2 a3 c0 + a1 a2 a3 b0 c0 + a0 a2 a3 b1 c0 + a2 a3 b0 b1 c0 + a0 a1 a3 b2 c0 + a1 a3 b0 b2 c0 + a0 a3 b1 b2 c0 + a3 b0 b1 b2 c0 + a0 a1 a2 b3 c0 + a1 a2 b0 b3 c0 + a0 a2 b1 b3 c0 + a2 b0 b1 b3 c0 + a0 a1 b2 b3 c0 + a1 b0 b2 b3 c0 + a0 b1 b2 b3 c0 + b0 b1 b2 b3 c0*
- *c5 = ….* ***Exponentially increasing complexity!***

Carry-lookahead adder

- *A balance between the two extremes:* ***ripple carry adder*** *and* ***sum-of-product adder****.*
- *Motivation:*
  - *If we didn't know the value of carry-in, what could we do?*
    - *We need to* ***look ahead*** *the carry first before we compute the final results.*
  - *When would we always generate a carry?*

    $g_i = a_i \, b_i$
  - *When would we propagate the carry?*

    $p_i = a_i + b_i$

$$c_{i+1} = a_i b_i + (a_i + b_i) c_i = g_i + p_i c_i$$

Carry-lookahead adder

- *Did we get rid of the ripple?*

  $c_1 = g_0 + p_0 c_0$

  $c_2 = g_1 + p_1 c_1$

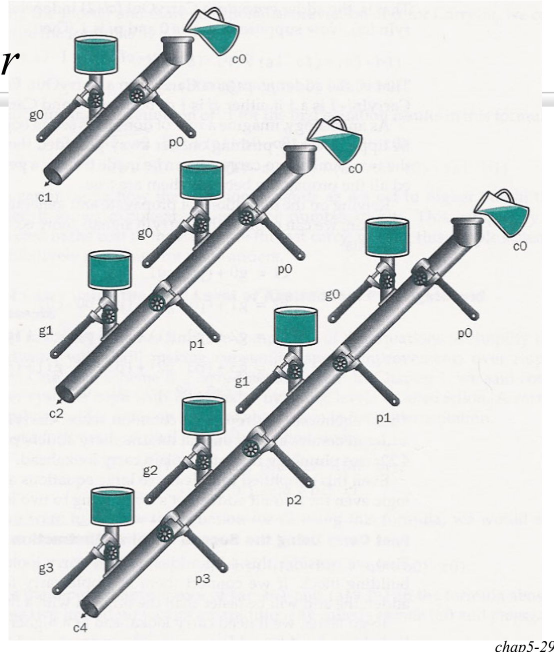
  $c_3 = g_2 + p_2 c_2$

  $c_4 = g_3 + p_3 c_3$

$$c_1 = g_0 + p_0 \cdot c_0$$
$$c_2 = g_1 + p_1 \cdot g_0 + p_1 p_0 \cdot c_0$$
$$c_3 = g_2 + p_2 \cdot g_1 + p_2 p_1 \cdot g_0 + p_2 p_1 p_0 \cdot c_0$$
$$c_4 = g_3 + p_3 \cdot g_2 + p_3 p_2 \cdot g_1 + p_3 p_2 p_1 \cdot g_0 + p_3 p_2 p_1 p_0 \cdot c_0$$

Carry-lookahead adder

- *A plumbing analogy*
  - *Notably, all p's and g's can be determined in one step.*
  - *Hence, c1, c2, c3, and c4 can be determined subsequently.*



Carry-lookahead adder

- *Can we build a 32-bit carry-lookahead adder?*
- *Answer: Perhaps, still very complicated.*

$$c_{31} = g_{30} + \ldots + p_{30}p_{29}p_{28}p_{27}p_{26}p_{25}p_{24}p_{23}p_{22}p_{21}p_{20}p_{19}p_{18} p_{17}p_{16}\, p_{15}p_{14}p_{13}p_{12}p_{11}p_{10}p_9p_8p_7p_6p_5p_4p_3g_2 + p_{30}p_{29}p_{28} p_{27}p_{26}p_{25}p_{24}p_{23}p_{22}p_{21}p_{20}p_{19}p_{18}p_{17}p_{16}\, p_{15}p_{14}p_{13}p_{12}p_{11} p_{10}p_9p_8p_7p_6p_5p_4p_3p_2g_1 + p_{30}p_{29}p_{28}p_{27}p_{26}p_{25}p_{24}p_{23} p_{22}p_{21}p_{20}p_{19}p_{18}p_{17}p_{16}\, p_{15}p_{14}p_{13}p_{12}p_{11}p_{10}p_9p_8p_7p_6p_5p_4 p_3p_2p_1g_0 + p_{30}p_{29}p_{28}p_{27}p_{26}p_{25}p_{24}p_{23}p_{22}p_{21}p_{20}p_{19}p_{18}p_{17} p_{16}\, p_{15}p_{14}p_{13}p_{12}p_{11}p_{10}p_9p_8p_7p_6p_5p_4p_3p_2p_1p_0c_0$$

Two-level carry-lookahead adder

- *Use 4-bit carry-lookahead adders as a basis to form a 16-bit adder*

$$P_0 = p_3 \cdot p_2 \cdot p_1 \cdot p_0$$
$$P_1 = p_7 \cdot p_6 \cdot p_5 \cdot p_4$$
$$P_2 = p_{11} \cdot p_{10} \cdot p_9 \cdot p_8$$
$$P_3 = p_{15} \cdot p_{14} \cdot p_{13} \cdot p_{12}$$
$$G_0 = g_3 + p_3 g_2 + p_3 p_2 g_1 + p_3 p_2 p_1 g_0$$
$$G_1 = g_7 + p_7 g_6 + p_7 p_6 g_5 + p_7 p_6 p_5 g_4$$
$$G_2 = g_{11} + p_{11} g_{10} + p_{11} p_{10} g_9 + p_{11} p_{10} p_9 g_8$$
$$G_3 = g_{15} + p_{15} g_{14} + p_{15} p_{14} g_{13} + p_{15} p_{14} p_{13} g_{12}$$

Two-level carry-lookahead adder

$$C_1 = G_0 + P_0 c_0$$
$$C_2 = G_1 + P_1 G_0 + P_1 P_0 c_0$$
$$C_3 = G_2 + P_2 G_1 + P_2 P_1 G_0 + P_2 P_1 P_0 c_0$$
$$C_4 = G_3 + P_3 G_2 + P_3 P_2 G_1 + P_3 P_2 P_1 G_0 + P_3 P_2 P_1 P_0 c_0$$

ALU Summary

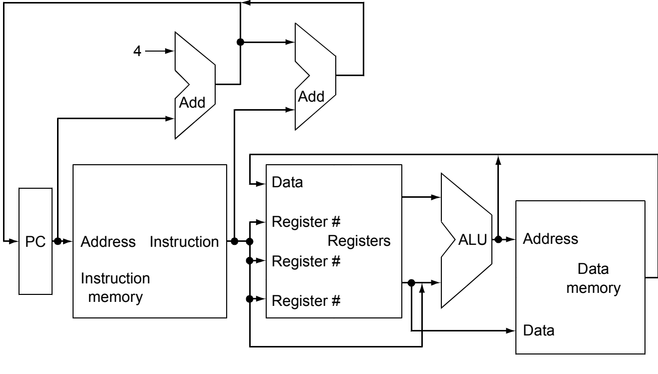
- *We can build an ALU to support MIPS addition (with faster carry-lookahead adder)*
- *Our focus is on comprehension, not performance*
- *Real processors use more sophisticated techniques for arithmetic*
- *Where performance is not critical, hardware description languages allow designers to completely automate the creation of hardware!*

```
module MIPSALU (ALUctl, A, B, ALUOut, Zero);
   input [3:0] ALUctl;
   input [31:0] A,B;
   output reg [31:0] ALUOut;
   output Zero;
   assign Zero = (ALUOut==0); //Zero is true if ALUOut is 0; goes anywhere
   always @(ALUctl, A, B) //reevaluate if these change
      case (ALUctl)
         0: ALUOut <= A & B;
         1: ALUOut <= A | B;
         2: ALUOut <= A + B;
         6: ALUOut <= A - B;
         7: ALUOut <= A < B ? 1:0;
         12: ALUOut <= ~(A | B); // result is nor
         default: ALUOut <= 0; //default to 0, should not happen;
      endcase
endmodule
```

FIGURE B.4.3 A Verilog behavioral definition of a MIPS ALU. This could be synthesized using a module library containing basic arithmetic and logical operations.

More implementation details

- *Now we are ready to see more implementation details.*
- *Abstract / Simplified View:*

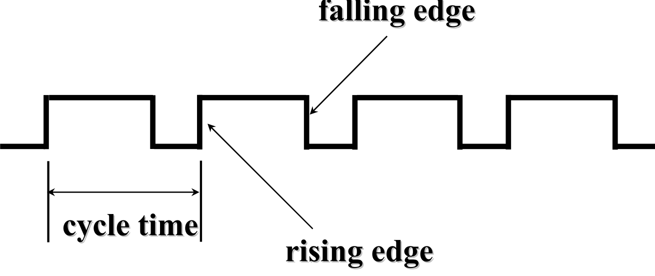


More implementation details

- *Two types of functional units:*
  - *Elements that operate on data values (combinational)*
    - *Given the same input, a combinational element always produces the same output.*
  - *Elements that contain state (sequential)*
    - *A state element contains internal storages, such as registers.*
    - *It is called "sequential" because its output depends on both the inputs and the contents of the internal state.*

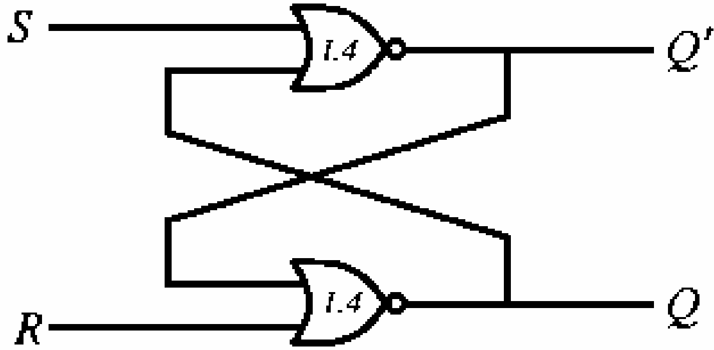
State elements

- *Unclocked versus Clocked*
- *Clocks used in synchronous logic*
  - *When should an element that contains state be updated?*



Example of an unclocked state element

- The set-reset latch
  - output depends on present inputs and also on past inputs



Example of clocked state elements

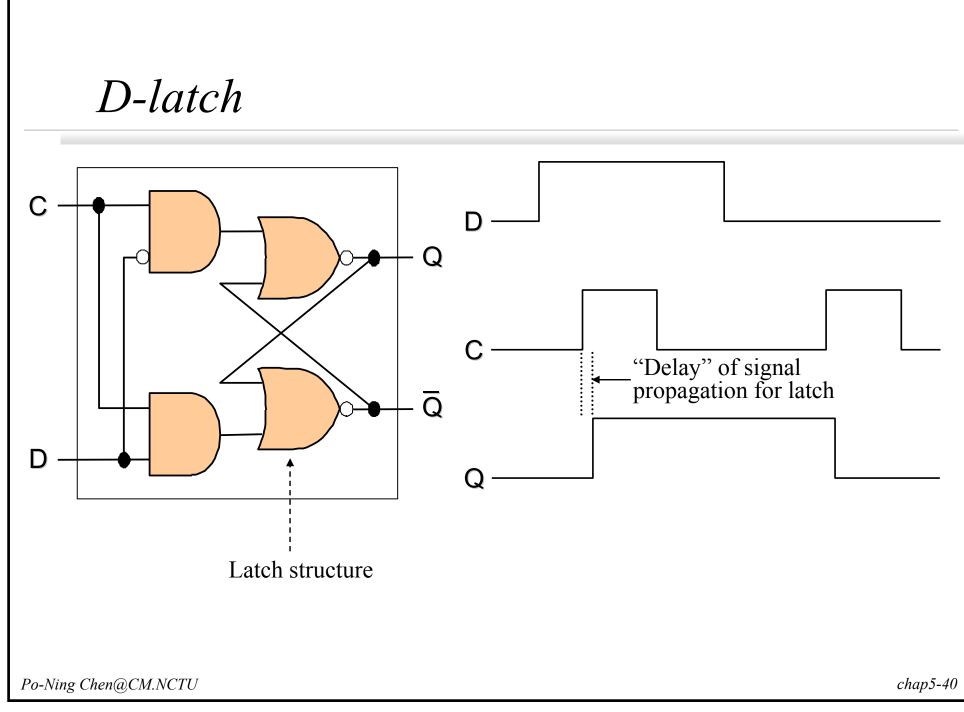
- Latches and flip-flops
  - Output is equal to the stored value inside the element (don't need to ask for permission to look at the value)
  - Change of state (value) is based on the clock

- Latches: state changes whenever the inputs change, and the clock is asserted (more like level-triggered methodology)
- Flip-flop: state changes only on a clock edge (edge-triggered methodology)

D-latch

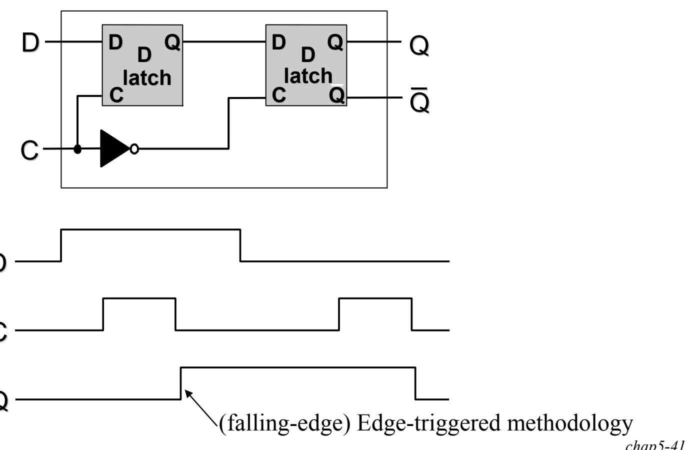
- *Two inputs:*
  - *the data value to be stored (D)*
  - *the clock signal (C) indicating when to read & store D*
- *Two outputs:*
  - *the value of the internal state (Q) and it's complement*

D-latch



D flip-flop

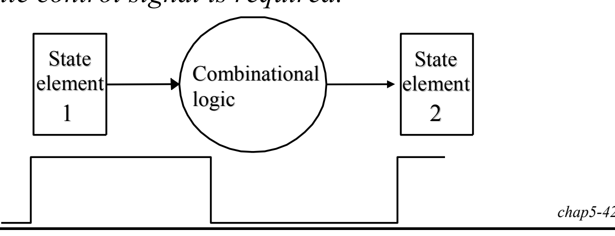
- *Output changes only on the clock edge*



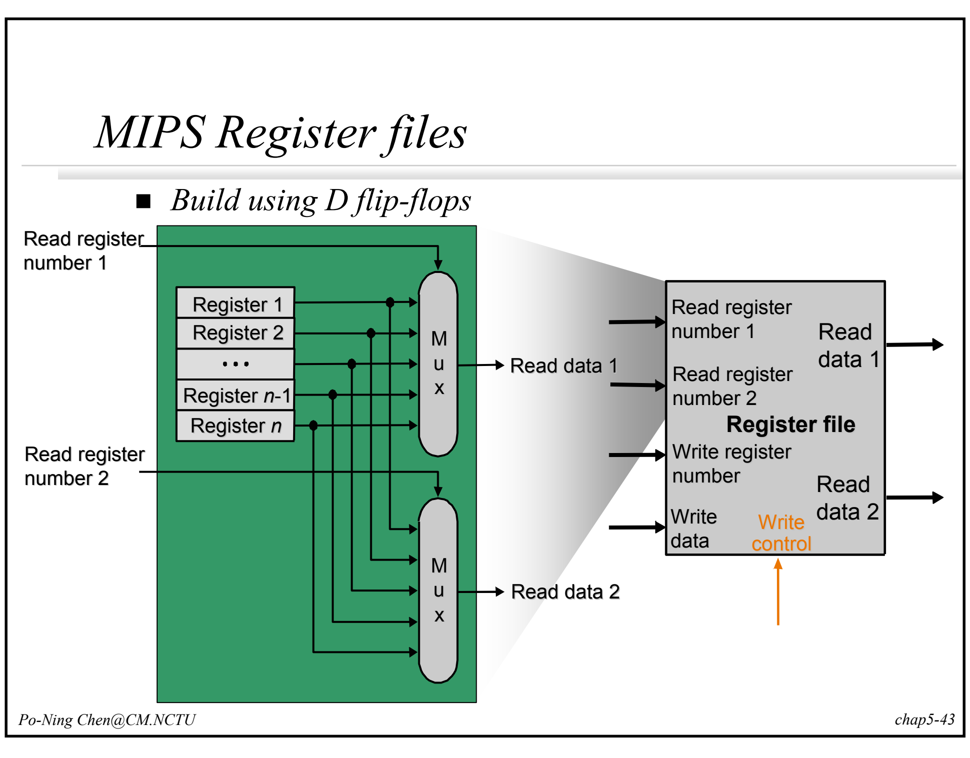
(falling-edge) Edge-triggered methodology

MIPS implementation consideration

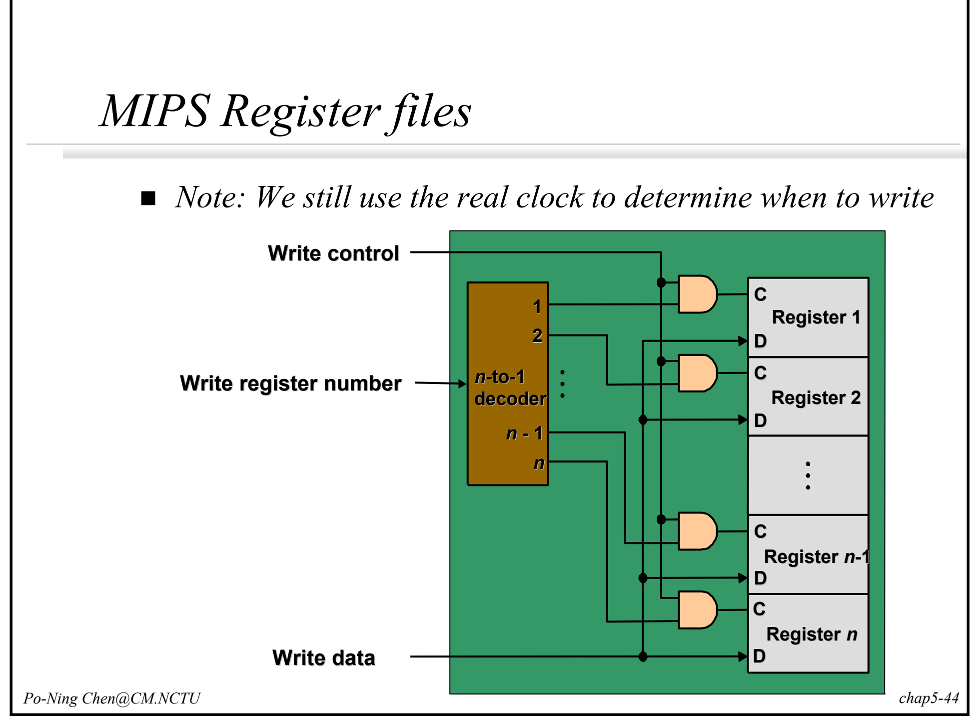
- *An edge triggered methodology*
- *Typical execution:*
  - *read contents of some state elements,*
  - *send values through some combinational logic*
  - *write results to one or more state elements*
- *The time necessary for the signals to reach state element 2 decides the length of the clock cycle.*
  - *If a state element is not updated on every clock, then an explicit write control signal is required.*



Clock cycle



MIPS Register files
Build using D flip-flops
Read register number 1
Register 1
Register 2
• • •
Register n-1
Register n
M u x
Read data 1
Read register number 2
M u x
Read data 2
Read register number 1
Read data 1
Read register number 2
Register file
Write register number
Read data 2
Write data
Write control
Po-Ning Chen@CM.NCTU
chap5-43



MIPS Register files
Note: We still use the real clock to determine when to write
Write control
Write register number
n-to-1 decoder
1
2
n - 1
n
C
Register 1
D
C
Register 2
D
C
Register n-1
D
C
Register n
D
Write data
Po-Ning Chen@CM.NCTU
chap5-44

Simple implementation for MIPS R-format instruction

- *Include the functional units we need for load-and-store instructions*

Instruction address

Instruction

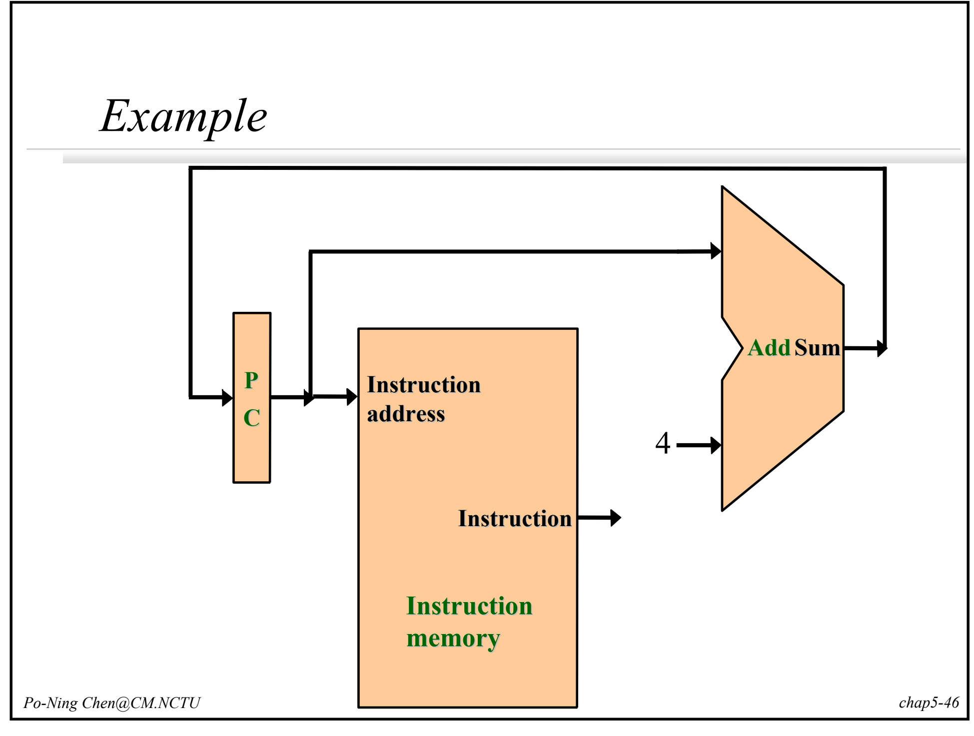
Instruction memory

P C

Add Sum

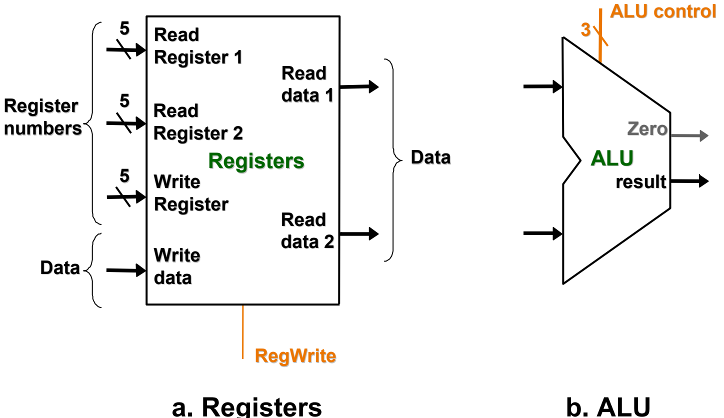
a. Instruction memory   b. Program Counter   c. Adder

Example



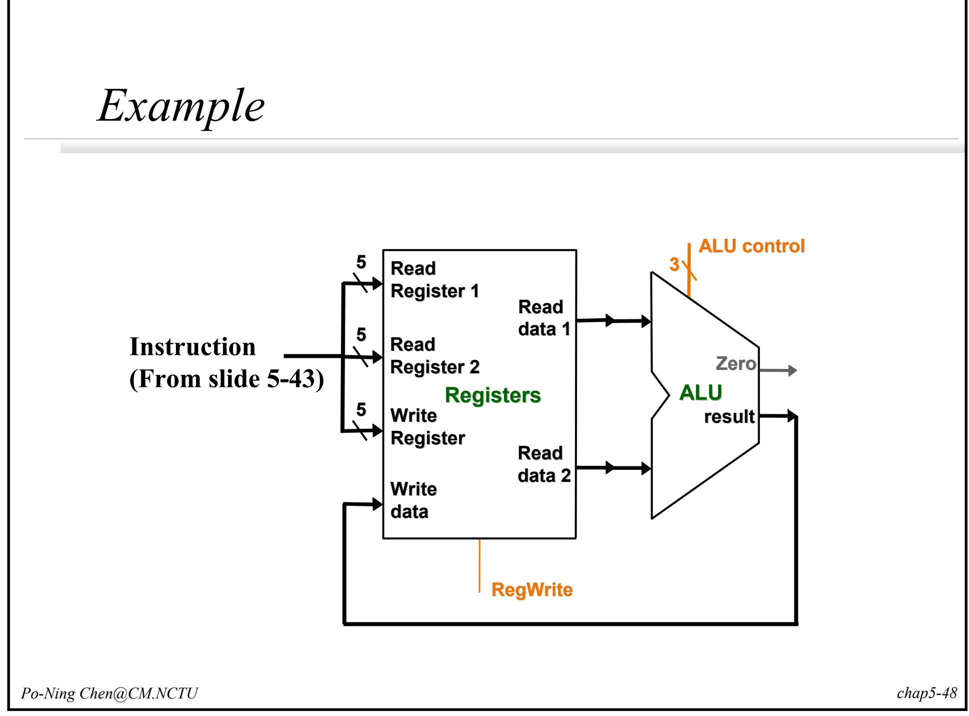
Simple implementation for MIPS R-format instruction

- Include the functional units we need for arithmetic instructions



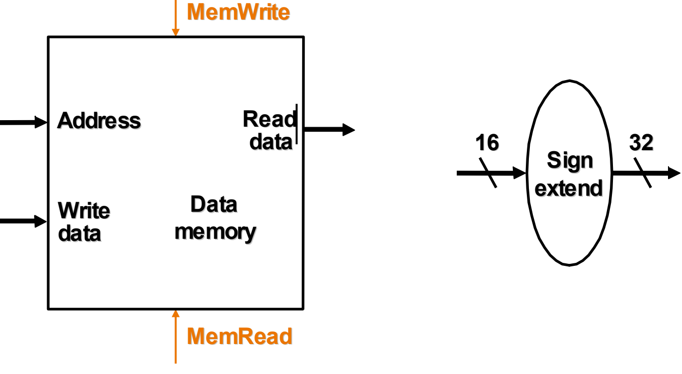
a. Registers b. ALU

Example



Simple implementation for MIPS R-format instruction

- *Include the functional units we need for memory-access (immediate-data) instructions*

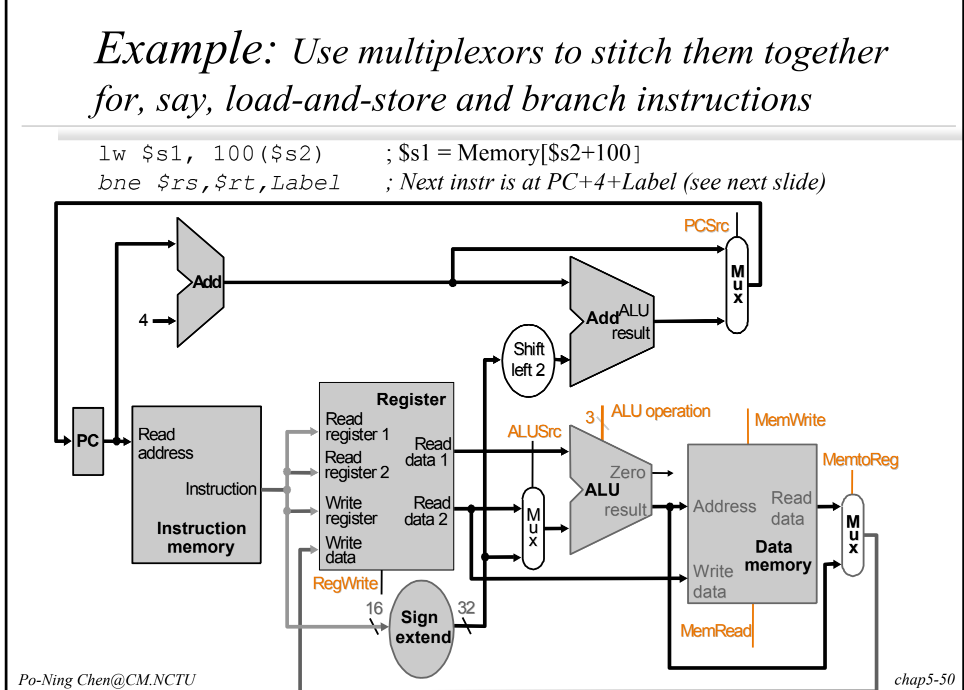


a. Data memory unit. **b. Sign extension unit**

Example: Use multiplexors to stitch them together for, say, load-and-store and branch instructions

`lw $s1, 100($s2)` ; $s1 = Memory[$s2+100]

`bne $rs,$rt,Label` ; *Next instr is at PC+4+Label (see next slide)*



Recall: PC-relative addressing

- *Example*

Exit: 80016+4+2*4 = 80028

```
Loop:  add $t1, $s3, $s3
       add $t1, $t1, $t1
       add $t1, $t1, $s6
       lw $t0, 0($t1)
       bne $t0, $s5, Exit
       add $s3, $s3, $s4
       j Loop
Exit:
```

Address						
80000	0	19	19	9	0	32
80004	0	9	9	9	0	32
80008	0	9	22	9	0	32
80012	35	9	8	0		
80016	**5**	**8**	**21**	**2**		
80020	9	19	20	19	0	32
80024	2	20000				
80028	…					

Control signals

- *Selecting the operations to perform (ALU, read/write, etc.)*
- *Controlling the flow of data (multiplexor inputs)*
- *Information comes from the 32 bits of the instruction*

Control signals

- *What should the ALU do with this instruction*

```
lw $1, 100($2)?
```

35	2	1	100
op	rs	rt	16 bit offset

Relation of the ALU design with machine language

- *Control lines:*

Instructions	*AInvert*	***BInvert***	***CarryIn***	*Operation*
AND	*0*	***0***	***0***	*00*
OR	*0*	***0***	***0***	*01*
ADD	*0*	***0***	***0***	*10*
SUB	*0*	***1***	***1***	*10*
SLT	*0*	***1***	***1***	*11*
NOR	*1*	***1***	***1***	*00*

These two bits can be combined into one, named *Bnegate*.

Control signals

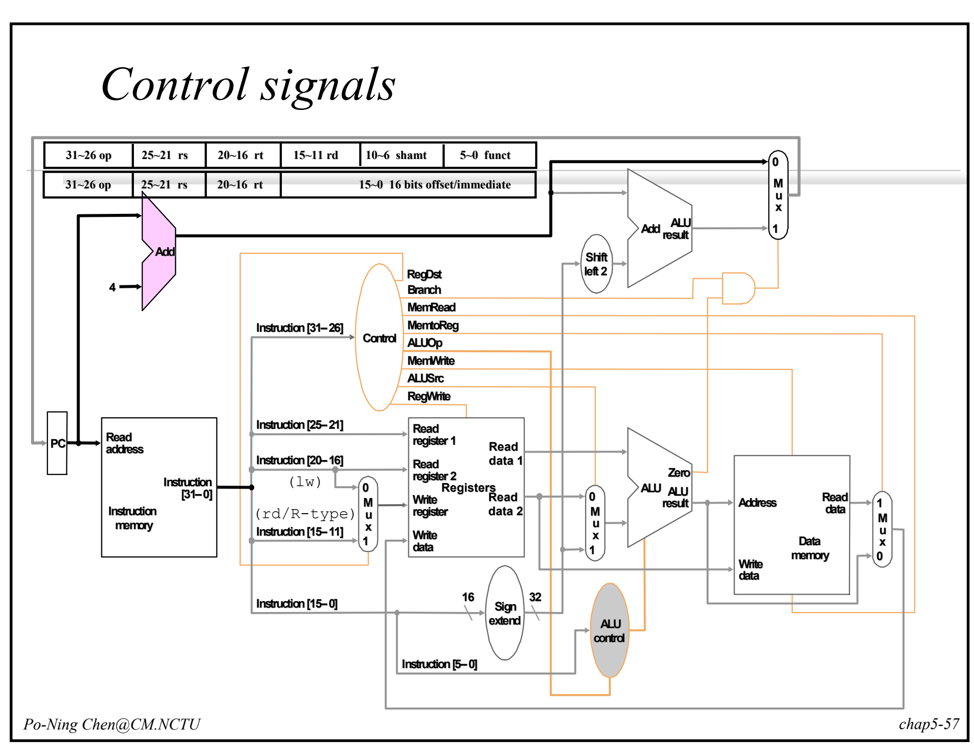
- *Must describe hardware to compute 4-bit ALU control input*
- *Also, given instruction type, type of instructions are differentiated through ALUOp:*

00 = `lw, sw`
01 = `beq,`
10 = `arithmetic`

Control signals

- *Describe it using a truth table (can turn into gates):*

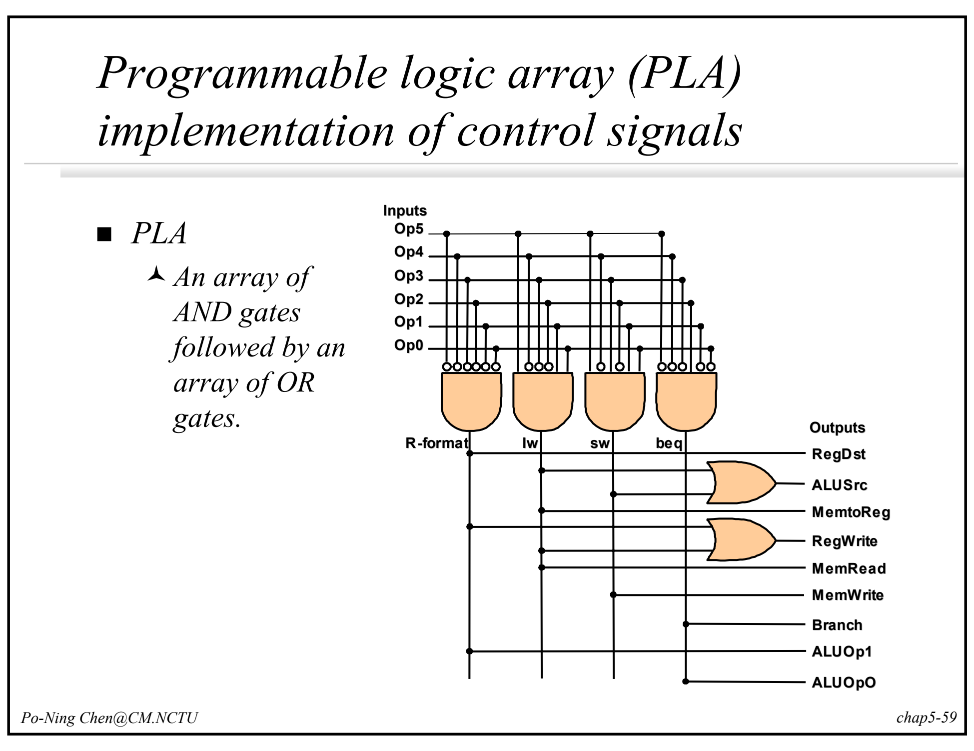
	ALUOp		Funct field						Operation
	ALUOp1	ALUOp0	F5	F4	F3	F2	F1	F0	
lw, sw	0	0	X	X	X	X	X	X	0010 (add)
beq	X	1	X	X	X	X	X	X	0110 (sub)
add	1	X	X	X	0	0	0	0	0010 (add)
sub	1	X	X	X	0	0	1	0	0110 (sub)
and	1	X	X	X	0	1	0	0	0000 (and)
or	1	X	X	X	0	1	0	1	0001 (or)
slt	1	X	X	X	1	0	1	0	0111 (slt)

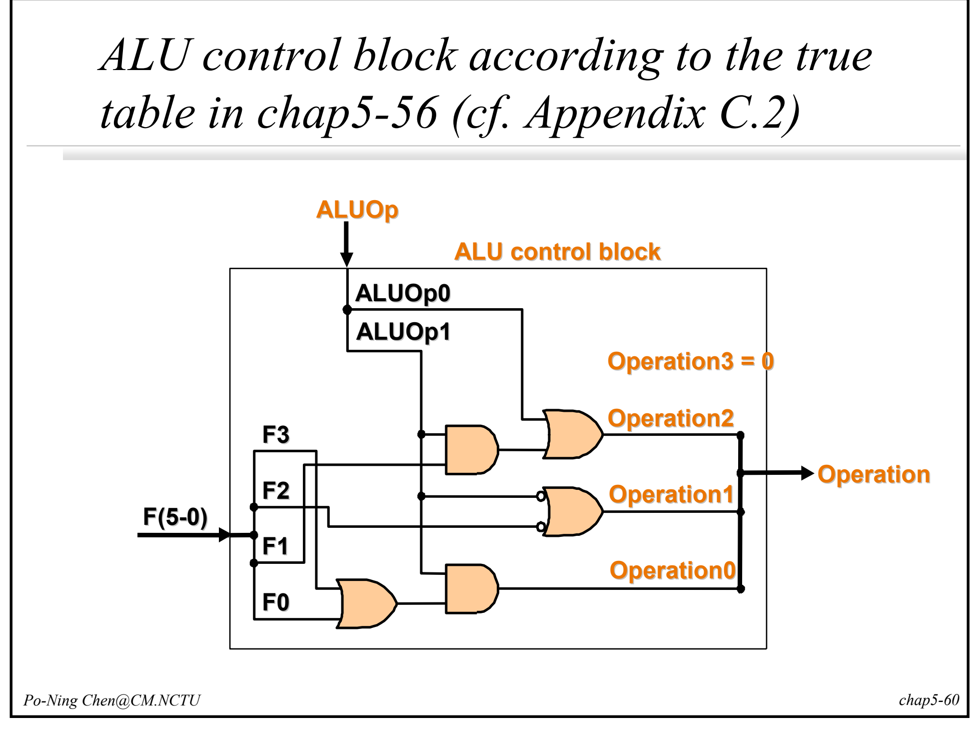


Control signals

Instruc-tion	Opcode in binary (op5~op0)	RegDst	ALUSrc	Memto-Reg	Reg Write	Mem Read	Mem Write	Branch	ALUOp1	ALUp0
R-format	000000	1	0	0	1	0	0	0	1	0
lw	100011	0	1	1	1	1	0	0	0	0
sw	101011	X	1	X	0	0	1	0	0	0
beq	000100	X	0	X	0	0	0	1	0	1

inputs *outputs*


Programmable logic array (PLA) implementation of control signals
PLA
An array of AND gates followed by an array of OR gates.
Inputs
Op5
Op4
Op3
Op2
Op1
Op0
R-format
lw
sw
beq
Outputs
RegDst
ALUSrc
MemtoReg
RegWrite
MemRead
MemWrite
Branch
ALUOp1
ALUOpO
Po-Ning Chen@CM.NCTU
chap5-59

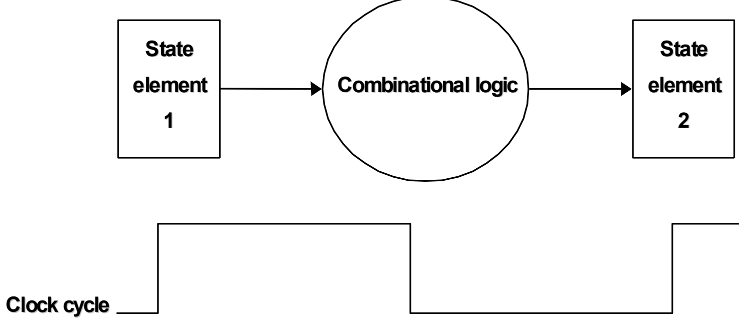

ALU control block according to the true table in chap5-56 (cf. Appendix C.2)
ALUOp
ALU control block
ALUOp0
ALUOp1
Operation3 = 0
Operation2
Operation1
Operation0
Operation
F3
F2
F(5-0)
F1
F0
Po-Ning Chen@CM.NCTU
chap5-60

Single cycle implementation

- *All of the logic is combinational*
- *We wait for everything to settle down, and the right thing to be done*
  - *ALU might not produce "right answer" right away*
  - *We use write signals along with clock to determine when to write*

Single cycle implementation

- *Cycle time determined by length of the longest path*



We are ignoring some details like setup and hold times

Single cycle implementation

- *Single Cycle Problems:*
  - *what if we had a more complicated instruction like floating point?*
  - *May exist a dominant instruction (with longest datapath delay), which is rarely used. Possibly, most of the instructions can be executed faster.*
- *One Solution:*
  - *use a "smaller" cycle time*
  - *have "**different instructions take different numbers of cycles**"*
  - *a "multicycle" datapath:*

Single cycle implementation

- *Example. Calculate cycle time assuming negligible delays except:*
  - *memory (2ns),*
  - *ALU and adders (2ns),*
  - *register file access (1ns)*

Instruction	Total	Functional units used by the instruction class				
ALU type	6 ns	Instruction fetch	Register access	ALU	Register access	
Load word	8 ns	Instruction fetch	Register access	ALU	Memory access	Register access
Store word	7 ns	Instruction fetch	Register access	ALU	Memory access	
Branch	5 ns	Instruction fetch	Register access	ALU		
Jump	2 ns	Instruction fetch				

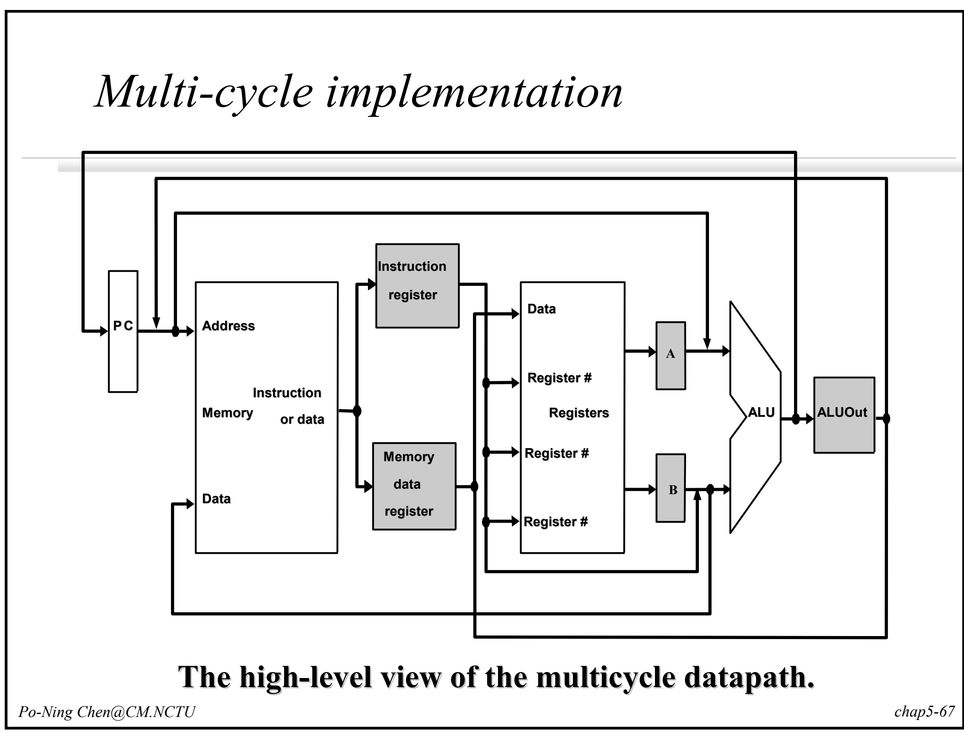
Single cycle implementation

- *Example. Calculate cycle time assuming negligible delays except:*
  - *memory (200ns),*
  - *ALU and adders (100ns),*
  - *register file access (50ns)*

Instruction	Total	Functional units used by the instruction class				
ALU type	400 ns	Instruction fetch	Register access	ALU	Register access	
Load word	600 ns	Instruction fetch	Register access	ALU	Memory access	Register access
Store word	550 ns	Instruction fetch	Register access	ALU	Memory access	
Branch	350 ns	Instruction fetch	Register access	ALU		
Jump	200 ns	Instruction fetch				

Multicycle approach

- *We will be reusing functional units*
  - *Allow a functional unit to be used more than once per instruction, as long as it is used on different clock cycles.*
  - *E.g., the same ALU can be used to compute address and to increment PC at different cycles*
  - *Require additional temporary registers to hold data between clock cycles of the same instruction, such as IR, MDR, A, B and ALUOut in the next slide.*
- *Our control signals will not be determined solely by instruction, but also by the cycle sequence number*
- *We'll use a* ***finite state machine*** *for control*



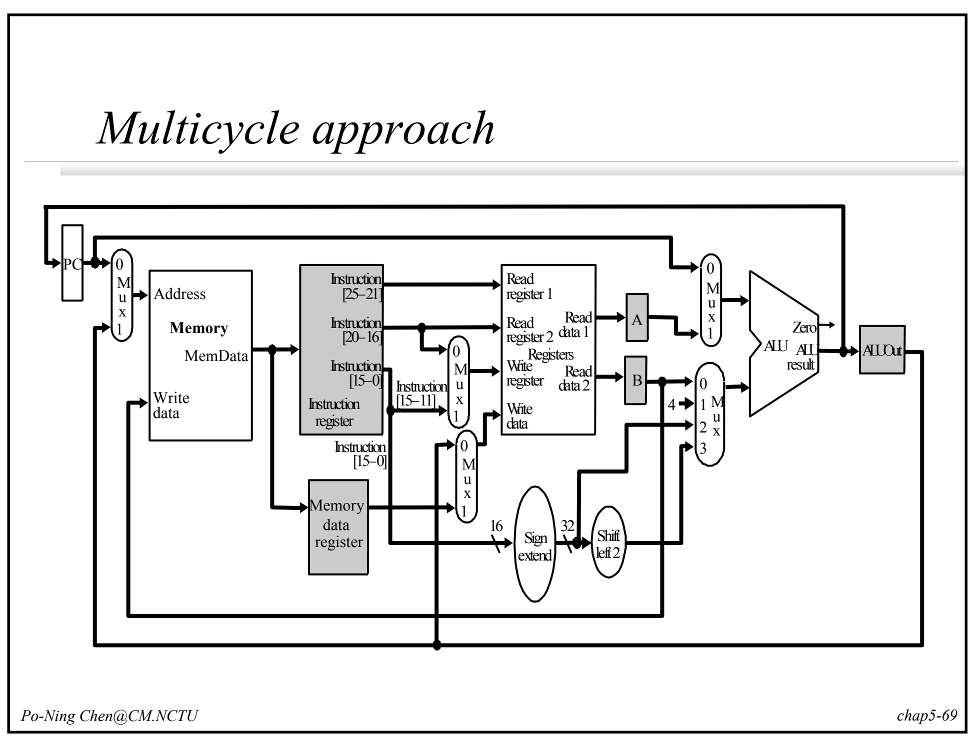
The high-level view of the multicycle datapath.

Multicycle approach

- *Break up the instructions into steps, each step takes a cycle*
  - *balance the amount of work to be done*
  - *restrict each cycle to use only one major functional unit*

- *At the end of a cycle*
  - *store values for use in later cycles (easiest thing to do)*
  - *Hence, introduce additional temporary "internal" registers*

Multicycle approach



Five execution steps

- *Exemplified steps*
  - *Instruction fetch*
  - *Instruction decode and register fetch*
  - *Execution, memory address computation, or branch completion*
  - *Memory access or R-type instruction completion*
  - *Memory read completion (Write-back) step*

INSTRUCTIONS TAKE FROM 3 - 5 CYCLES!

Step 1: Instruction fetch

- *Use PC to get instruction and put it in the Instruction Register.*
- *Increment the PC by 4 and put the result back in the PC.*
- *Can be described succinctly using RTL "Register-Transfer Language"*

```
IR <= Memory[PC];
PC <= PC + 4;
```

ALU has been used once here!

Step 2: Instruction decode and register fetch

- *Read registers* `rs` *and* `rt` ***in case*** *we need them*
- *Compute the branch address* ***in case*** *the instruction is a branch*
- *RTL:*

```
A <= Reg[IR[25:21]];
B <= Reg[IR[20:16]];
ALUOut <= PC + (sign-extend(IR[15:0]) << 2);
```

Step 3: Instruction dependent

- *ALU is performing one of three functions, based on instruction type*
  - *Memory Reference:*
    `ALUOut <= A + sign-extend(IR[15:0]);` *address calculation*
  - *Arithmetic-logical instruction (R-type):*
    `ALUOut <= A op B;`
  - *Branch:*
    `if (A==B) PC <= ALUOut;`
  - *Jump:*
    `PC <= {PC[31:28],(IR[25:0]],2’b00)};`

  # `{x,y}` is the Verilog notation for concatenation of bit fields of `x` and `y`

Step 4: R-type or memory-access

- *Loads and stores access memory*
```
MDR <= Memory[ALUOut];
```
or
```
Memory[ALUOut] <= B;
```

- *R-type instructions finish*
```
Reg[IR[15:11]] <= ALUOut;
```

The write actually takes place at the end of the cycle on the edge

Step 5: Write back step

- `Reg[IR[20:16]]= MDR;`

Summary

Step name	Action for R-type instructions	Action for memory-reference instructions	Action for branches	Action for jumps
Instruction fetch	IR <= Memory[PC] PC <= PC + 4			
Instruction decode/register fetch	A <= Reg [IR[25:21]] B <= Reg [IR[20:16]] ALUOut <= PC + (sign-extend (IR[15:0]) << 2)			
Execution, address computation, branch/ jump completion	ALUOut <= A op B	ALUOut <= A + sign-extend (IR[15:0])	if (A ==B) then PC <= ALUOut	PC <= {PC [31-28], (IR[25:0],2'b00)}
Memory access or R-type completion	Reg [IR[15:11]] <= ALUOut	Load: MDR <= Memory[ALUOut] or Store: Memory [ALUOut] <= B		
Memory read completion		Load: Reg[IR[20-16]] <= MDR		

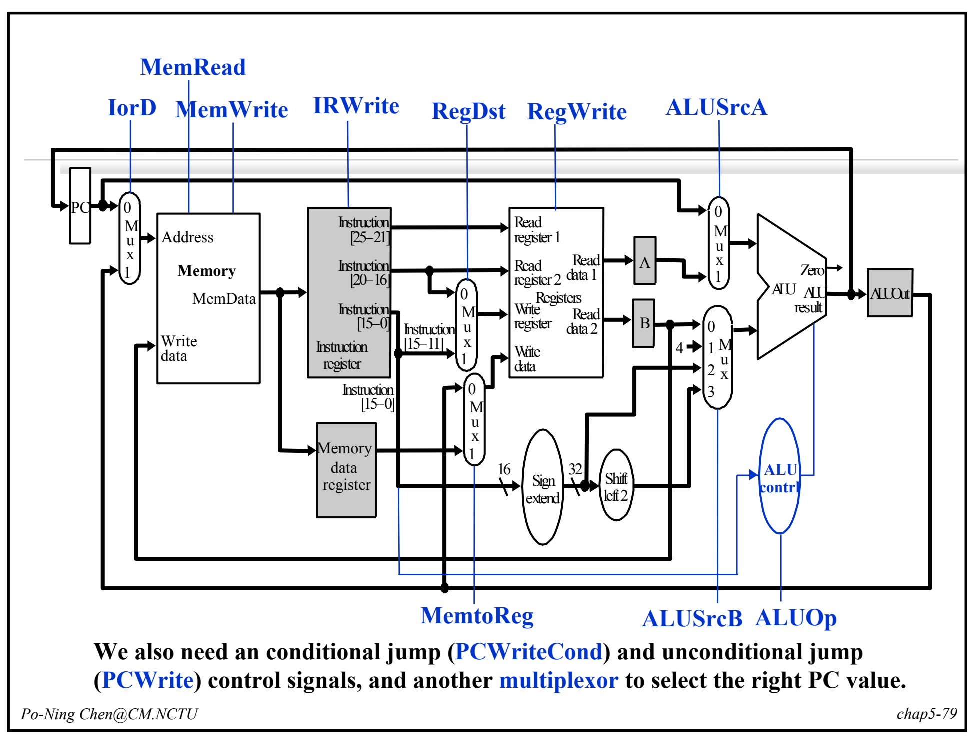
Example

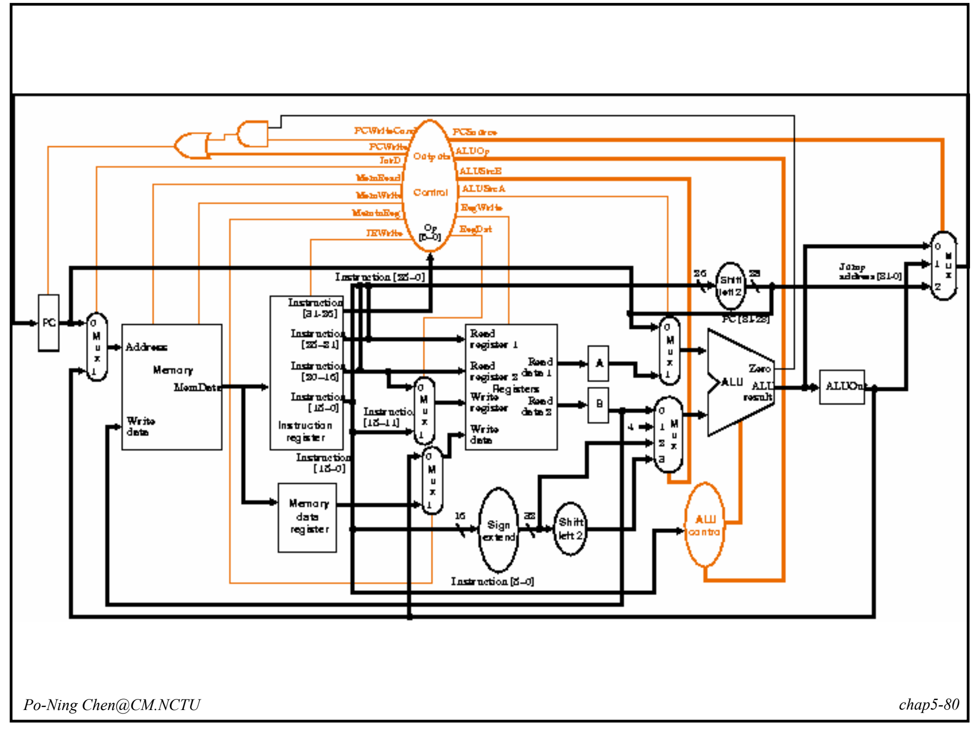
- *How many cycles will it take to execute this code?*

```
        lw $t2, 0($t3)        5
        lw $t3, 4($t3)        5
        beq $t2, $t3, Label   3
        add $t5, $t2, $t3     4
        sw $t5, 8($t3)        4
Label:  ...
```


Implementing the control signals

- *Value of control signals is dependent upon:*
  - *what instruction is being executed*
  - *which step is being performed*
- *Use the information we've accumulated to specify a finite state machine*
  - *specify the finite state machine graphically, or*
  - *use microprogramming*
- *Implementation can be derived from specification*
- *Usual implementation approach*
  - *Finite state machine*
  - *Microprogramining*


MemRead
IorD
MemWrite
IRWrite
RegDst
RegWrite
ALUSrcA
PC
Address
Memory
MemData
Write data
Instruction [25–21]
Instruction [20–16]
Instruction [15–0]
Instruction register
Instruction [15–11]
Instruction [15–0]
Memory data register
Read register 1
Read register 2
Registers
Write register
Write data
Read data 1
Read data 2
A
B
4
Zero
ALU
ALU result
ALUOut
16
Sign extend
32
Shift left 2
ALU control
MemtoReg
ALUSrcB
ALUOp
We also need an conditional jump (PCWriteCond) and unconditional jump (PCWrite) control signals, and another multiplexor to select the right PC value.
Po-Ning Chen@CM.NCTU
chap5-79


PCWriteCond
PCWrite
IorD
MemRead
MemWrite
MemtoReg
IRWrite
PCSource
ALUOp
ALUSrcB
ALUSrcA
RegWrite
RegDst
Outputs
Control
Op [5–0]
Instruction [25–0]
Jump address [31-0]
PC [31-28]
Shift left 2
Address
Memory
MemData
Write data
Instruction register
Memory data register
Read register 1
Read register 2
Registers
Write register
Write data
Read data 1
Read data 2
ALU
Zero
ALU result
ALUOut
Sign extend
Shift left 2
ALU control
Instruction [5–0]
Po-Ning Chen@CM.NCTU
chap5-80

Actions of the 1-bit control signals

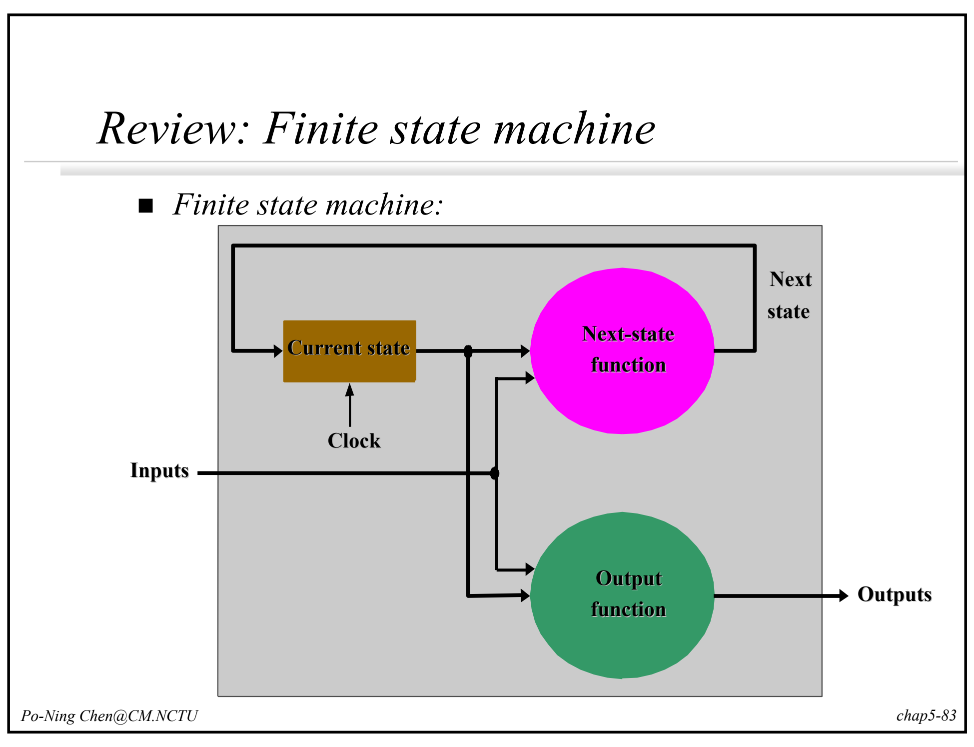
Signal name	Effect when deasserted	Effect when asserted
RegDst	The register file destination number for the Write register comes from the rt field.	The register file destination number for the Write register comes from the rd field.
RegWrite	None	The general-purpose register selected by the Write register number is written with the value of the Write data input.
ALUSrcA	The first ALU operand is the PC.	The first ALU operand comes from the A register.
MemRead	None	Content of memory at the location specified by the Address input is put on Memory data output.
MemWrite	None	Memory contents at the location specified by the Address input is replaced by value on Write data input.
MemtoReg	The value fed to the register file Write data input comes from ALUOut.	The value fed to the register file Write data input comes from the MDR.
IorD	The PC is used to supply the address to the memory unit.	ALUOut is used to supply the address to the memory unit.
IRWrite	None	The output of the memory is written into the IR.
PCWrite	None	The PC is written; the source is controlled by PCSource.
PCWriteCond	None	The PC is written if the Zero output from the ALU is also active.

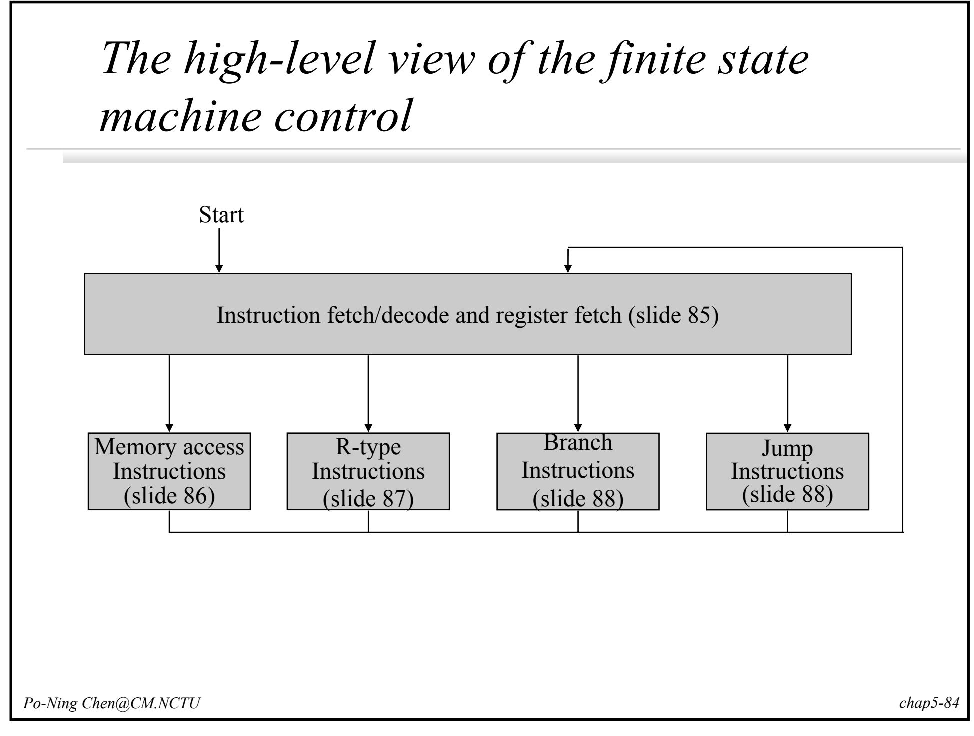
Actions of the 2-bit control signals

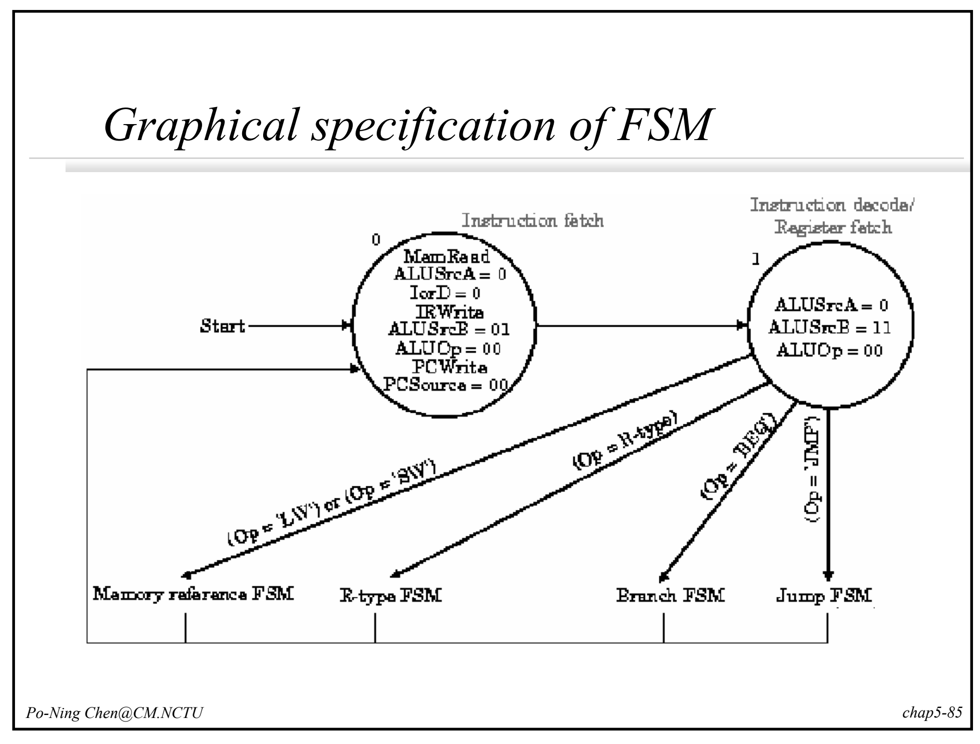
Signal name	Value	Effect
ALUOp	00	The ALU performs an add operation.
	01	The ALU performs a subtract operation.
	10	The funct field of the instruction determines the ALU operation.
ALUSrcB	00	The second input to the ALU comes from the B register.
	01	The second input to the ALU is the constant 4.
	10	The second input to the ALU is the sign-extended, lower 16 bits of the IR.
	11	The second input to the ALU is the sign-extended, lower 16 bits of the IR shifted left 2 bits.
PCSource	00	Output of the ALU (PC + 4) is sent to the PC for writing.
	01	The contents of ALUOut (the branch target address) are sent to the PC for writing.
	10	The jump target address (IR[25–0] shifted left 2 bits and concatenated with PC + 4[31–28]) is sent to the PC for writing.

Review: Finite state machine

- *Finite state machines:*
  - *A set of states and*
  - *next state function (determined by current state and the input)*
  - *output function (determined by current state and possibly input)*
    - *We'll use a Moore machine (output based only on current state)*


Review: Finite state machine
Finite state machine:
Next
state
Current state
Next-state
function
Clock
Inputs
Output
function
Outputs
Po-Ning Chen@CM.NCTU
chap5-83


The high-level view of the finite state machine control
Start
Instruction fetch/decode and register fetch (slide 85)
Memory access
Instructions
(slide 86)
R-type
Instructions
(slide 87)
Branch
Instructions
(slide 88)
Jump
Instructions
(slide 88)
Po-Ning Chen@CM.NCTU
chap5-84


Graphical specification of FSM
Instruction fetch
Instruction decode/
Register fetch
0
MemRead
ALUSrcA = 0
IorD = 0
IRWrite
ALUSrcB = 01
ALUOp = 00
PCWrite
PCSource = 00
1
ALUSrcA = 0
ALUSrcB = 11
ALUOp = 00
Start
(Op = 'LW') or (Op = 'SW')
(Op = R-type)
(Op = 'BEQ')
(Op = 'JMP')
Memory reference FSM
R-type FSM
Branch FSM
Jump FSM
Po-Ning Chen@CM.NCTU
chap5-85

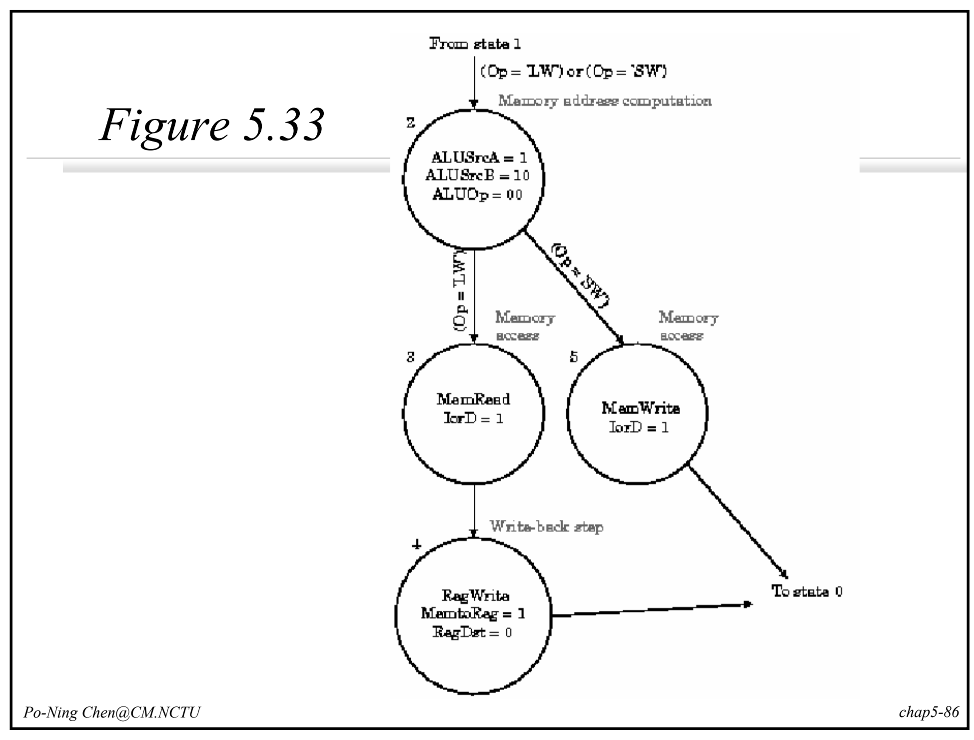


Figure 5.33
From state 1
(Op = 'LW') or (Op = 'SW')
Memory address computation
2
ALUSrcA = 1
ALUSrcB = 10
ALUOp = 00
(Op = 'LW')
(Op = 'SW')
Memory access
Memory access
3
MemRead
IorD = 1
5
MemWrite
IorD = 1
Write-back step
4
RegWrite
MemtoReg = 1
RegDst = 0
To state 0
Po-Ning Chen@CM.NCTU
chap5-86

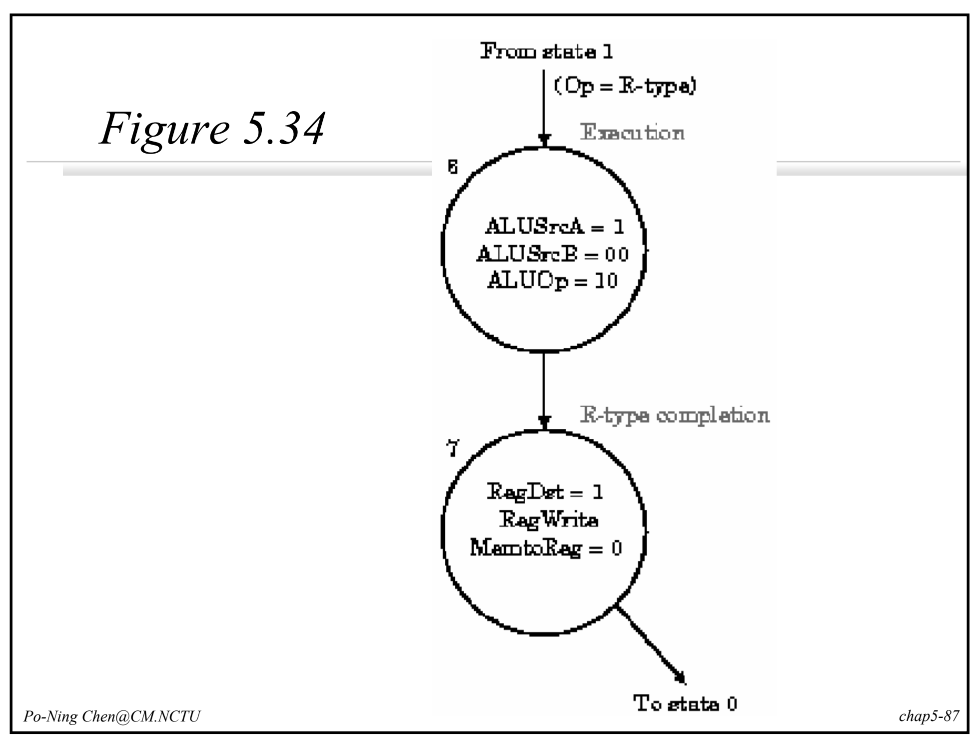
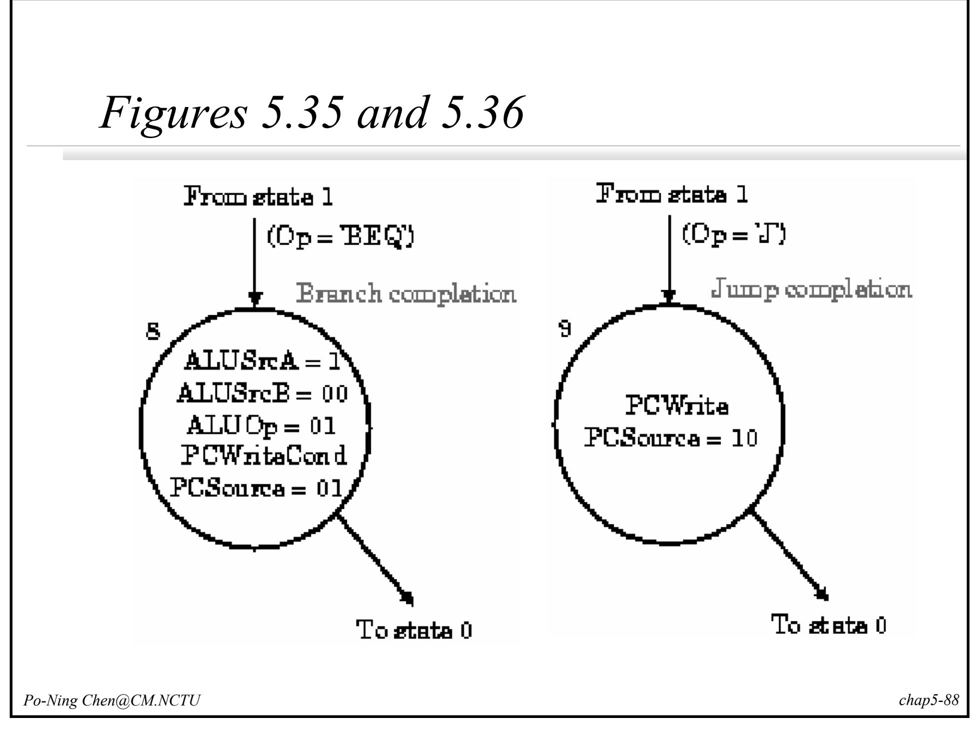
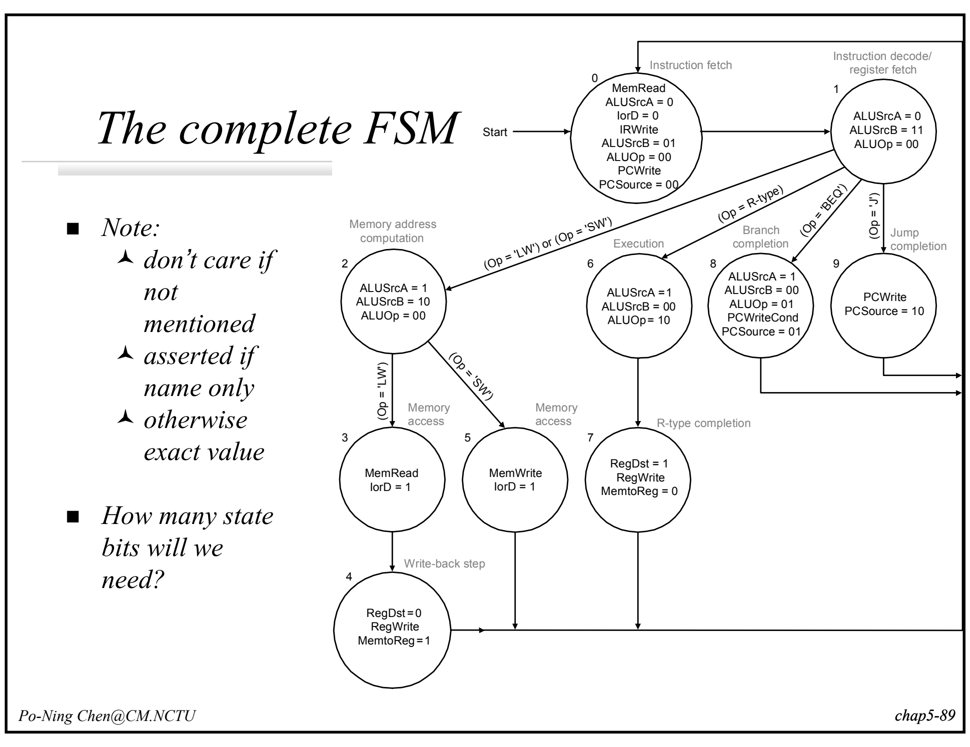


Figure 5.34
From state 1
(Op = R-type)
Execution
6
ALUSrcA = 1
ALUSrcB = 00
ALUOp = 10
R-type completion
7
RegDst = 1
RegWrite
MemtoReg = 0
To state 0
Po-Ning Chen@CM.NCTU
chap5-87


Figures 5.35 and 5.36
From state 1
(Op = 'BEQ')
Branch completion
8
ALUSrcA = 1
ALUSrcB = 00
ALUOp = 01
PCWriteCond
PCSource = 01
To state 0
From state 1
(Op = 'J')
Jump completion
9
PCWrite
PCSource = 10
To state 0
Po-Ning Chen@CM.NCTU
chap5-88



- *The logic equations for the control unit shown in a shorthand form*

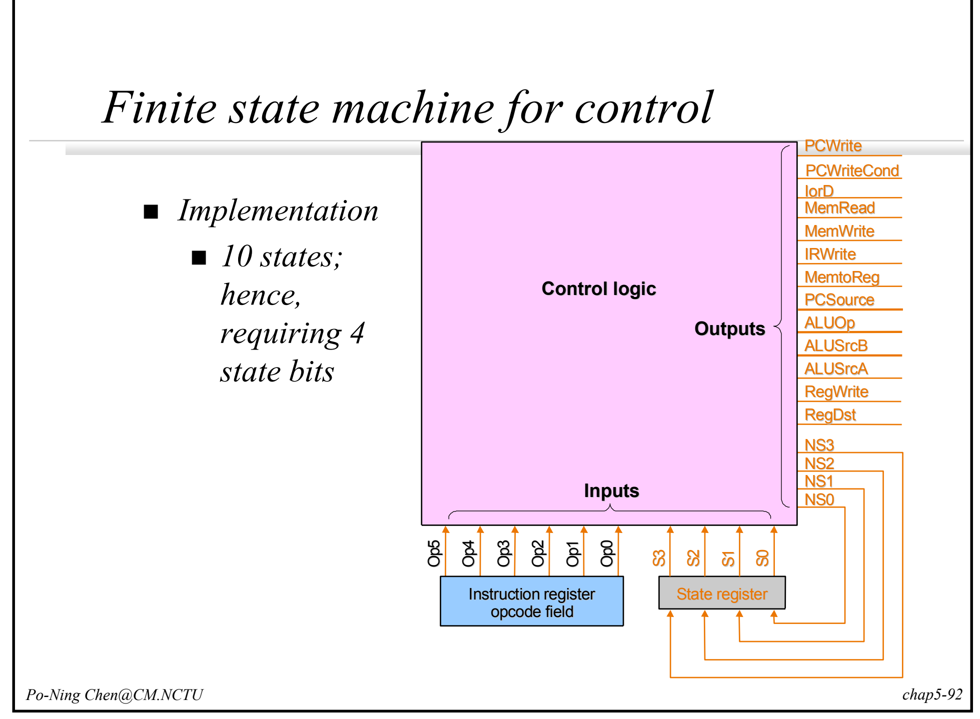
Output	Current states	Op
PCWrite	State0 + State9	
PCWriteCond	State8	
IorD	Stae3+ + State5	
MemRead	State0 + State3	
MemWrite	State5	
IRWrite	State0	
MemtoReg	State4	
PCSource1	State9	
PCSource0	State8	
ALUOp1	State6	
ALUOP0	State8	
SLUSrcB1	State1 + State2	
ALUSrcB0	State0 + State1	
ALUSrcA	State2 + State6 + State8	
RegWrite	State4 + State7	
RegDst	State7	

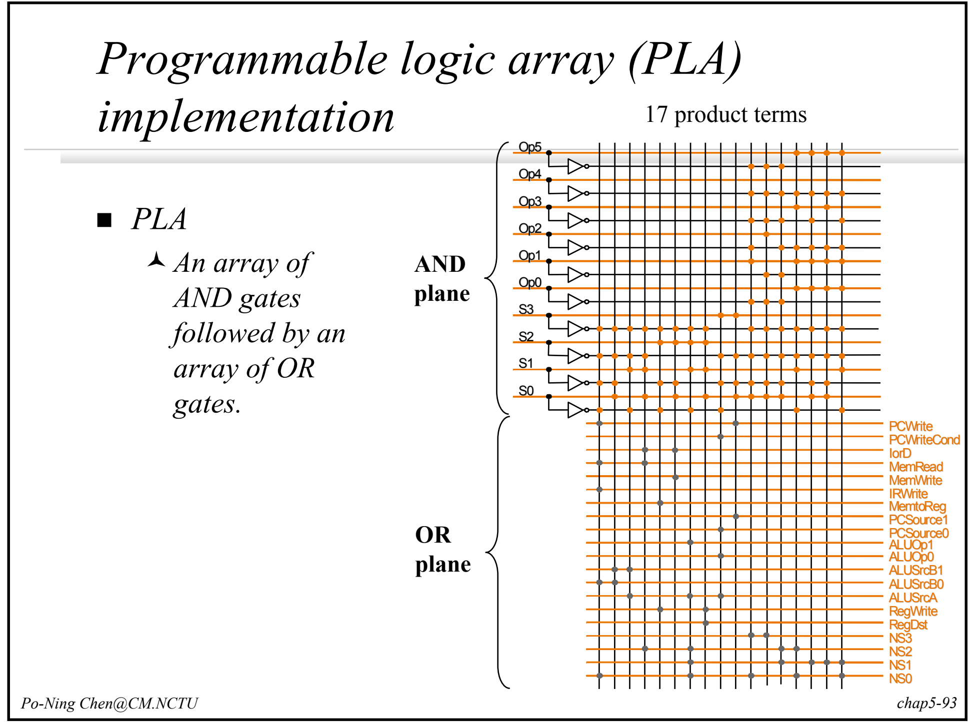
- *The logic equations for the control unit shown in a shorthand form (cont.)*

Output	Current states	Op
NextState0	State4 + State5 + State7 + State8+ State9	
NextState1	State0	
NextStste2	State1	Op = 'lw' or 'sw'
NextState3	State2	Op = 'lw'
NextState4	State3	
NextState5	State2	Op = 'sw'
NextState6	State1	Op = 'R-type'
NextState7	State6	
NextState8	State1	Op = 'beq'
NextState9	State1	Op = 'j'

Finite state machine for control

- *Implementation*
  - *10 states; hence, requiring 4 state bits*

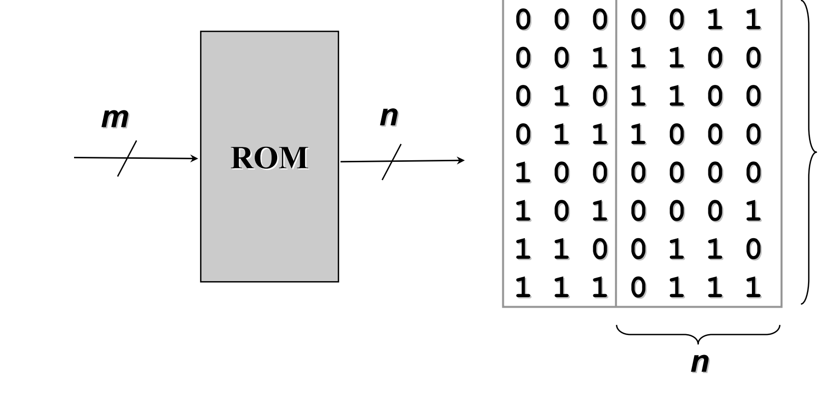




ROM implementation

- *ROM = "Read Only Memory"*
  - *values of memory locations are fixed ahead of time*
- *A ROM can be used to implement a truth table*
  - *if the address is* m*-bits, we can address* 2^m *entries in the ROM.*
  - *our outputs are the bits of data that the address points to.*

ROM implementation



ROM implementation

- *How many inputs are there?*

  6 bits for opcode, 4 bits for state = 10 address lines (i.e., 2^{10} = 1024 different addresses)

- *How many outputs are there?*

  16 datapath-control outputs, 4 state bits = 20 outputs

ROM implementation

- *ROM is $2^{10} \times 20$ = 20K bits (and a rather unusual size)*
- *Rather wasteful, since for lots of the entries, the outputs are the same*
  — *i.e., opcode is often ignored*

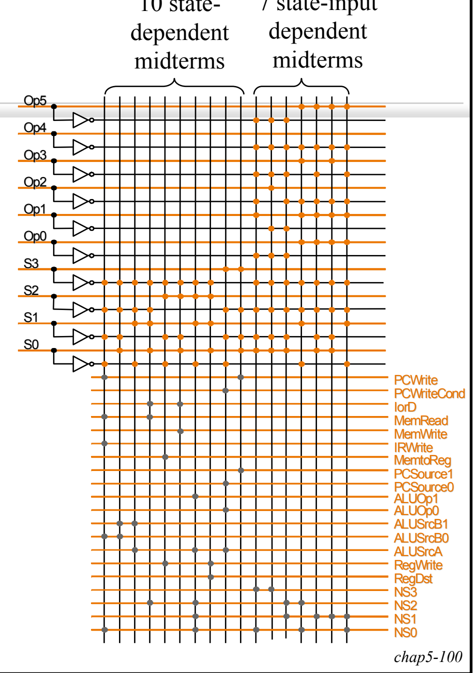
ROM implementation

- *Advantage of a Moore machine*
  - *The outputs only depend on the states*
- *So we can break up a Moore machine into two parts*
  - *4 state bits tell you the 16 outputs, $2^4 \times 16$ bits of ROM*
  - *10 bits tell you the 4 next state bits, $2^{10} \times 4$ bits of ROM*
  - *Total: 4.3K bits of ROM*

PLA implementation

- *PLA is much smaller*
  - *can share product terms*
  - *only need entries that produce an active output*
  - *can take into account don't cares*

PLA implementation



- *Size is (#inputs × #product-terms) + (#outputs × #product-terms)*

For this example =

(10×17)+(20×17) = 510 *PLA cells*

<< 4.3K

- *PLA cells usually about the size of a ROM cell (slightly bigger)*

PLA implementation

- *Take advantage of a Moore machine (separating the design into two pieces)*

4 state bits, 10 product-terms ⇒ 16 outputs

10 bits , 7 product-terms ⇒ 4 next state bits

$(4\times10)+(10\times16) = 200$ *PLA cells*

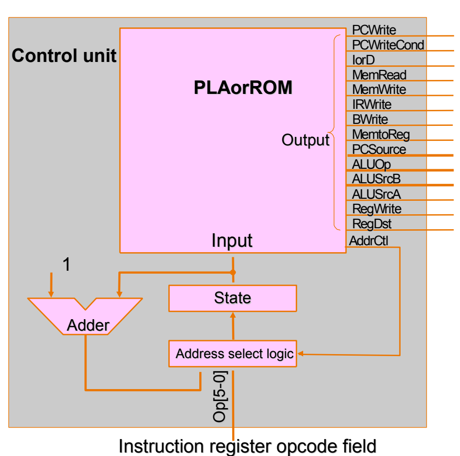
$(10\times7)+(7\times4) = 98$ *PLA cells*

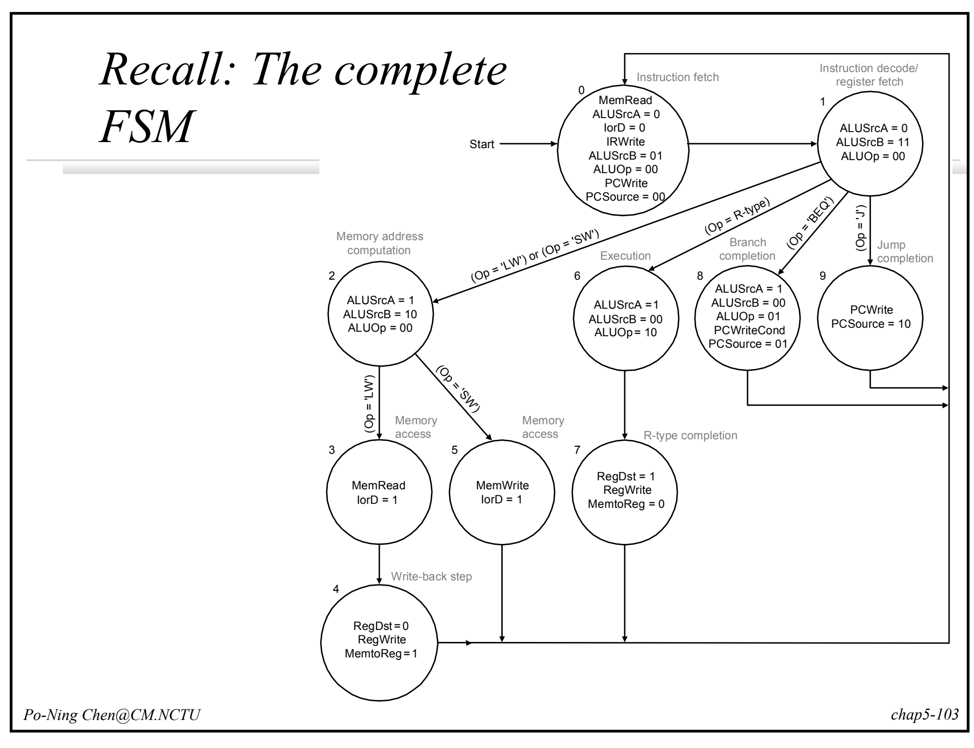
---

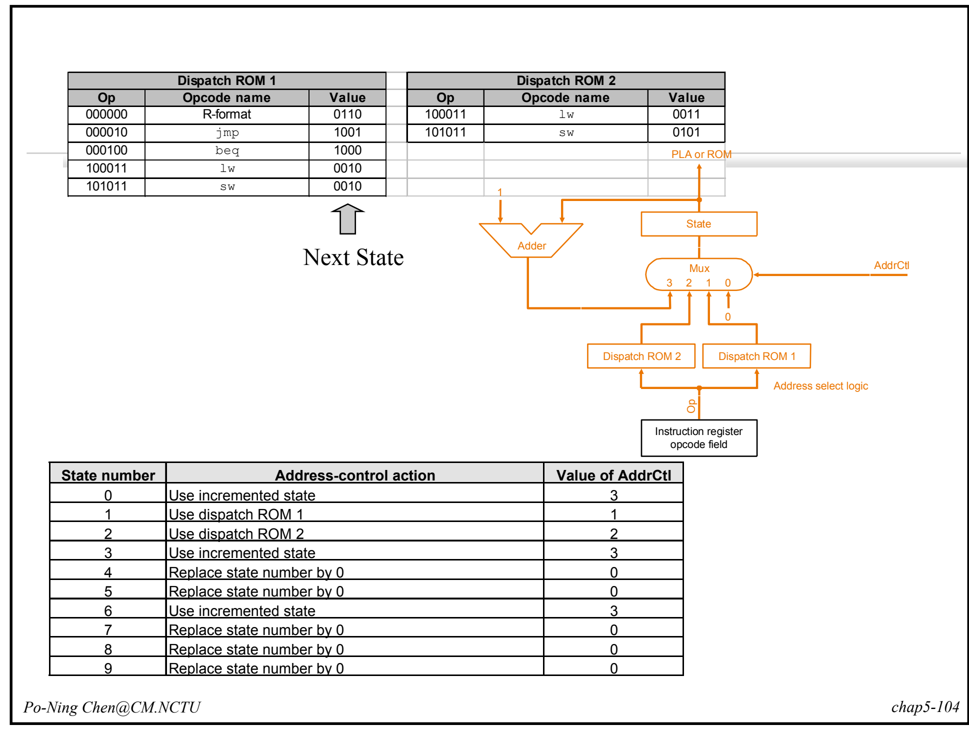
298 *PLA cells*

Another implementation style

- *Observation: the "next state" is very often = current state + 1*
  - *So we may alternative use an explicit control signal **AddrCtl** to choose from the next state and branch state to simplify the design.*







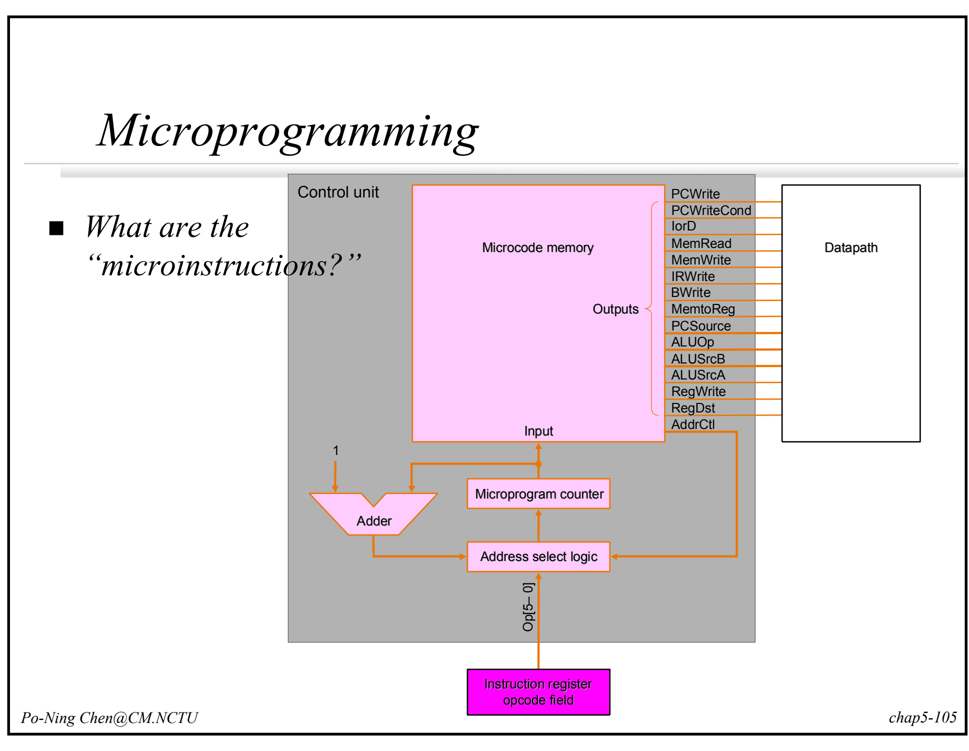
Dispatch ROM 1		
Op	Opcode name	Value
000000	R-format	0110
000010	jmp	1001
000100	beq	1000
100011	lw	0010
101011	sw	0010

Dispatch ROM 2		
Op	Opcode name	Value
100011	lw	0011
101011	sw	0101

State number	Address-control action	Value of AddrCtl
0	Use incremented state	3
1	Use dispatch ROM 1	1
2	Use dispatch ROM 2	2
3	Use incremented state	3
4	Replace state number by 0	0
5	Replace state number by 0	0
6	Use incremented state	3
7	Replace state number by 0	0
8	Replace state number by 0	0
9	Replace state number by 0	0

Microprogramming

- *What are the "microinstructions?"*



Microprogramming

Field name	Function of field
ALU control	Specify the operation being done by the ALU during this clock; the result is always written in ALUOut.
SRC1	Specify the source for the first ALU operand.
SRC2	Specify the source for the second ALU operand.
Register control	Specify read or write for the register file, and the source of the value for a write.
Memory	Specify read or write, and the source for the memory. For a read, specify the destination register.
PCWrite control	Specify the writing of the PC.
Sequencing	Specify how to choose the next microinstruction to be executed.

FIGURE 5.7.1 Each microinstruction contains these seven fields.

Microinstruction format

Field name	Value	Signals active	Comment
	Add	ALUOp = 00	Cause the ALU to add.
ALU control	Subt	ALUOp = 01	Cause the ALU to subtract; this implements the compare for branches.
	Func code	ALUOp = 10	Use the instruction's function code to determine ALU control.
SRC1	PC	ALUSrcA = 0	Use the PC as the first ALU input.
	A	ALUSrcA = 1	Register A is the first ALU input.
	B	ALUSrcB = 00	Register B is the second ALU input.
SRC2	4	ALUSrcB = 01	Use 4 as the second ALU input.
	Extend	ALUSrcB = 10	Use output of the sign extension unit as the second ALU input.
	Extshft	ALUSrcB = 11	Use the output of the shift-by-two unit as the second ALU input.
	Read		Read two registers using the rs and rt fields of the IR as the register numbers and putting the data into registers A and B.
Register control	Write ALU	RegWrite, RegDst = 1, MemtoReg = 0	Write a register using the rd field of the IR as the register number and the contents of the ALUOut as the data.
	Write MDR	RegWrite, RegDst = 0, MemtoReg = 1	Write a register using the rt field of the IR as the register number and the contents of the MDR as the data.
	Read PC	MemRead, lorD = 0	Read memory using the PC as address; write result into IR (and the MDR).
Memory	Read ALU	MemRead, lorD = 1	Read memory using the ALUOut as address; write result into MDR.
	Write ALU	MemWrite, lorD = 1	Write memory using the ALUOut as address, contents of B as the data.
	ALU	PCSource = 00 PCWrite	Write the output of the ALU into the PC.
PC write control	ALUOut-cond	PCSource = 01, PCWriteCond	If the Zero output of the ALU is active, write the PC with the contents of the register ALUOut.
	jump address	PCSource = 10, PCWrite	Write the PC with the jump address from the instruction.
	Seq	AddrCtl = 11	Choose the next microinstruction sequentially.
Sequencing	Fetch	AddrCtl = 00	Go to the first microinstruction to begin a new instruction.
	Dispatch 1	AddrCtl = 01	Dispatch using the ROM 1.
	Dispatch 2	AddrCtl = 10	Dispatch using the ROM 2.

Microprogramming

- *Fetch: Need two microinstructions to complete the action.*
- *Mem1: Calculation of the memory address*
- *LW2: Need 2 microinstructions (memory read followed by register file write)*
- *Rformat1: Need 2 microinstructions (ALU operation followed by register file write)*

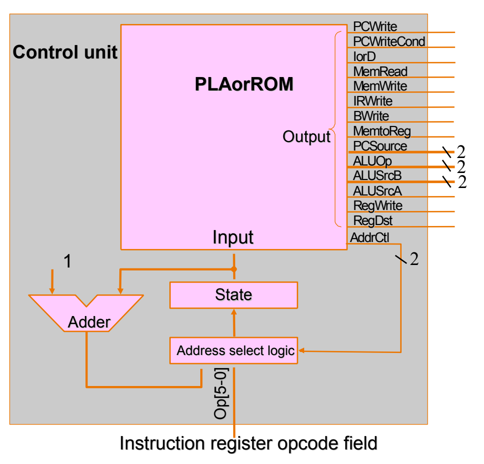
Label	ALU control	SRC1	SRC2	Register control	Memory	PCWrite control	Sequencing
Fetch	Add	PC	4		Read PC	ALU	Seq
	Add	PC	Extshft	Read			Dispatch 1
Mem1	Add	A	Extend				Dispatch 2
LW2					Read ALU		Seq
				Write MDR			Fetch
SW2					Write ALU		Fetch
Rformat1	Func code	A	B				Seq
				Write ALU			Fetch
BEQ1	Subt	A	B			ALUOut-cond	Fetch
JUMP1						Jump address	Fetch

Microprogramming

- *A specification methodology*
  - *appropriate if hundreds of opcodes, modes, cycles, etc.*
  - *signals specified symbolically using microinstructions*
- *Indeed, microprogramming can be thought of as a text representation of a finite state machine.*

Maximally vs. Minimally encoded microprogramming

- *Maximally - No encoding:*
  - *1 bit for each datapath control signal (e.g., 19 bit control word in the graph shown right)*
  - *faster, requires more memory (ROM approach) or logic (PLA approach)*
  - *used for Vax 780 — an astonishing 400K of memory!*



Maximally vs. Minimally encoded microprogramming

- *Minimally - Lots of encoding:*
  - *send the microinstructions through* ***combinational logic*** *to get control signals*
    - *E.g., if only eight possible control signal values are possible for 8-bit wide control word, we can use a 3-to-8 decoder to generate the control word through a narrower 3-bit microinstruction.*
  - *Slower*
  - *Example. The designers of the 8086 considered its 21-bit microinstruction to be more "minimally coded" than other single-chip computers of the time.*

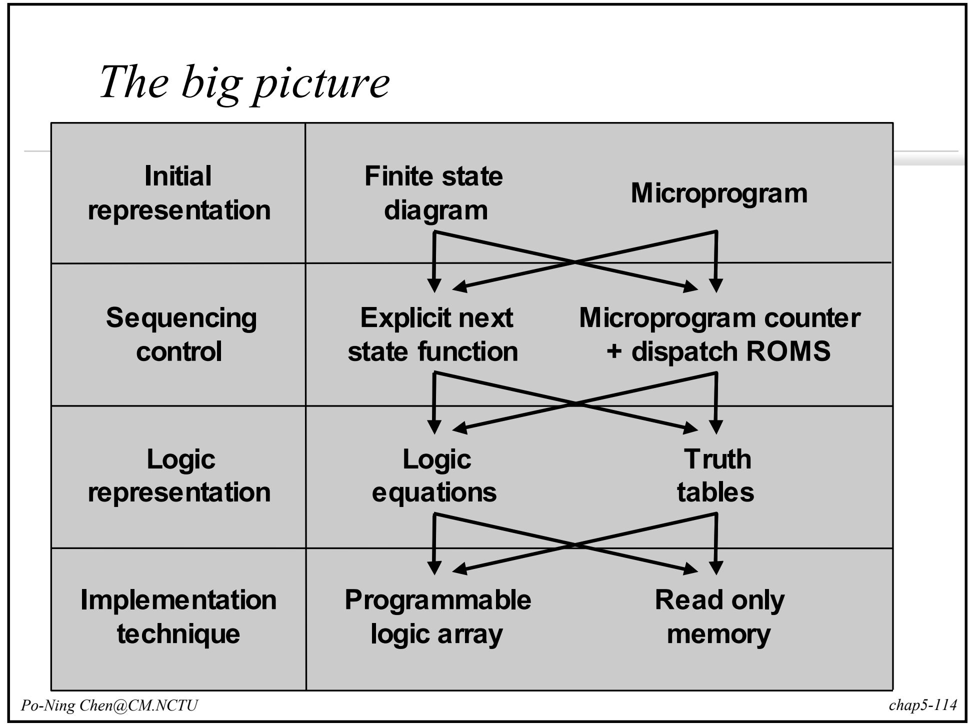
Maximally vs. Minimally encoded microprogramming

- *Historical context of CISC:*
  - *Use a ROM (or even RAM) to hold the microcode*
  - *It's easy to add new instructions*

Microcode: Trade-offs

- *Implementation of off-chip ROM Advantages*
  - *Easy to change since values are in memory*
  - *Can emulate other architectures*
  - *Can make use of internal registers*
- *Implementation Disadvantages, SLOWER now that:*
  - *ROM is no longer faster than RAM*

The big picture



Exception and interrupt

- *An exception is an unexpected event from within the processor*
  - *E.g., arithmetic overflow, divide-by-zero, and undefined instruction.*
- *An interrupt is an event that also causes an unexpected change in control flow but comes from outside of the processor.*
  - *Usually used by I/O devices to communicate with the CPU*
- *In INTELx86 system, both exceptions and interrupts are treated as "interrupts".*
- *In MIPS system, exception is referred to all unexpected events that cause the change of control flow, while interrupt is restricted to those unexpected events originating from external devices.*

Exception and interrupt

- *For the CPU, what to do when exception occurs?*
- *Answer:*
  - *No idea. The CPU simply* ***stores*** *the current situation, and then* ***transfers*** *the PC to some pre-specified address. E.g., C00000H for the MIPS machine.*
  - *It is the Operating System who shall pre-store various exception-handling program onto the "pre-specified address".*
  - *The exception-handling program then perform necessary correcting to the system, according to the value in the Exception Status Register, which records the cause of the exception.*

Exception and interrupt

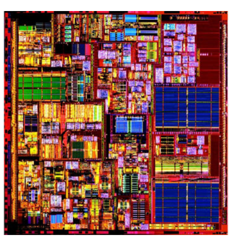
- *In some system, CPU may be capable of jumping to several exception-handling addresses according to the cause of exceptions, a technique named "vectored interrupts."*
  - *E.g., undefined instruction -> C0000000H, and arithmetic overflow -> C0000020H.*
- *This ease a little the burden of the operating system for exception handling.*
- *INTEL uses a dynamic vectored interrupt, in which the starting address of the exception-handling program is stored in a fixed place in a 4-byte form.*

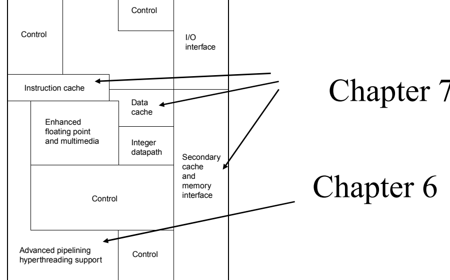
Historical Perspective

- *In the '60s and '70s microprogramming was very important for implementing machines*
- *This led to more sophisticated ISAs and the VAX*
- *In the '80s RISC processors based on pipelining became popular*
- *Pipelining the microinstructions is also possible!*
- *Implementations of IA-32 architecture processors since 486 use:*
  - *"hardwired control" for simpler instructions*
    (few cycles, FSM control implemented using PLA or random logic)
  - *"microcoded control" for more complex instructions*
    (large numbers of cycles, central control store)
- *The IA-64 architecture uses a RISC-style ISA and can be implemented without a large central control store*

Pentium 4

- *Pipelining is important (the last IA-32 without it was 80386 in 1985)*



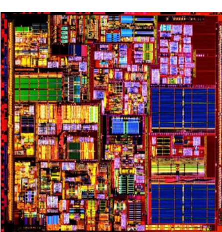


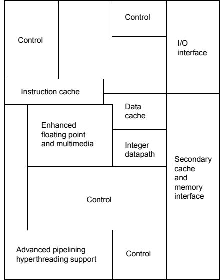
- *Pipelining is used for the simple instructions favored by compilers*

  “Simply put, a high performance implementation needs to ensure that the simple instructions execute quickly, and that the burden of the complexities of the instruction set penalize the complex, less frequently used, instructions”

Pentium 4

- *Somewhere in all that “control we must handle complex instructions*





- *Processor executes simple microinstructions, 70 bits wide (hardwired)*
- *120 control lines for integer datapath (400 for floating point)*
- *If an instruction requires more than 4 microinstructions to implement, control from microcode ROM (8000 microinstructions)*
- *It’s complicated!*

Summary

- *If we understand the instructions…*
  We can build a simple processor!
- *If instructions take different amounts of time, multi-cycle is better*
- *Datapath implemented using:*
  - *Combinational logic for arithmetic*
  - *State holding elements to remember bits*
- *Control implemented using:*
  - *Combinational logic for single-cycle implementation*
  - *Finite state machine for multi-cycle implementation*

Suggestive exercises

- *5.1~5.6, 5.8~5.18, 5.29~5.30, 5.32, 5.47, 5.50, 5.51, 5.58*