

# DDCO

## UNIT - 5

CLASS NOTES

feedback/corrections: vibha@pesu.pes.edu

Vibha Masti

## ADDRESSING MODES

- Method/ technique/ mechanism used to specify location of an operand

### 1) Register Only

- Operands found in registers
- example:

add \$s0, \$t2, \$t3  
sub \$t8, \$s1, \$0

### 2) Immediate

- 16-bit immediate operands
- example:

add \$s4, \$t5, -73  
ori \$t3, \$t7, 0xFF

### 3) Base Addressing

- address of operand is  
(base address + sign-extended immediate)
- example :

lw \$s4, 72(\$0)  
sw \$t2, -25(\$t1)

#### 4) PC Relative

0x10      beq \$t0, \$0, else  
 0x14      addi \$v0, \$0, 1  
 0x18      addi \$sp, \$sp, i  
 0x1C      jr \$ra  
 0x20      else: addi \$a0, \$a0, -1  
 0x24      j factorial

always increments first  
 3 more to reach

#### Assembly Code

beq \$t0, \$0, else  
 (beq \$t0, \$0, 3)

#### Field Values

op	rs	rt	imm
4	8	0	3

6 bits      5 bits      5 bits      5 bits      6 bits

that's why  
3

#### 5) Pseudo direct Addressing \*

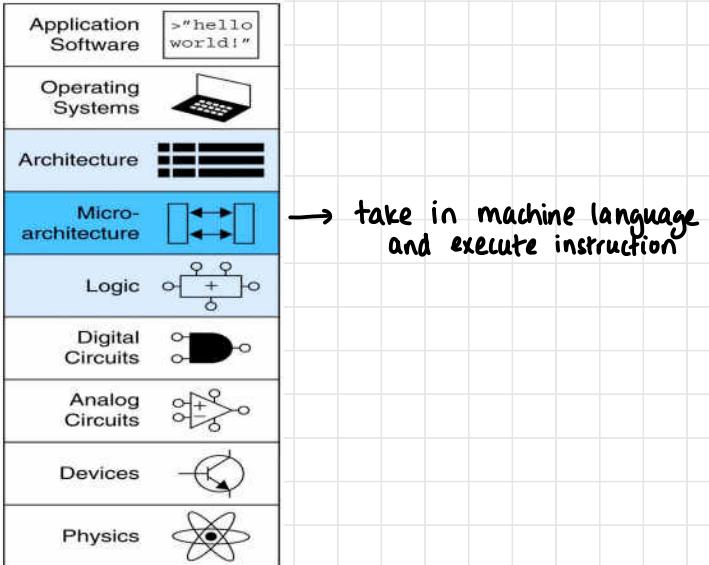
0x0040005C      j      sum

...

0x004000A0      sum: add      \$v0, \$a0, \$a1

jump target address	J T A	0000 0000 0100 0000 0000 0000 1010 0000 (0x004000A0)													
		26-bit addr	0000	0000 0100 0000 0000 0000 1010 0000 (0x0100028)	0	1	0	0	0	2	8				
Field Values										Machine Code					
<table border="1"> <tr> <td>op</td> <td>imm</td> </tr> <tr> <td>3</td> <td>0x0100028</td> </tr> </table>		op	imm	3	0x0100028	<table border="1"> <tr> <td>op</td> <td>addr</td> </tr> <tr> <td>000011</td> <td>00 0001 0000 0000 0000 0010 1000 (0xC100028)</td> </tr> </table>		op	addr	000011	00 0001 0000 0000 0000 0010 1000 (0xC100028)			↑ divisible by 4	
op	imm														
3	0x0100028														
op	addr														
000011	00 0001 0000 0000 0000 0010 1000 (0xC100028)														

## MIPS Microarchitecture



## Architecture

- Architectural state: PC, 32 registers, Memory (state block in FSM)
- Architecture : instruction set specification, architectural state

## Microarchitecture

- How to implement an architecture in logic/ hardware
- Processor:
  - Datapath: functional blocks
  - Control: control signals
- Multiple microarchitectures for same architecture (vary in speed and power)

- Multiple implementations for a single architecture

- inefficient**
- Single-cycle: each instruction executes in a single cycle
  - Pipelined: each instruction broken into a series of steps and multiple instructions executed at once
  - Multi-cycle: each instruction broken into a series of shorter steps  
**we study**

## Processor Performance

$$\text{Execution time} = \frac{\text{no.of instructions}}{\text{instruction}} \times \frac{\text{cycles}}{\text{cycle}} \times \frac{\text{seconds}}{\text{cycle}}$$

- Definitions:

**reciprocals**

- CPI: cycles per instruction
- clock period: seconds per cycle (reciprocal of clock frequency)
- IPC: instructions per cycle (usually less than 1)

- Optimise cost, power, performance (resources, energy, speed)

## Subset of MIPS Instructions

- R-type instructions and, or, add, sub, slt, sll
- Memory instructions lw, sw
- Branch instructions beq

set on less than  
shift left  
(barrel shifter)

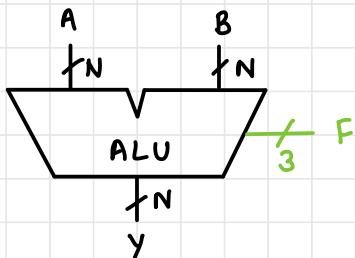
Intel and Advanced Micro Devices (AMD) both sell compatible microprocessors conforming to the x86 architecture. Intel Pentium III and Pentium 4 microprocessors were largely advertised according to clock frequency in the late 1990s and early 2000s, because Intel offered higher clock frequencies than its competitors. However, Intel's main competitor, AMD, sold Athlon microprocessors that executed programs faster than Intel's chips at the same clock frequency. Why?

- instructions / clock cycle
- frequency not the only factor; lower clock frequency but better performing
- Zen, Zen 2 - AMD microarchitectures that implement x86 architecture

### MULTI-CYCLE PROCESSOR

- Processor: datapath (functional blocks), control (control signals)
  - ↓ where data flows
  - ↓ manipulates data
- Register file + ALU

### ALU - Arithmetic and Logic Unit



F <sub>2:0</sub>	Function
000	A & B AND
001	A   B OR
010	A + B add
011	not used
100	A & ~B
101	A   ~B
110	A - B sub
111	SLT shift left

ALU Control[2:0]

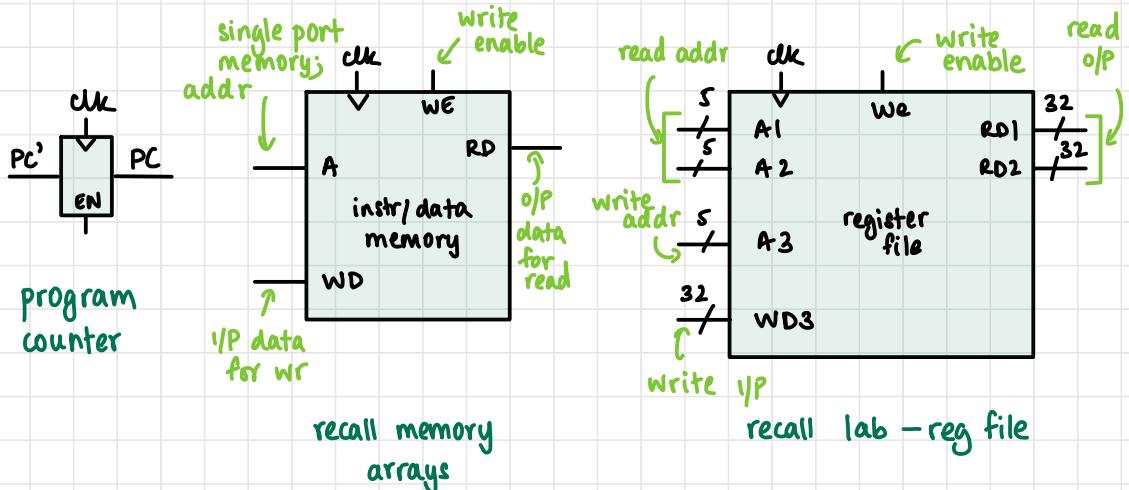
8 functions:  
3 inputs

## Architectural State

- PC
- Registers
- Memory

3 components

→ outside microprocessor ; external

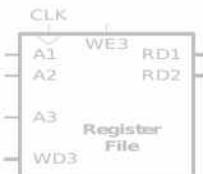
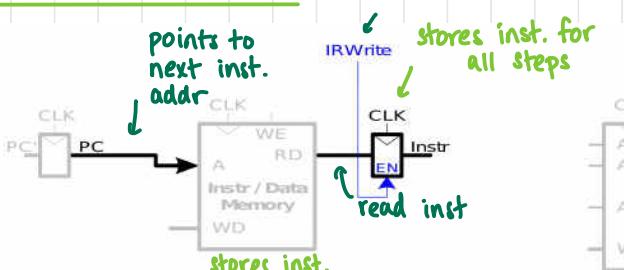


## Load Word Instruction

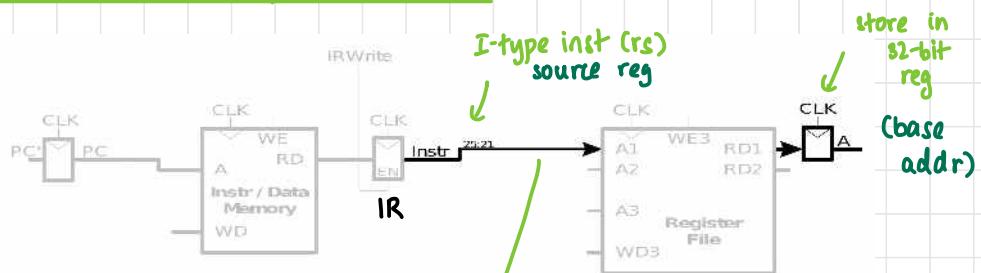
LW \$50, 5(\$t1) → load from memory  
 $\text{rt}$        $\text{rs}$

- Address calculation  
 base address (\$t1) + offset (5)

## Step 0 - Fetch Instruction



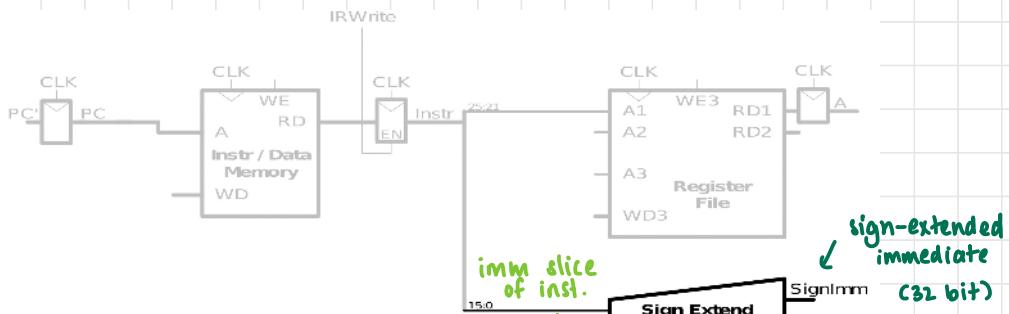
## Step 1 - Read source operands from RF



Recall: I-type instruction

OP	rs	rt	imm
6 bits	5 bits	5 bits	16 bits
31:26	25:21	20:16	15:0

## Step 2 - Read & Sign Extend the Immediate

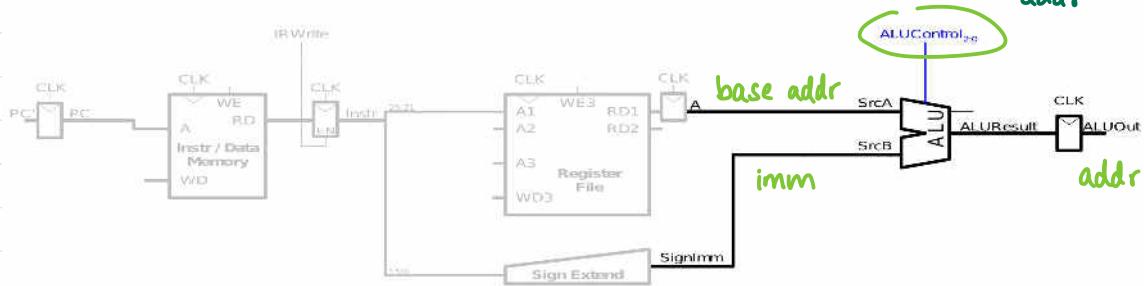


Recall: I-type instruction

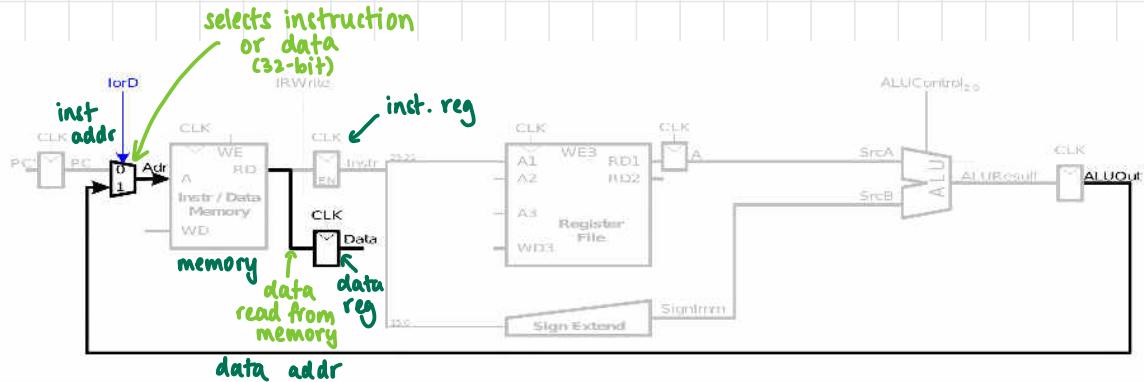
OP	rs	rt	imm
6 bits	5 bits	5 bits	16 bits
31:26	25:21	20:16	15:0

### Step 3 - Compute Memory Address

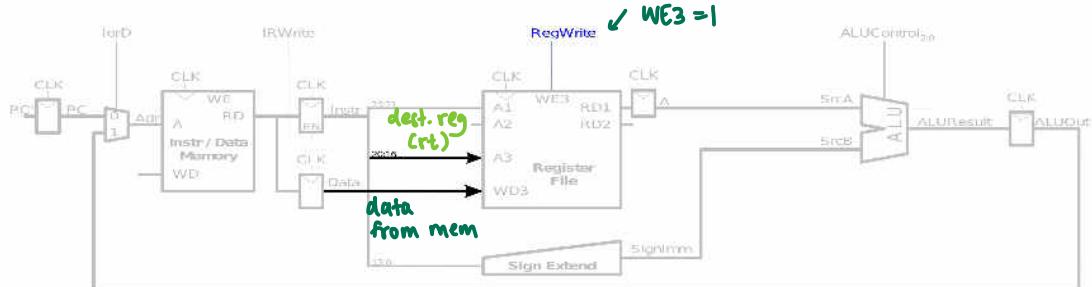
add immediate to base addr



### Step 4 - Read from Memory



### Step 5 - Write data back into RF

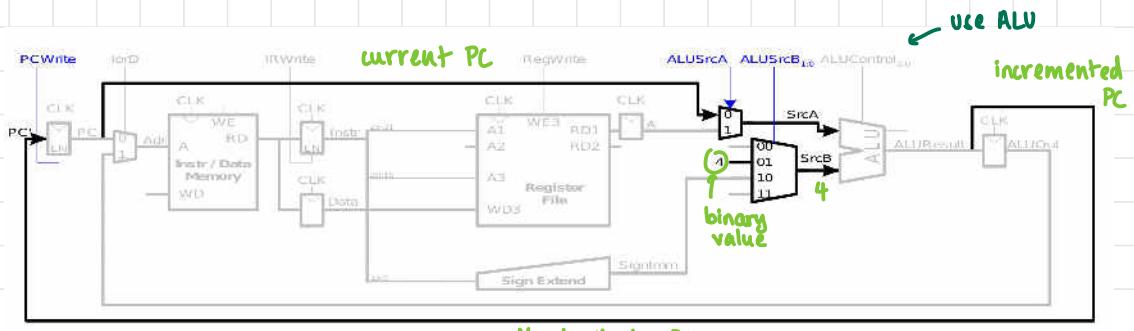


Recall: I-type instruction

OP	rs	rt		imm
6 bits	5 bits	5 bits		16 bits
31:26	25:21	20:16		15:0

### Step 6 - Increment PC

- increment by 4 bytes (32-bit is 4 bytes, byte-addressable)



write back to PC

- Only one adder (inside ALU)
- In lab datapath design, two adders used (separate adder for PC)
- Would using two adders in current design help? How?
- Note that in both cases, architecture remains the same

## Other Instructions

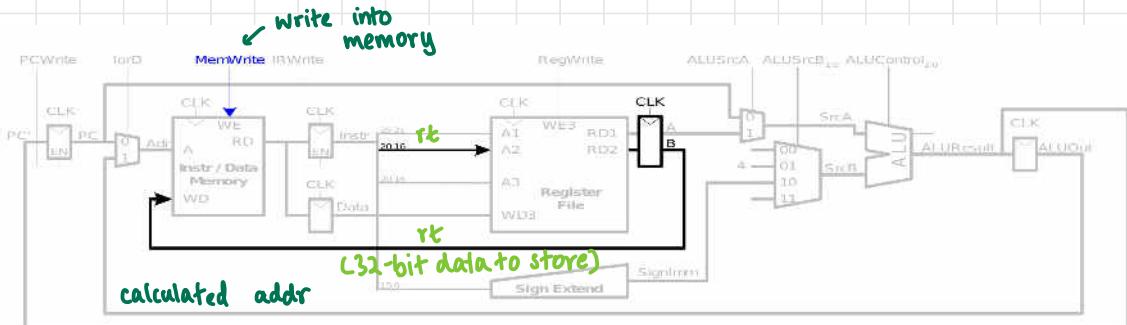
- SW
- R-type (add, sub, and, ...)
- beq

## Store Word Instruction

SW \$s0, 7(\$t0) # write the value in \$s0 to loc 7  
 $t_0 \quad r_s$

- Address computation same as for lw
- Register contents to be written to main memory
- Steps 1, 2 and 3 same as in lw
- Step 4: Register contents to be written into computed memory address

## Write to Memory



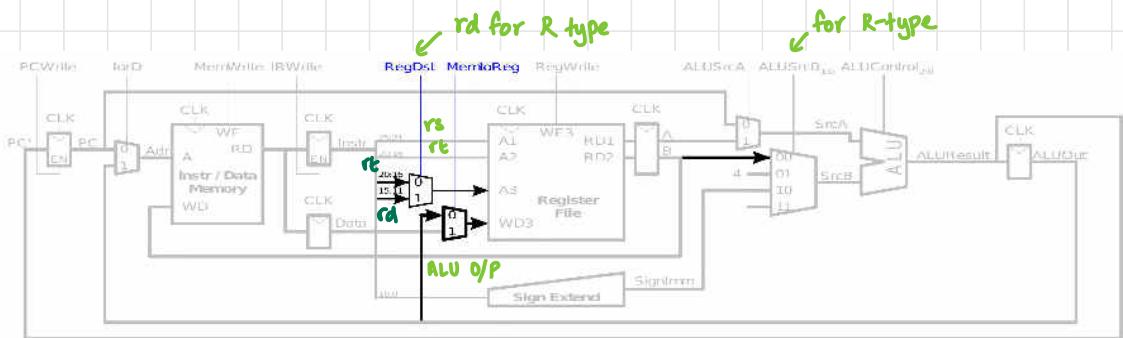
Recall: I-type instruction

OP	rs	rt	imm
6 bits	5 bits	5 bits	16 bits
31:26	25:21	20:16	15:0

## R-type Instructions

- Step 1 (Fetch): same as lw
  - Step 2: similar to lw (no sign extending; read 2 operands)
  - Step 3: write ALU output into destination register
- Read from rs and rt
- Write ALU Result to reg file
- Write to rd (instead of rt)

add \$s0, \$s1, \$s2  
 rd rs rt



Recall: R-type instruction

OP	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits
31:26	25:21	20:16	15:11	10:6	5:0

## beq, Instruction

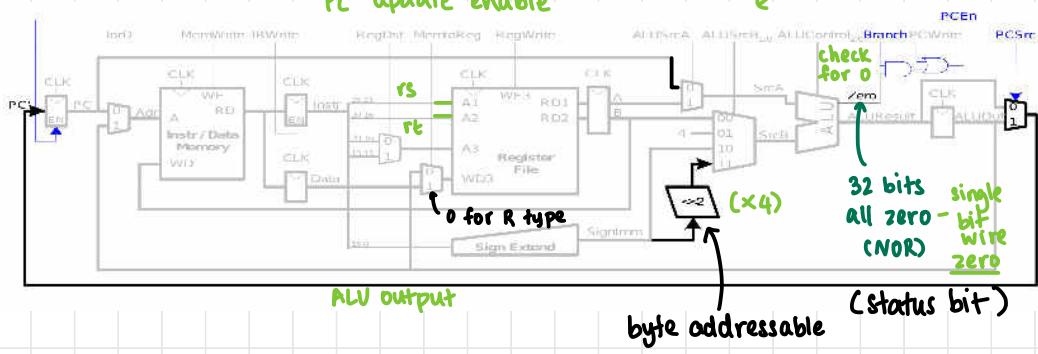
- Step 1: fetch
- Step 2: compare register contents
- Step 3: change PC contents (if registers equal)

beq \$s0, \$s1, loop (PC relative)

$rs == rt?$  offset value  $\times 4$  ( $<< 2$ ) next PC inst  
 $BTA = (\text{sign-extended immediate } << 2) + (\text{PC} + 4)$

PC update enable

↙ 00 then 11

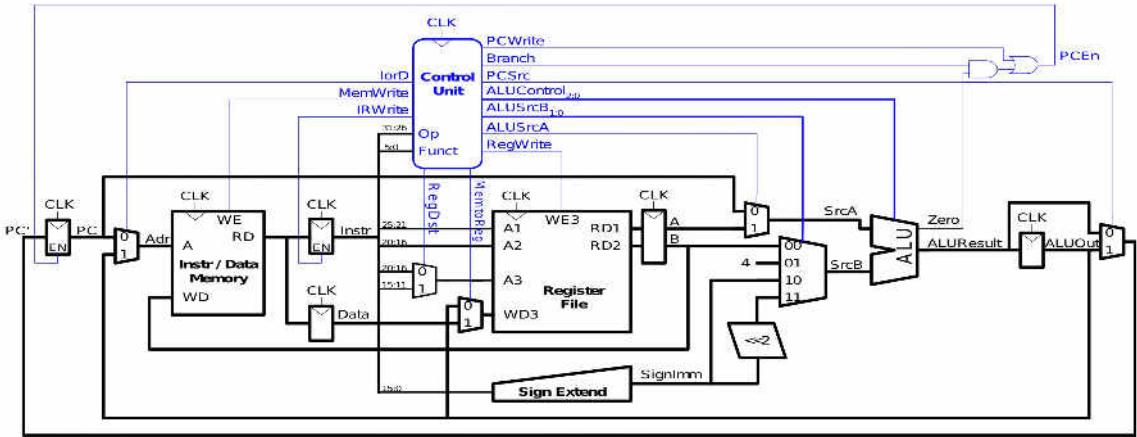


Recall: I-type instruction

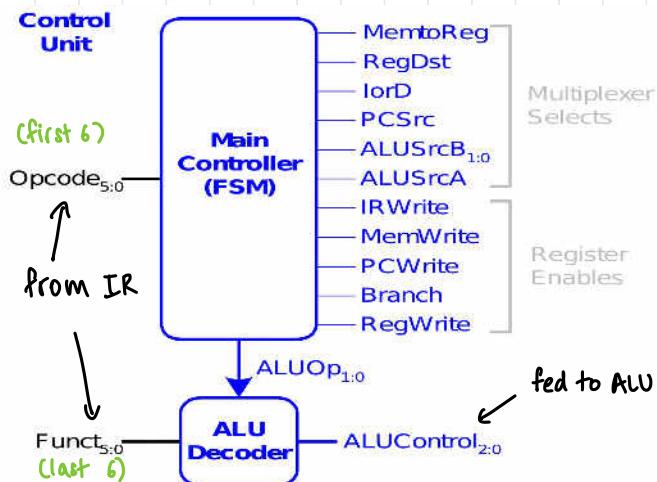
OP	rs	rt	imm
6 bits	5 bits	5 bits	16 bits
31:26	25:21	20:16	15:0

## Control Unit

- clockwise synchronisation

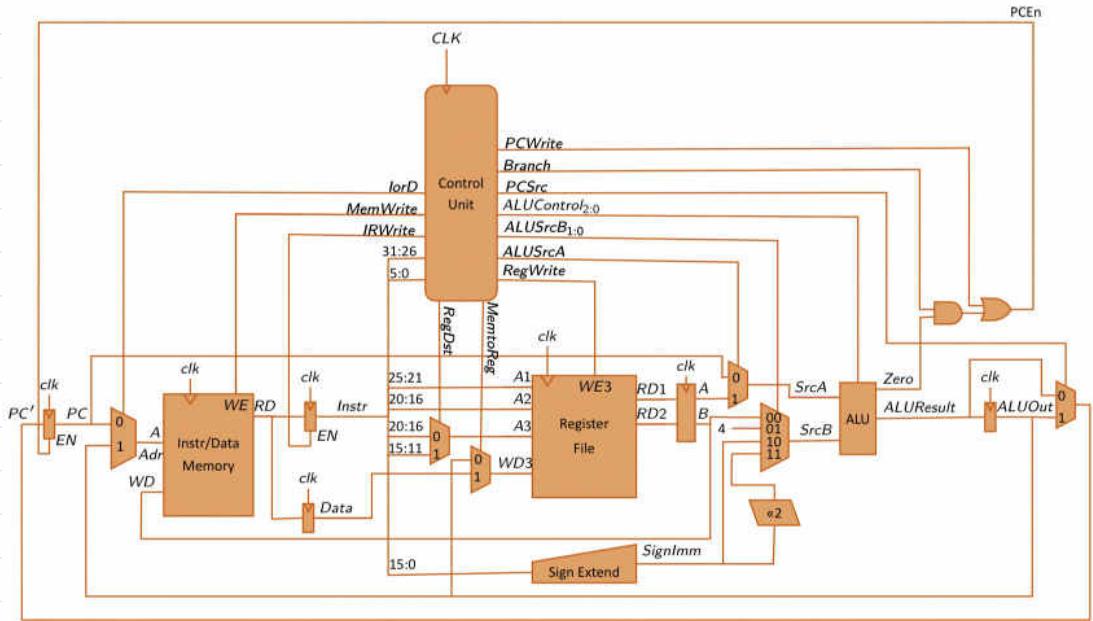


## Control Logic



ALUOp[1:0]	Funct[5:0]	ALUControl[2:0]	ALU Operation (instructions served)
00	don't care	010	+ (addi, lw, sw)
01	don't care	110	- (beq, bne)
1x	32 (100000)	010	+ (add)
1x	34 (100010)	110	- (sub)
1x	36 (100100)	000	∩ (and)
1x	37 (100101)	001	U (or)
1x	42 (101010)	111	- (slt)

## MIPS Multi-Cycle Datapath



## lw Instruction Steps

0. Fetch
1. Read Register
2. Read and Sign-extend immediate
3. Compute memory address
4. Read data from memory
5. Write data back to register file
6. Increment PC

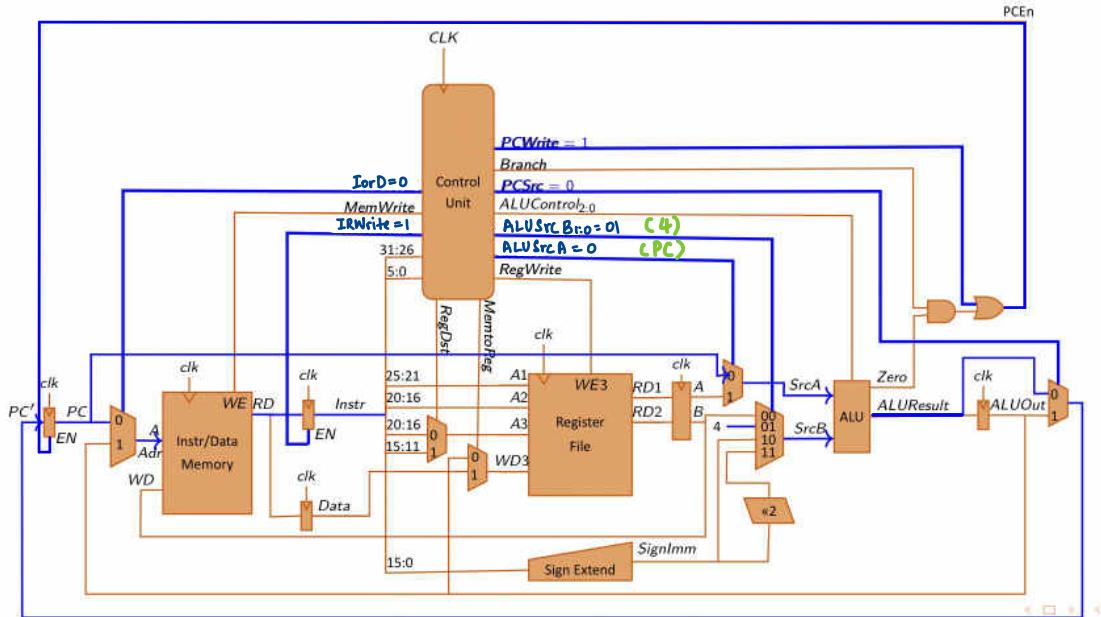
## Execute in Five Clock Cycles

1. Fetch and increment PC - no conflict in datapath
2. Read register and read/ sign extend immediate - no conflict
3. Compute memory address
4. Read data from memory
5. Write data back to register file

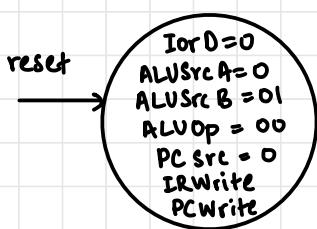
same order

# Clock cycle #1

- fetch instruction

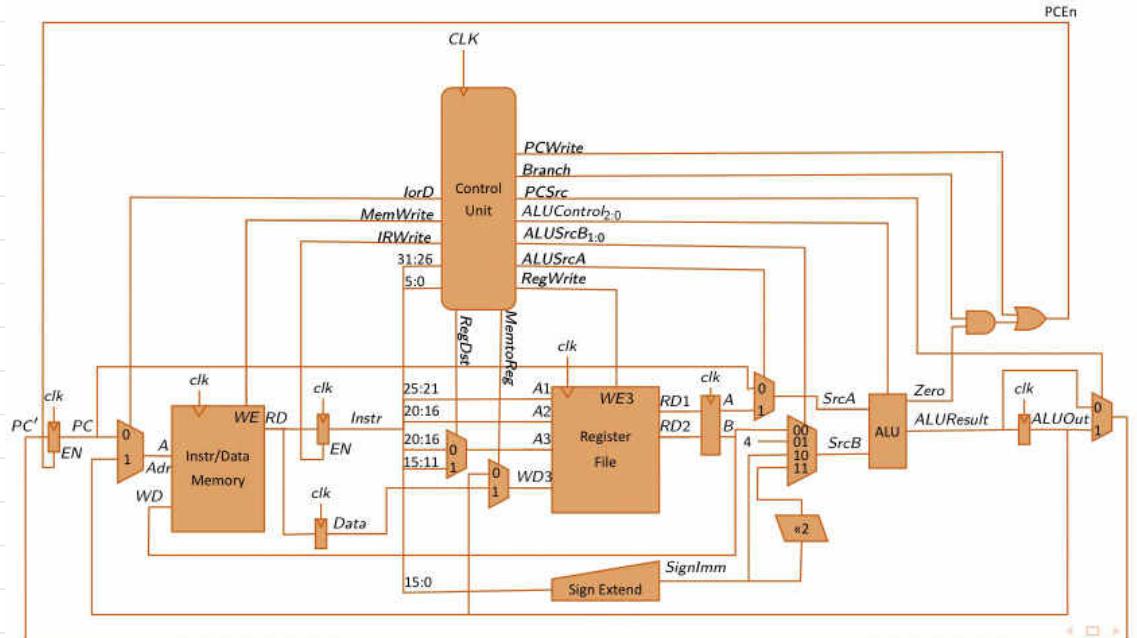


So: fetch



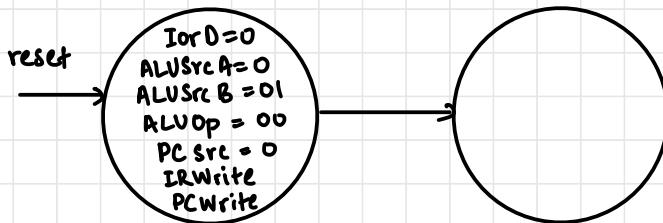
## clock cycle #2

- decode state

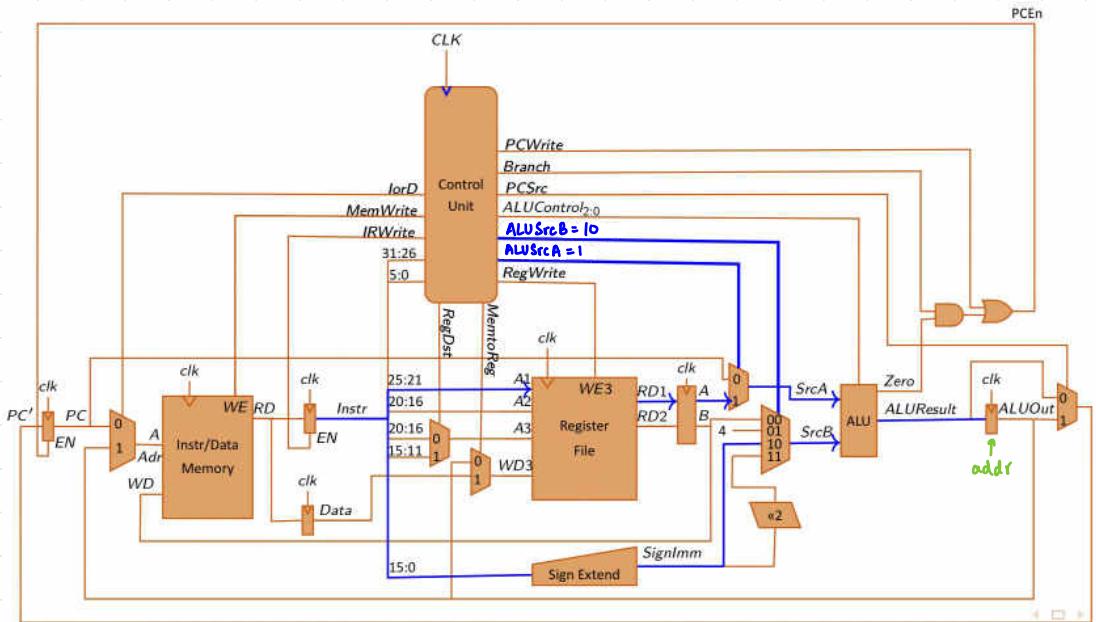


SO: fetch

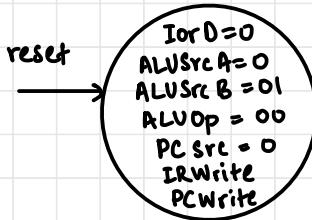
SI: decode



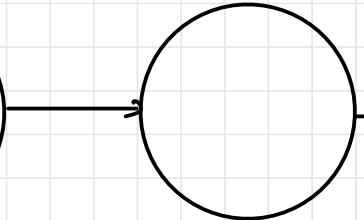
## Clock cycle #3



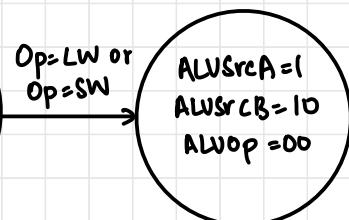
So: fetch



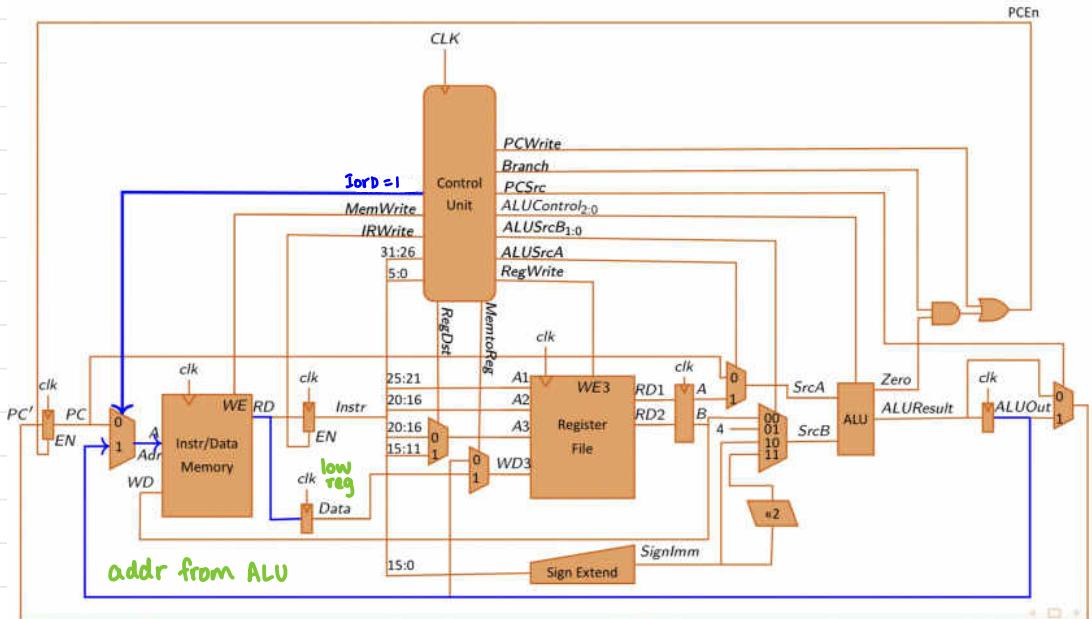
Sl: decode



S2: MemAddr



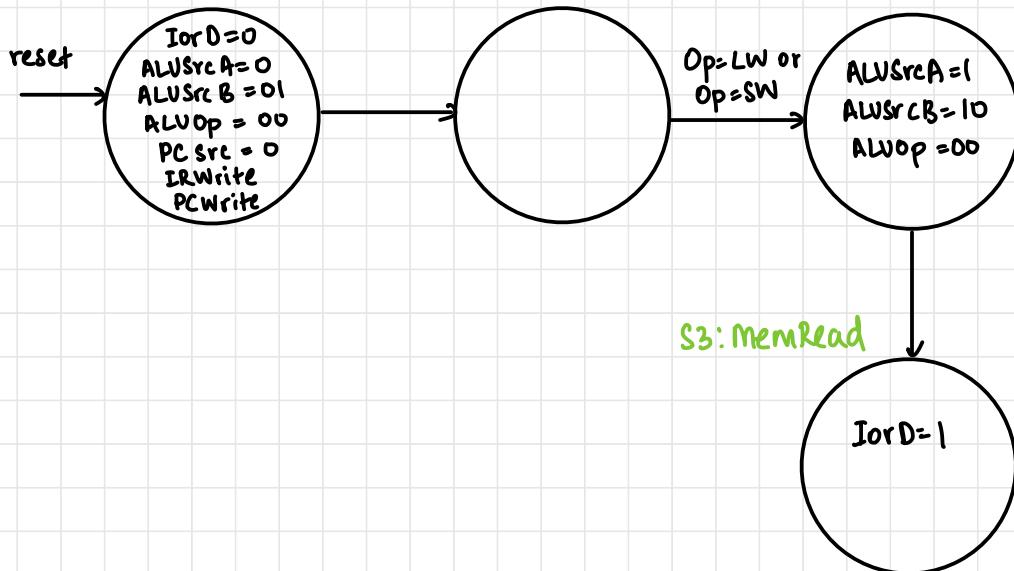
## Clock cycle #4



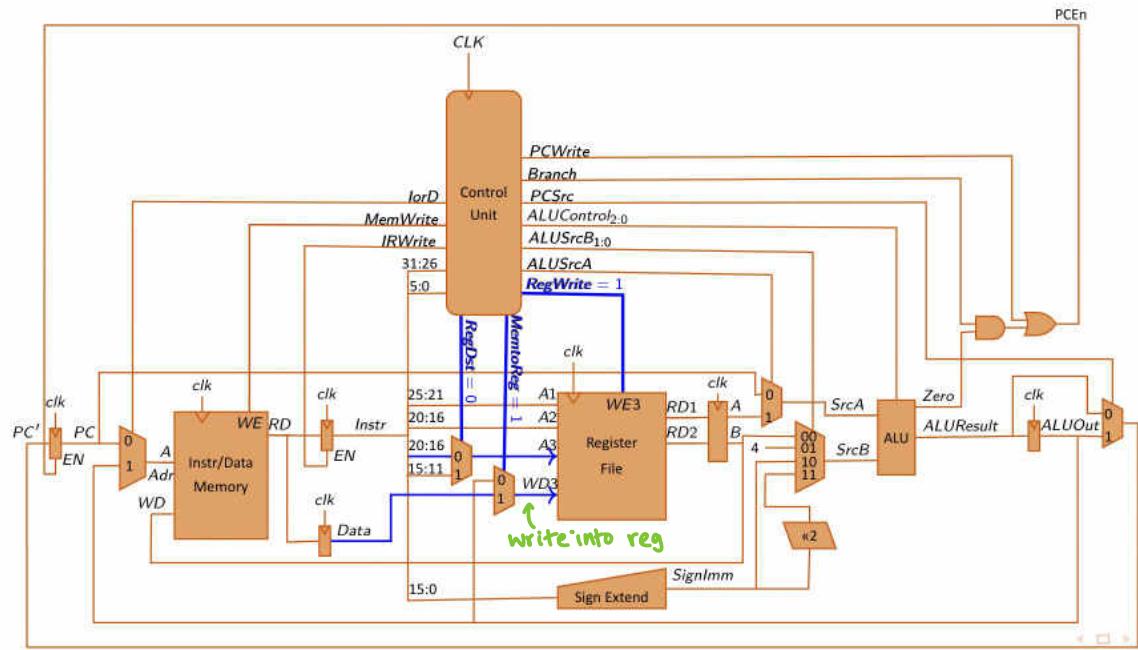
S0: fetch

S1: decode

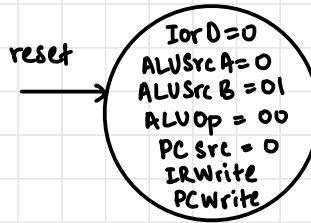
S2: MemAddr



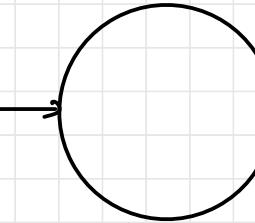
## Clock cycle #5



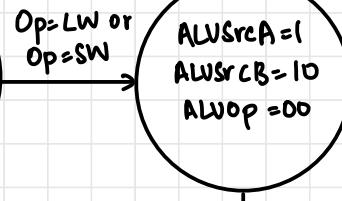
S0: fetch



S1: decode



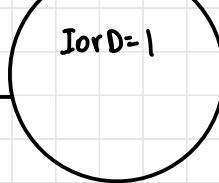
S2: MemAddr



S4: MemWriteBack



S3: MemRead

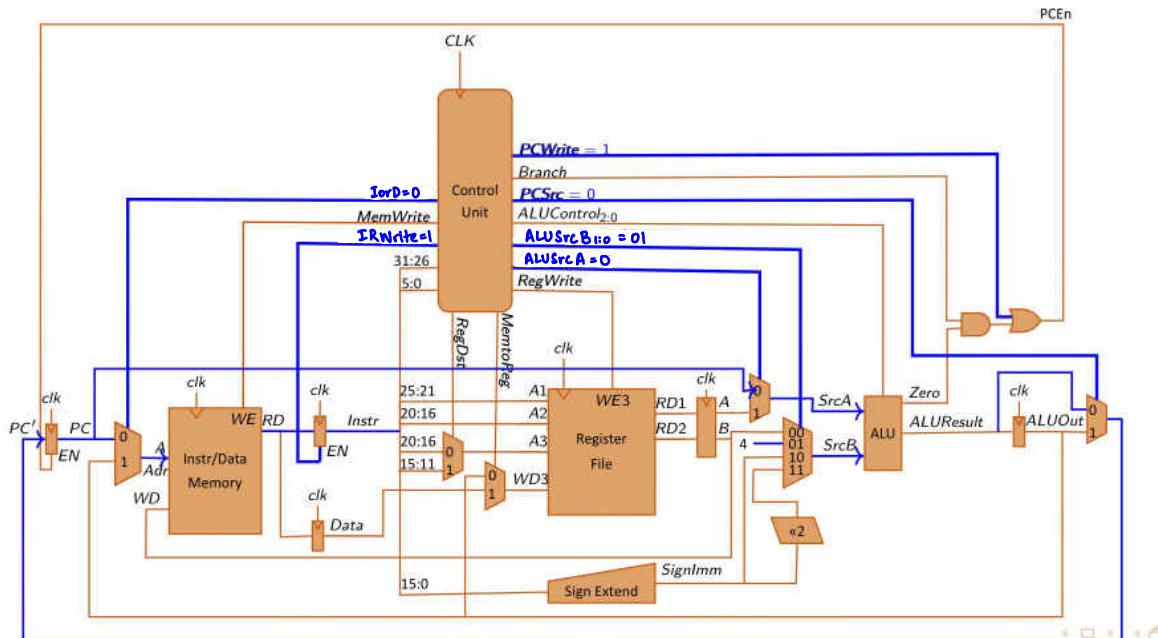


## Store Word Instruction

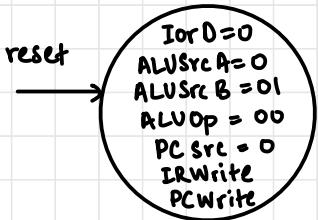
- 4 clock cycles

### Clock Cycle #1

- fetch instruction

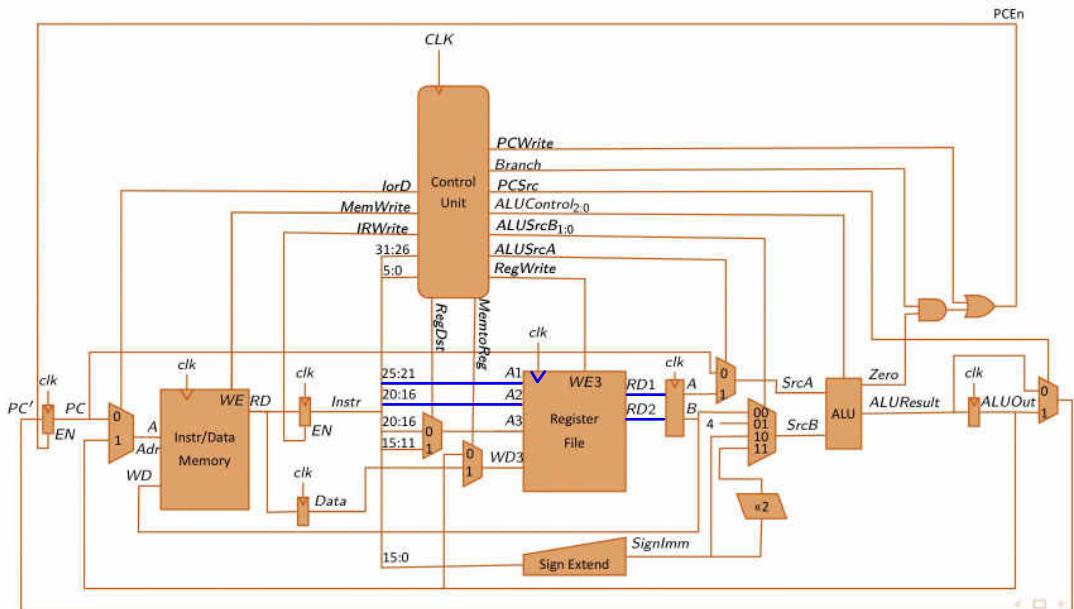


So: fetch



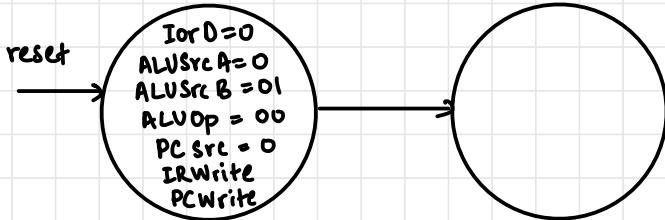
## Clock cycle #2

- decode state

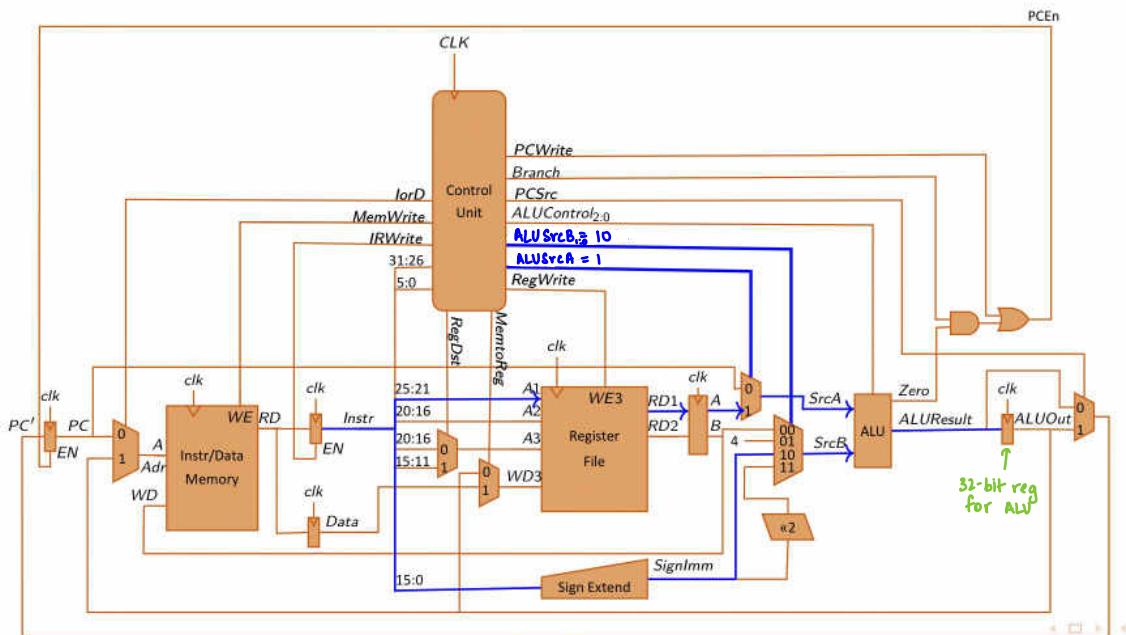


So: fetch

Si: decode



## Clock cycle #3

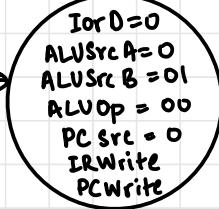


So: fetch

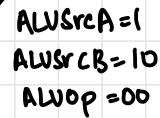
Sl: decode

S2: MemAddr

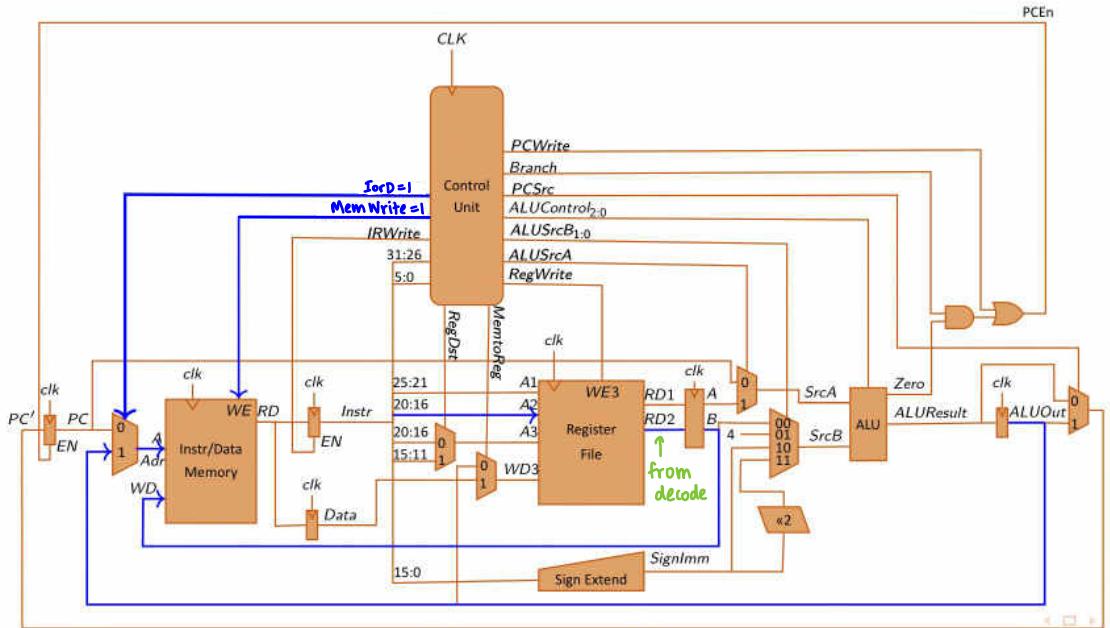
reset



$Op = LW \text{ or } Op = SW$



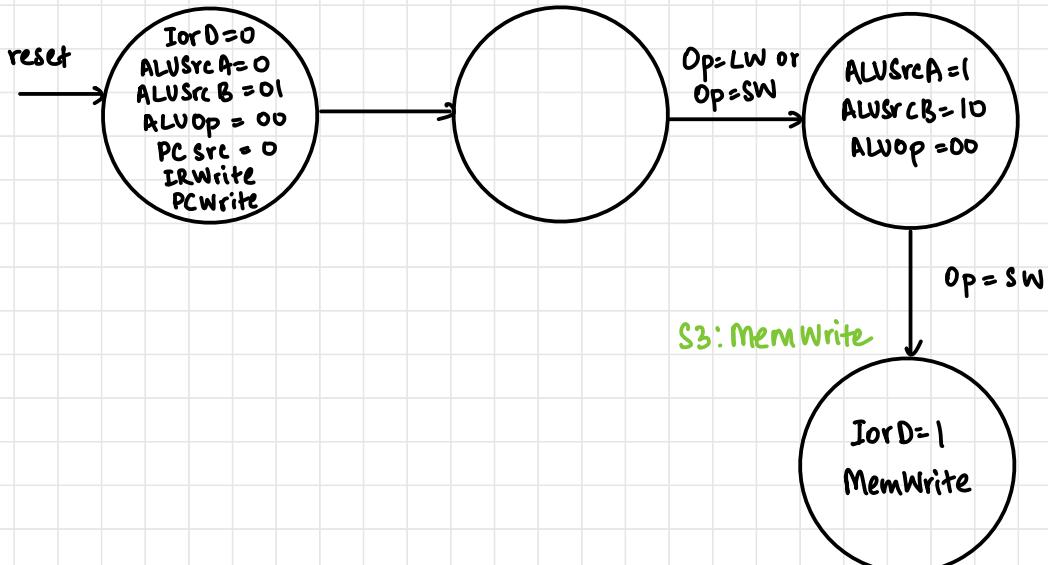
## Clock cycle #4



S0: fetch

S1: decode

S2: MemAddr

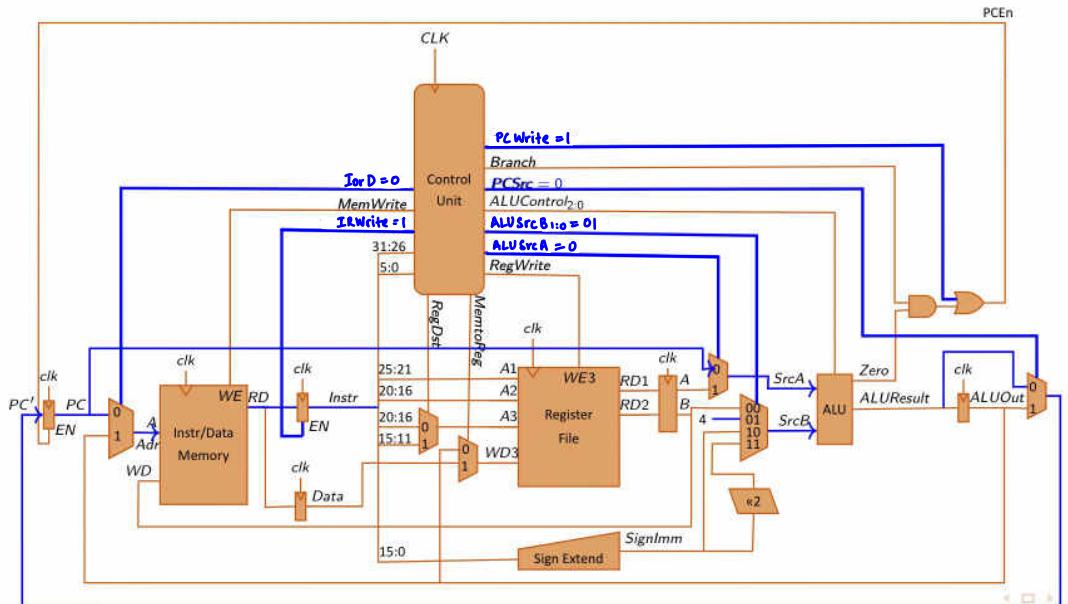


## R-Type Instruction

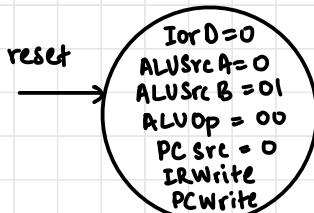
- 4 clock cycles

### Clock cycle #1

- fetch instruction

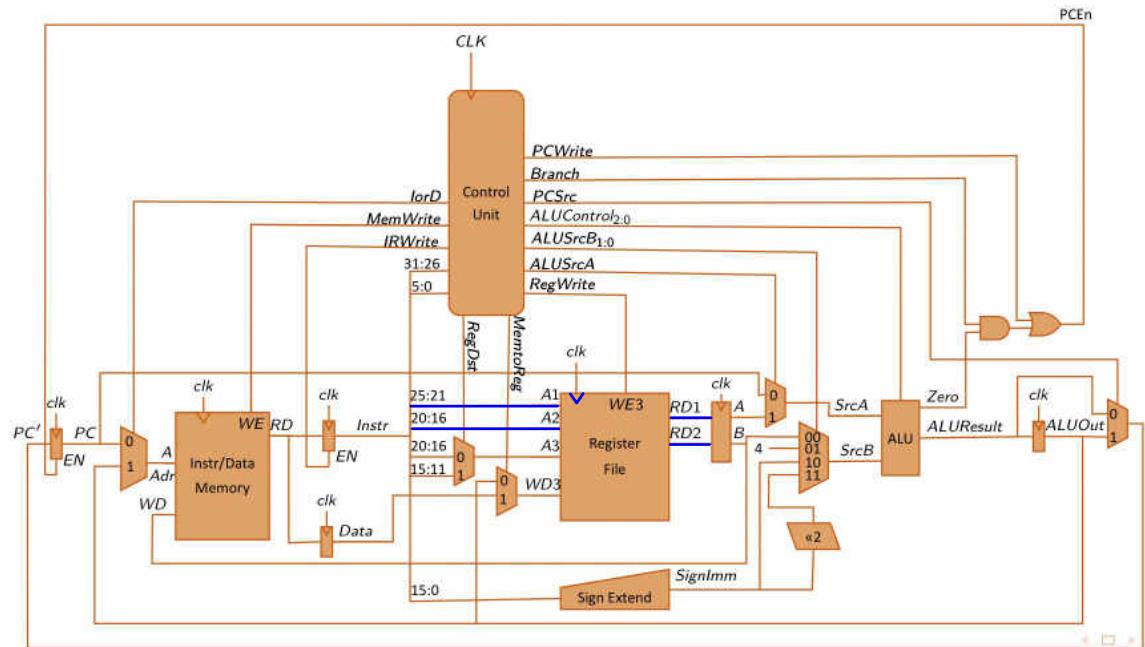


so: fetch



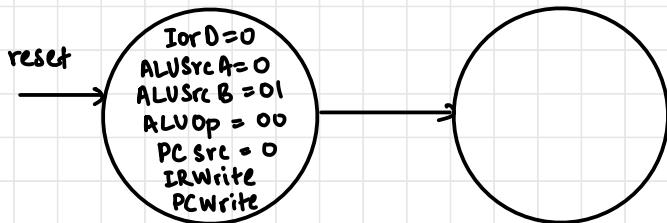
## Clock cycle #2

- decode state

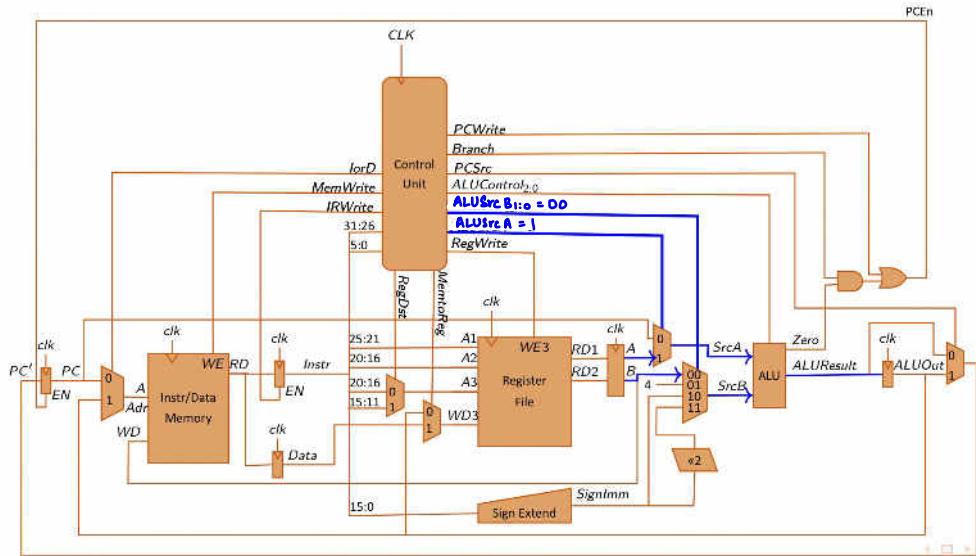


So: fetch

SI: decode



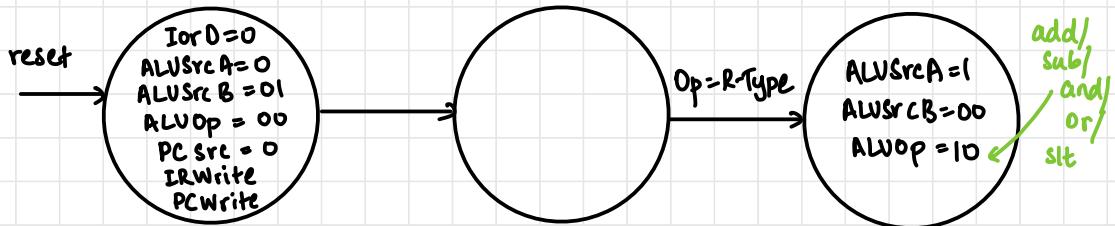
## clock cycle #3



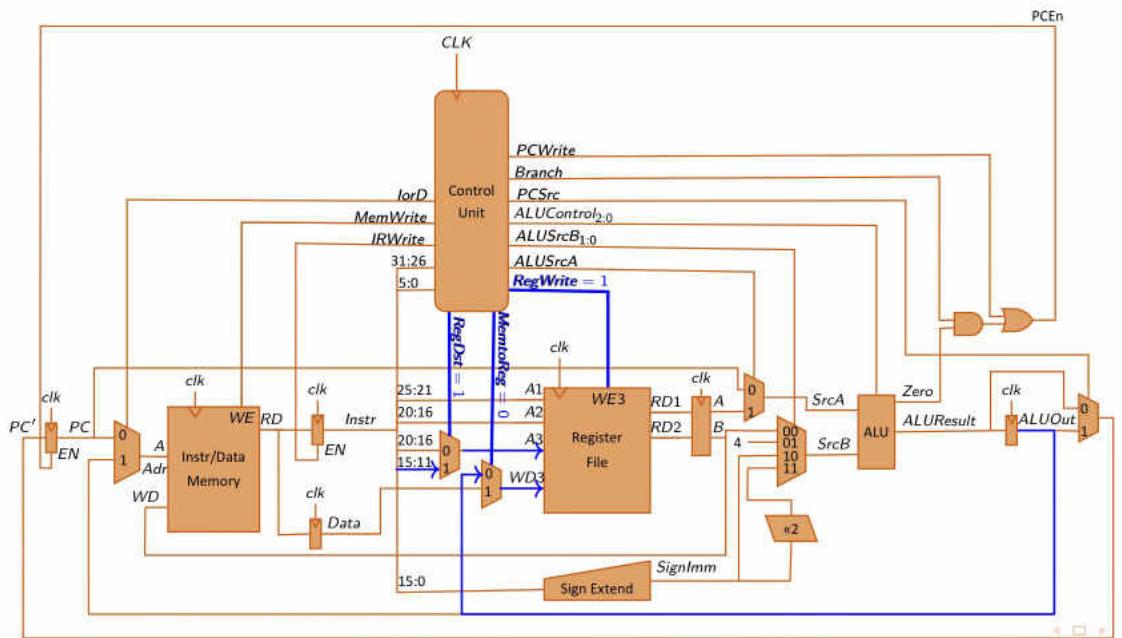
S0: fetch

S1: decode

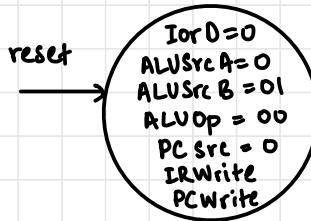
S6: execute



## Clock cycle #4



S0: fetch



S1: decode

S2: execute

*Op=R-Type*

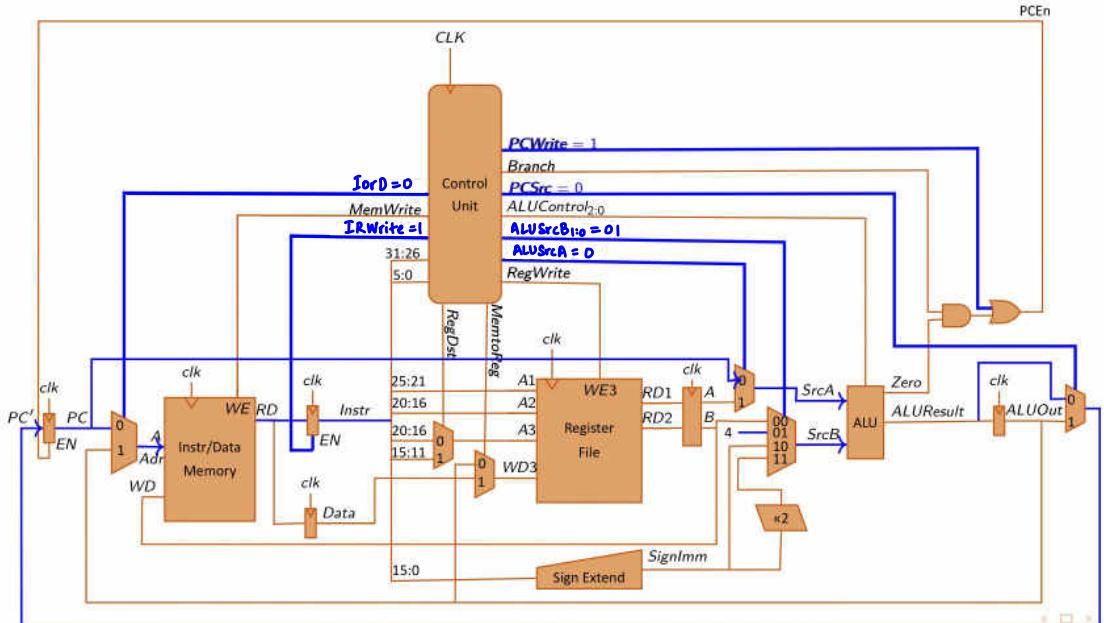
*ALUSrcA = 1*  
*ALUSrcB = 00*  
*ALUOp = 10*

S3: ALU Writeback

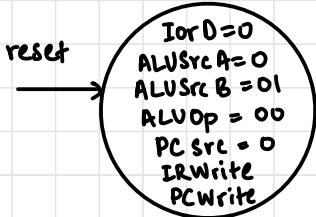
*RegDst = 1*  
*MemtoReg = 0*  
*RegWrite*

## Branch if Equal Instruction

Clock cycle #1

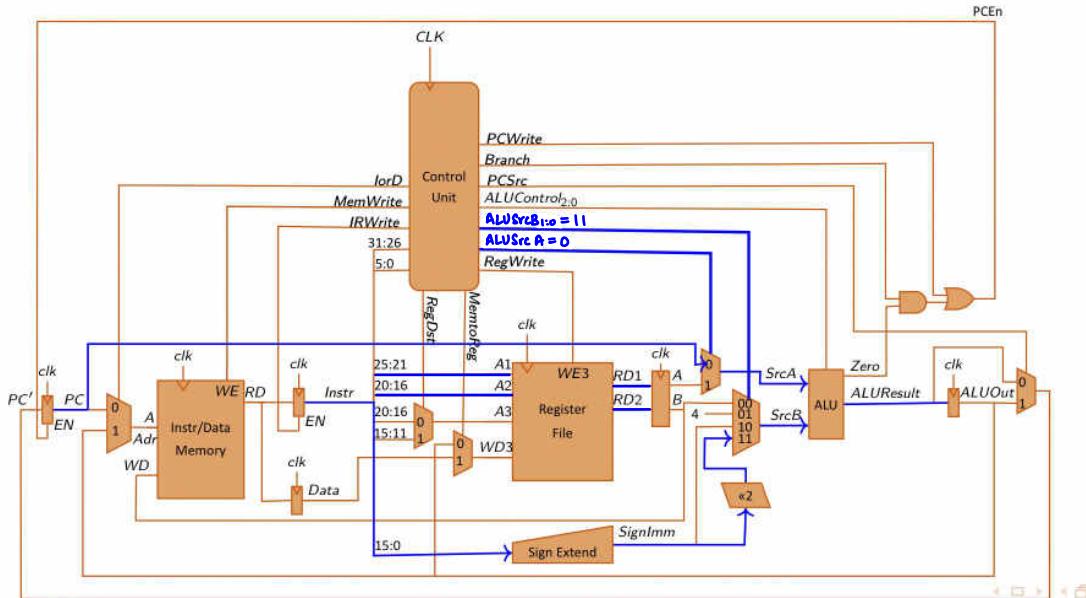


So: fetch



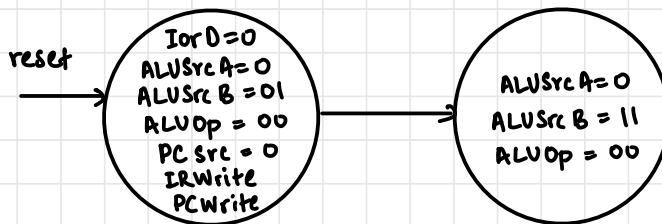
## Clock Cycle #2

- decode
- compute branch target address

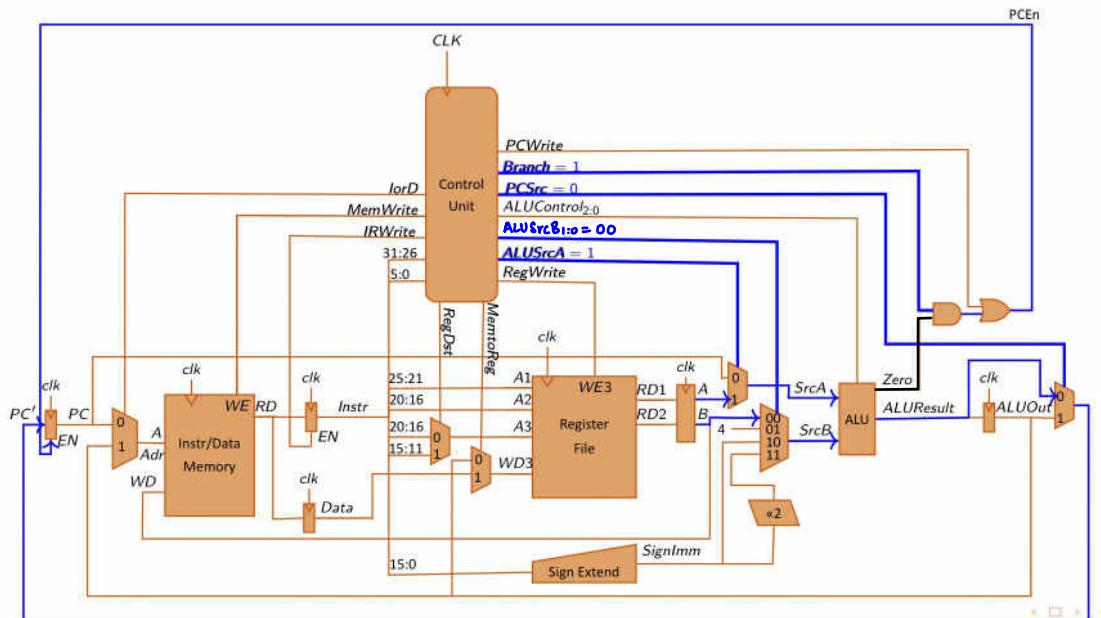


SO: fetch

SI: decode



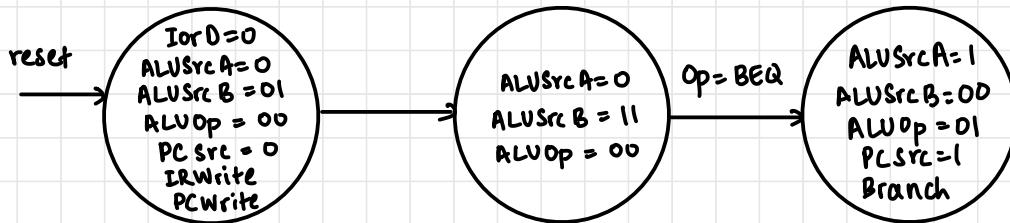
## Clock cycle #3



S0: fetch

S1: decode

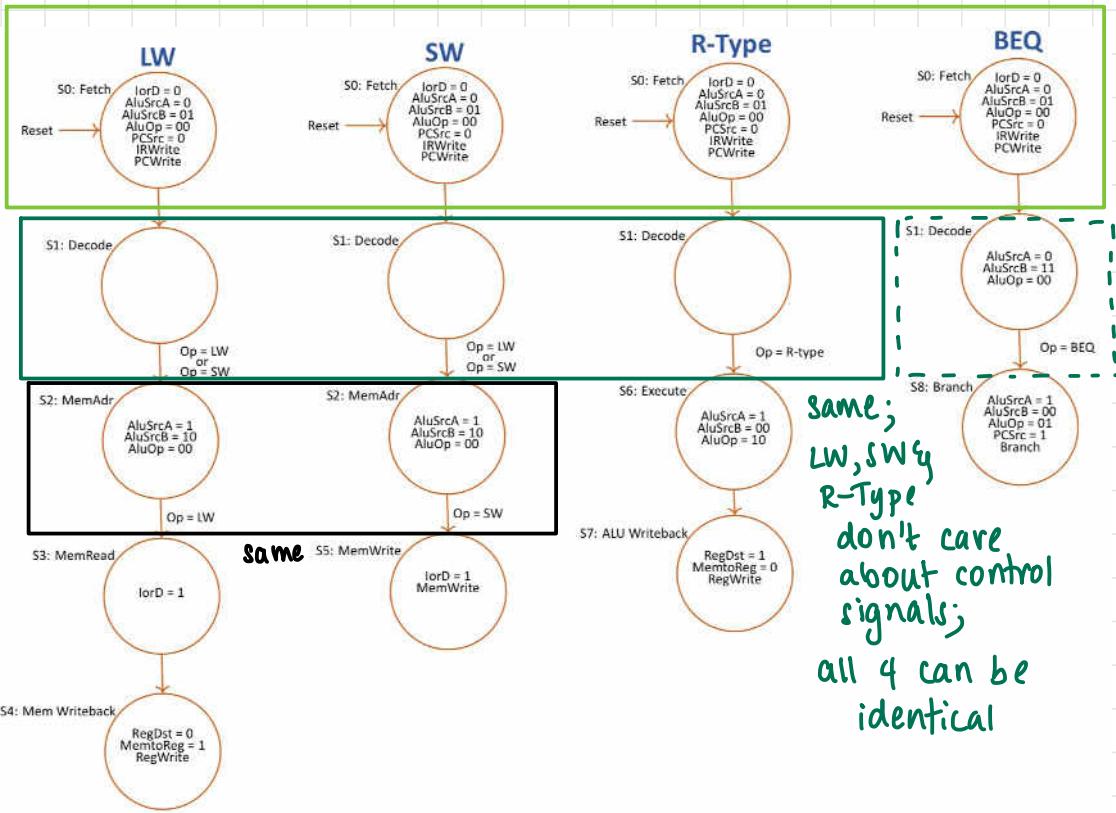
S2: Branch



## Control FSM

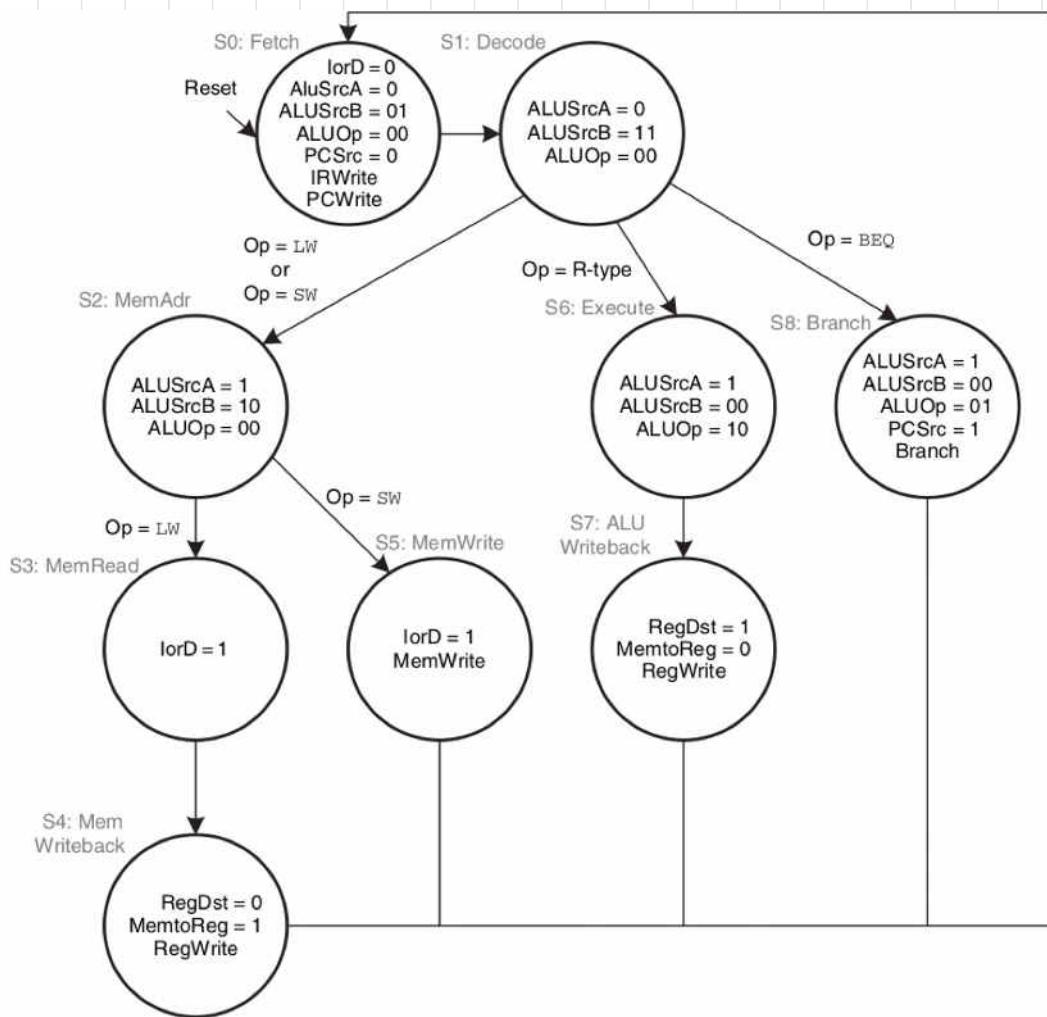
- For each instruction, FSMs

same

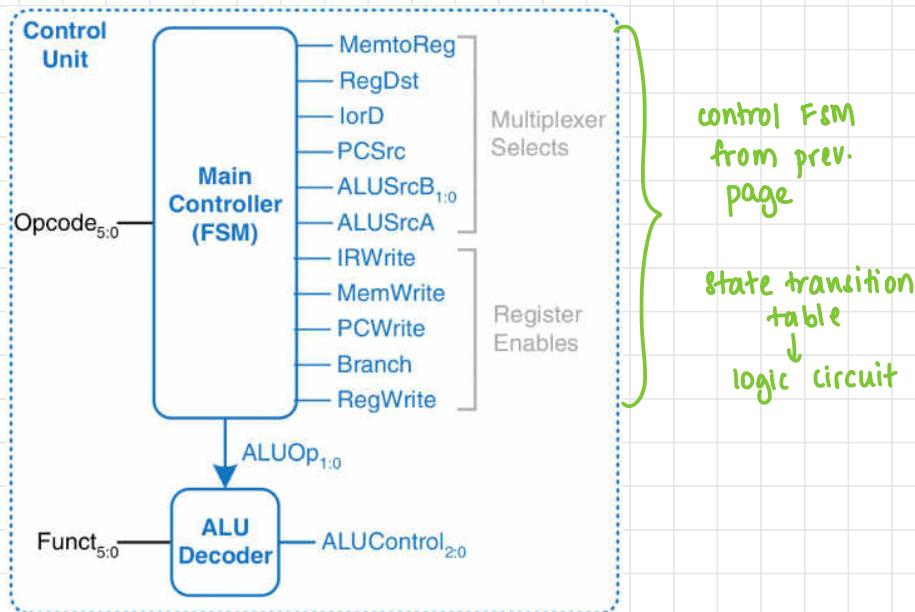


- Each final state connects back to first state (start state) to start next cycle
- First 2 states made common for all instructions, then branching for each instruction

## Control FSM for Multi-Cycle Datapath



## Control Unit



## ALU Decoder

ALUOp	Funct	ALUControl
00	X	010 (add)
X1	X	110 (subtract)
1X	100000 (add)	010 (add)
1X	100010 (sub)	110 (subtract)
1X	100100 (and)	000 (and)
1X	100101 (or)	001 (or)
1X	101010 (slt)	111 (set less than)

## Cycles per Instruction

- Performance measure
- As low as possible: goal

Instruction	CPI
lw	5
sw	4
R-type	4
beq	3

### Question !

no floating point

The SPECINT2000 benchmark consists approximately of 25% loads, 10% stores, 11% branches, 2% jumps and 52% R-type instructions. Determine the average CPI for this benchmark.

$$0.25 \times 5 + 0.10 \times 4 + 0.11 \times 3 + 0.02 \times 3 + 0.52 \times 4 \\ = 4.12 \text{ CPI}$$

## Systolic Array Matrix Multiply

- General purpose processor – functions specified by software
- Special purpose processor – hardware accelerators
- Focus: matrix multiplication  $(64 \times 64) \times 10$  matrices
  - ↪ Software time
  - ↪ Hardware time
  - ↪ Comparison

## Matrix Multiplication in Software

$$\cdot A_{m \times p} \times B_{p \times n} = C_{m \times n}$$

- 3 nested loops

```
for (i = 0; i < m; ++i) {  
    for (j = 0; j < n; ++j) {  
        for (k = 0; k < p; ++k) {  
            C[i][j] += A[i][k] * B[k][j];  
        }  
    }  
}
```

$m \times n \times p$   
times

## Computation Performed Every Iteration

- Innermost statement
- Floating point operations - 2: multiplication followed by addition
- Computations each iteration
  - loop variable increment and comparison
  - conditional branch
  - floating point multiply and add
- Assume all three computations occur in parallel in every iteration
- Assume time required each iteration is time required to perform floating point multiplication and addition

- Assume 2  $64 \times 64$  matrices  $\Rightarrow m, n, p = 64$

```

for (i = 0; i < m; ++i) {
    for (j = 0; j < n; ++j) {
        for (k = 0; k < p; ++k) {
            C[i][j] += A[i][k] * B[k][j];
        }
    }
}

```

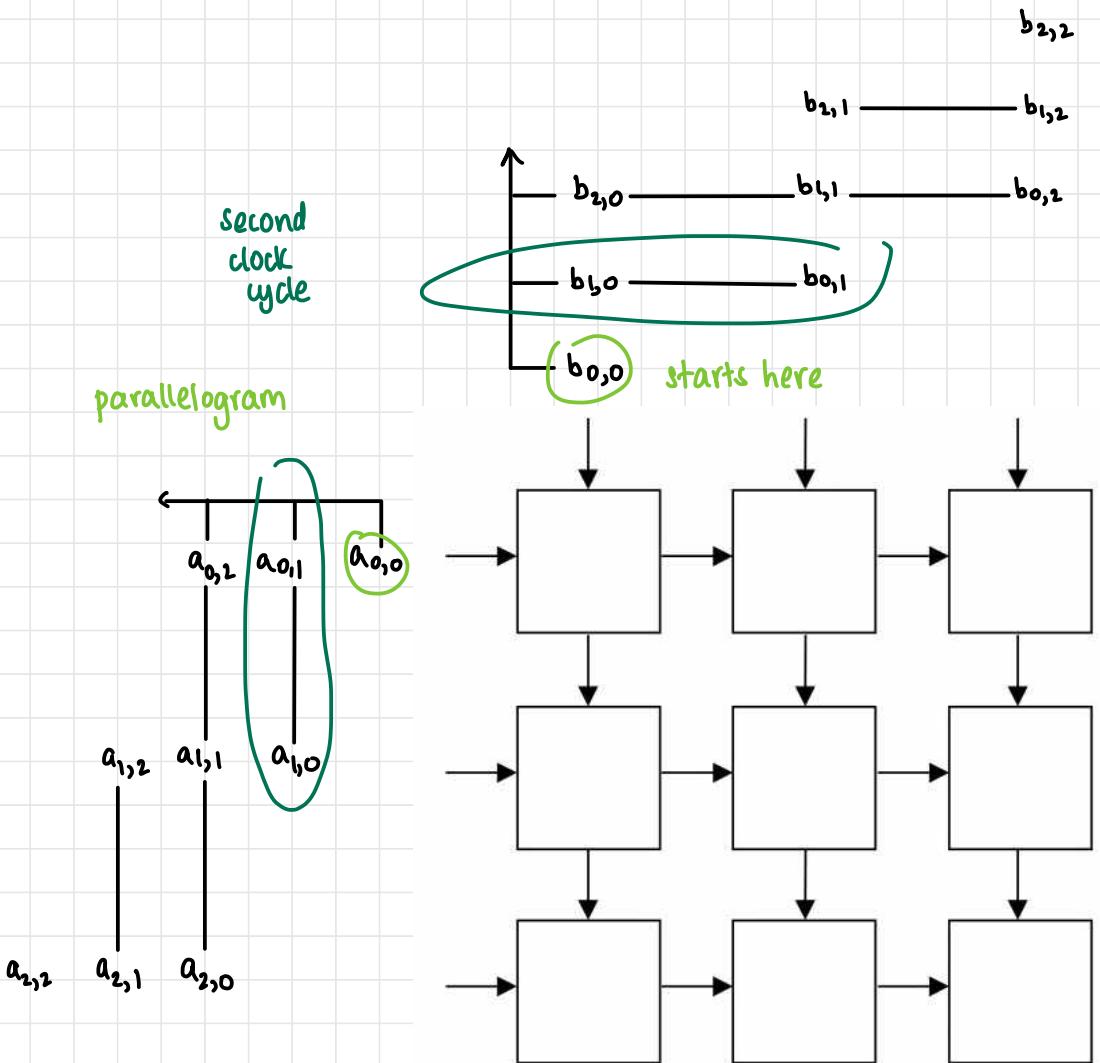
- Assume clock speed of microprocessor = 4 GHz  $\Rightarrow$  clock period = 0.25 ns
- Assume one FPO done every clock cycle
- $\therefore$  each loop iteration takes 2 clock cycles = 0.5 ns
- Number of loop iterations =  $64^3$
- $\therefore$  total time taken = 0.131 ms = 131072 ns
- Time taken in a ten-core microprocessor for 10 matrices (parallel)  
= 0.131 ms = 131072 ns

### Matrix Multiplication in Hardware

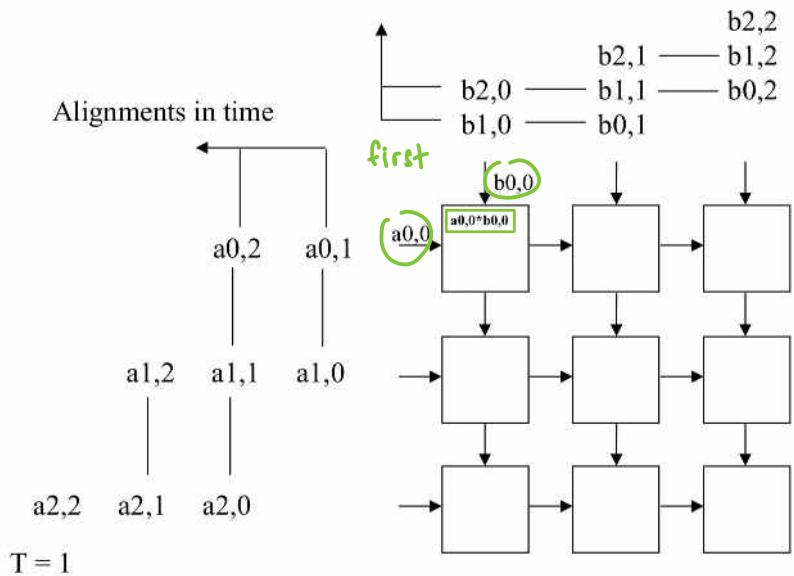
- Used 2D array of Processing Elements (PEs) for  $n \times n$  matrix multiplication
- Each PE can perform multiply + accumulate in same clock cycle
- Matrix multiplication in  $3n - 2$  cycles

## Systolic Array Multiplier

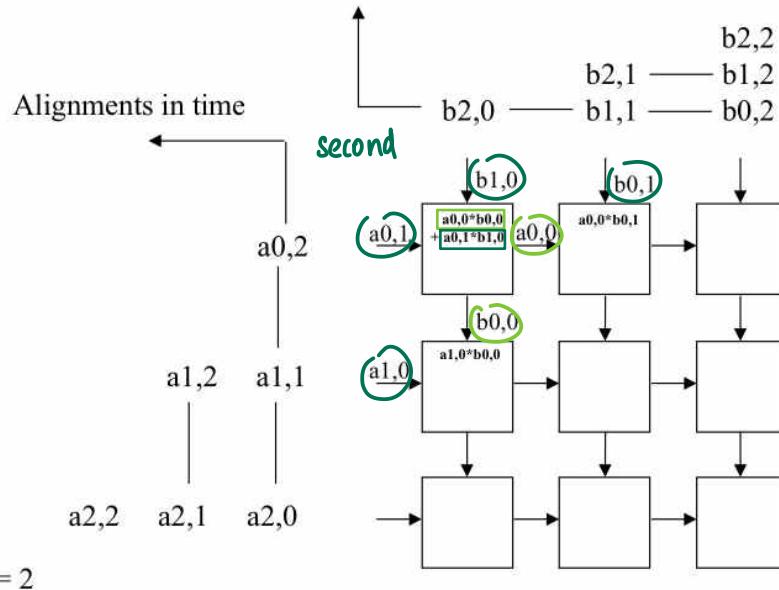
- $3 \times 3$  matrices



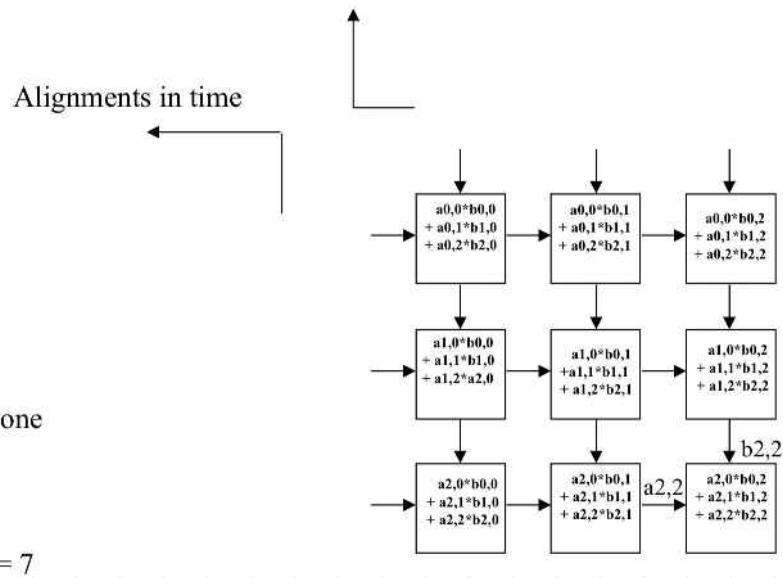
- $T=1$



- $T=2$



- So on, until final clock cycle
- Keeps accumulating & passing to neighbours



## Time Taken

- $3n-2$  clock cycles
- ASIC - Application Specific Integrated Circuit ; fast but expensive and have clock cycle  $\approx 1.5 \text{ ns}$
- FPGA - Field Programmable Gate Array : clock cycle  $\approx 4 \text{ ns}$
- For  $n=64$ ,  $3 \times 64 - 2 = 191$  clock cycles
- Time taken for matrix multiplication =  $191 \times 4 = 764 \text{ ns}$
- Time taken for 10 = 7640 ns

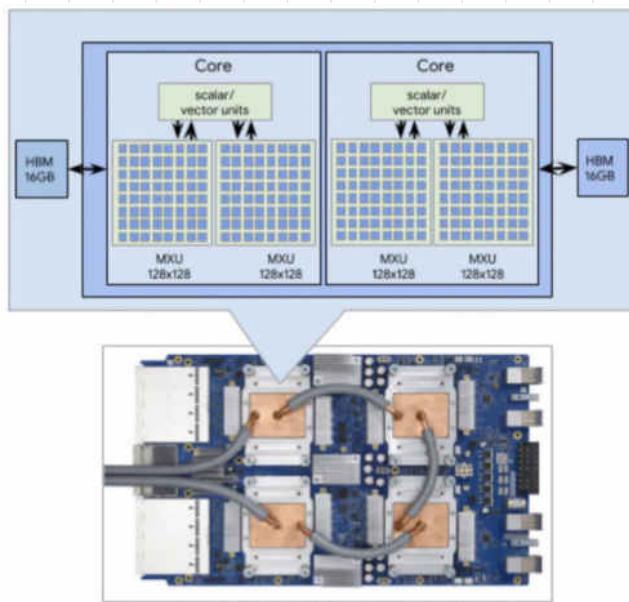
## Speedup for 10 64x64 Matrix Multiplies

$$= \frac{\text{software time}}{\text{hardware time}} = \frac{131072}{7640} \approx 17$$

- 17x faster using systolic array multiplier than microprocessor (general purpose hardware)

## Google Tensor Processing Unit (TPU)

- Thirst for computation higher but Moore's Law is halting
- Used for ML/AI <https://cloud.google.com/tpu>



TPU v3 - 4 chips, 2 cores per chip

- $128 \times 128$  systolic array multiplier
- 700 MHz (1.4 ns clock period)

## Think About It

- Systolic array  $n \times n$  matrix multiplication takes  $3n - 2$  clock cycles
  - But each PE computes only for  $n$  clock cycles
  - Can a new matrix multiplication be started before the previous one is complete?
  - How many matrix multiplies can be active in the systolic array at any given time?
- 
- Idle for  $\sim 2n$  clock cycles
  - 3 at once?