



EECS 318 CAD
Computer Aided Design

LECTURE 7:
Multicycle CPU

*Instructor: Francis G. Wolff
wolff@eecs.cwrn.edu*

Case Western Reserve University

This presentation uses powerpoint animation: please viewshow

MIPS instructions



ALU

alu \$rd,\$rs,\$rt

\$rd = \$rs <alu> \$rt

ALUi

alui \$rd,\$rs,value

\$rd = \$rs <alu> value

**Data
Transfer**

lw \$rt,offset(\$rs)

\$rt = Mem[\$rs + offset]

sw \$rt,offset(\$rs)

Mem[\$rs + offset] = \$rt

Branch

beq \$rs,\$rt,offset

\$pc = (\$rd == \$rs)? (pc+4+offset):(pc+4);

Jump

j address

pc = address

MIPS fixed sized instruction formats



R - Format

| | | | | | |
|----|----|----|----|-------|------|
| op | rs | rt | rd | shamt | func |
|----|----|----|----|-------|------|

ALU

alu \$rd,\$rs,\$rt

I - Format

| | | | |
|----|----|----|-----------------|
| op | rs | rt | value or offset |
|----|----|----|-----------------|

ALUi

alui \$rt,\$rs,value

**Data
Transfer**

**lw \$rt,offset(\$rs)
sw \$rt,offset(\$rs)**

Branch

beq \$rs,\$rt,offset

J - Format

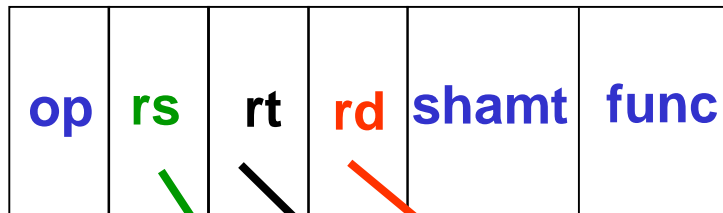
| | |
|----|------------------|
| op | absolute address |
|----|------------------|

Jump

j address

Assembling Instructions

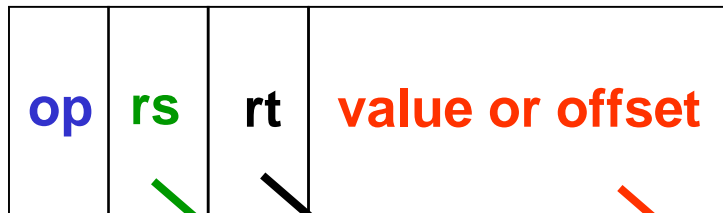
Suppose there are 32 registers, addu opcode=001001, addi op=001000



0x00400020

addu \$23, \$0, \$31

001001:00000:11111:10111:00000:000000



0x00400024

addi \$17, \$0, 5

001000:00000:00101:0000000000000000101

MIPS instruction formats

Arithmetic

addi \$rt, \$rs, value

add \$rd, \$rs, \$rt

Data Transfer

lw \$rt, offset(\$rs)

sw \$rt, offset(\$rs)

Conditional branch

beq \$rs, \$rt, offset

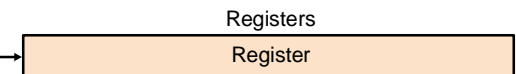
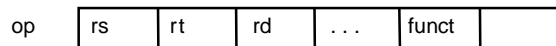
Unconditional jump

j address

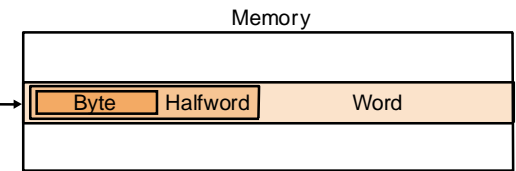
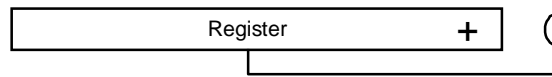
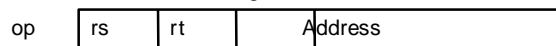
1. Immediate addressing



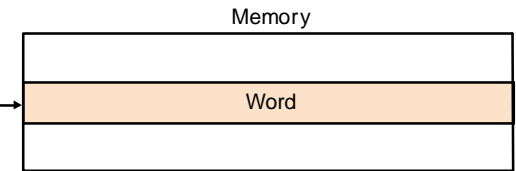
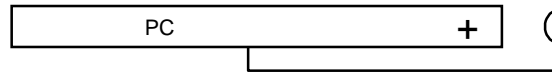
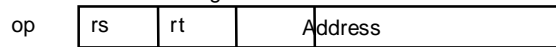
2. Register addressing



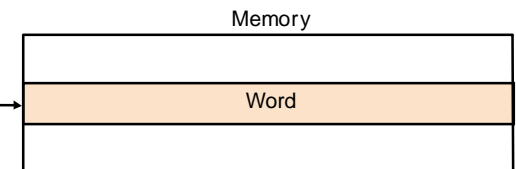
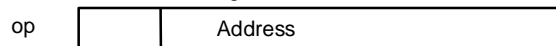
3. Base addressing



4. Relative PC addressing



5. Pseudodirect addressing



MIPS registers and conventions



| <u>Name</u> | <u>Number</u> | <u>Conventional usage</u> |
|-------------|---------------|---|
| \$0 | 0 | Constant 0 |
| \$v0-\$v1 | 2-3 | Expression evaluation & function return |
| \$a0-\$a3 | 4-7 | Arguments 1 to 4 |
| \$t0-\$t9 | 8-15,24,35 | Temporary (not preserved across call) |
| \$s0-\$s7 | 16-23 | Saved Temporary (preserved across call) |
| \$k0-\$k1 | 26-27 | Reserved for OS kernel |
| \$gp | 28 | Pointer to global area |
| \$sp | 29 | Stack pointer |
| \$fp | 30 | Frame pointer |
| \$ra | 31 | Return address (used by function call) |

C function to MIPS Assembly Language

```
int power_2(int y) { /* compute x=2^y; */  
    register int x, i; x=1; i=0; while(i<y) { x=x*2; i=i+1; }  
    return x;  
}
```

Exit condition of a while loop is
if (i >= y) then goto w2

Assembler .s

Comments

```
    addi  $t0, $0, 1    # x=1;  
    addu  $t1, $0, $0   # i=0;  
w1:  bge  $t1,$a0,w2    # while(i<y) { /* bge= greater or equal */  
    addu  $t0, $t0, $t0 # x = x * 2; /* same as x=x+x; */  
    addi  $t1,$t1,1     # i = i + 1;  
    beq   $0,$0,w1      # }  
w2:  addu  $v0,$0,$t0   # return x;  
    jr    $ra           # jump on register ( pc = ra; )
```

Power_2.s: MIPS storage assignment

Byte address, not word address

.text

0x00400020 **addi** \$8, \$0, 1 # addi \$t0, \$0, 1

0x00400024 **addu** \$9, \$0, \$0 # addu \$t1, \$0, \$0

0x00400028 **bge** \$9, \$4, 2 # bge \$t1, \$a0, w2

0x0040002c **addu** \$8, \$8, \$8 # addu \$t0, \$t0, \$t0

0x00400030 **addi** \$9, \$9, 1 # addi \$t1, \$t1, 1

0x00400034 **beq** \$0, \$0, -3 # beq \$0, \$0, w1

0x00400038 **addu** \$2, \$0, \$8 # addu \$v0, \$0, \$t0

0x0040003c **jr** \$31 # jr \$ra

2 words
after pc
fetch

after bge
fetch pc is
0x00400030
plus 2
words is
0x00400038

Machine Language Single Stepping

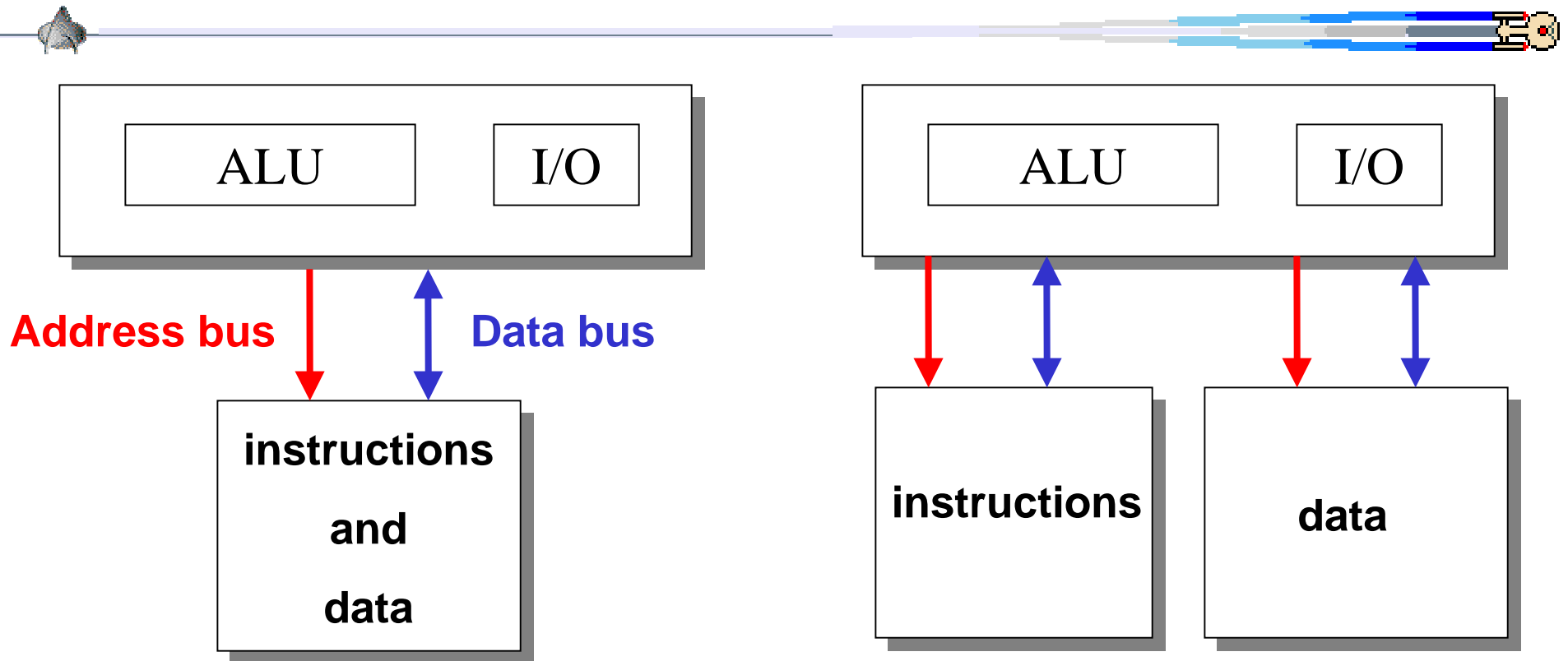


Assume power2(0); is called; then \$a0=0 and \$ra=700018

Values changes after the instruction!

| \$pc | \$v0 | \$a0 | \$t0 | \$t1 | \$ra | |
|-----------------|----------|------|----------|----------|--------|----------------------------|
| | \$2 | \$4 | \$8 | \$9 | \$31 | |
| 00400020 | ? | 0 | ? | ? | 700018 | addi \$t0, \$0, 1 |
| 00400024 | ? | 0 | 1 | ? | 700018 | addu \$t1, \$0, \$0 |
| 00400028 | ? | 0 | 1 | 0 | 700018 | bge \$t1,\$a0,w2 |
| 00400038 | ? | 0 | 1 | 0 | 700018 | add \$v0,\$0,\$t0 |
| 0040003c | 1 | 0 | 1 | 0 | 700018 | jr \$ra |
| 00700018 | ? | 0 | 1 | 0 | 700018 | ... |

Von Neuman & Harvard CPU Architectures



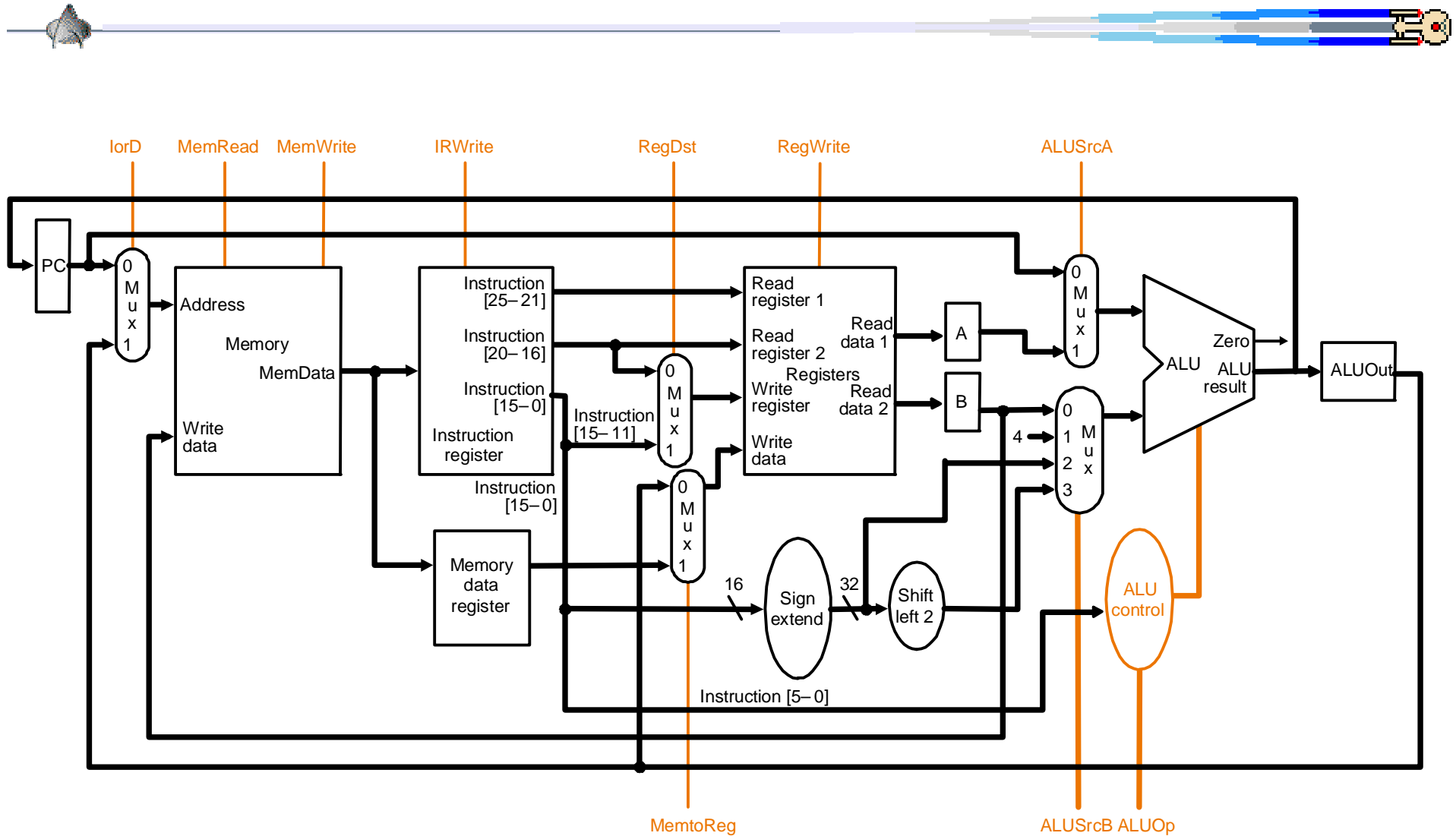
Von Neuman architecture

Area efficient but requires higher bus bandwidth because instructions and data must compete for memory.

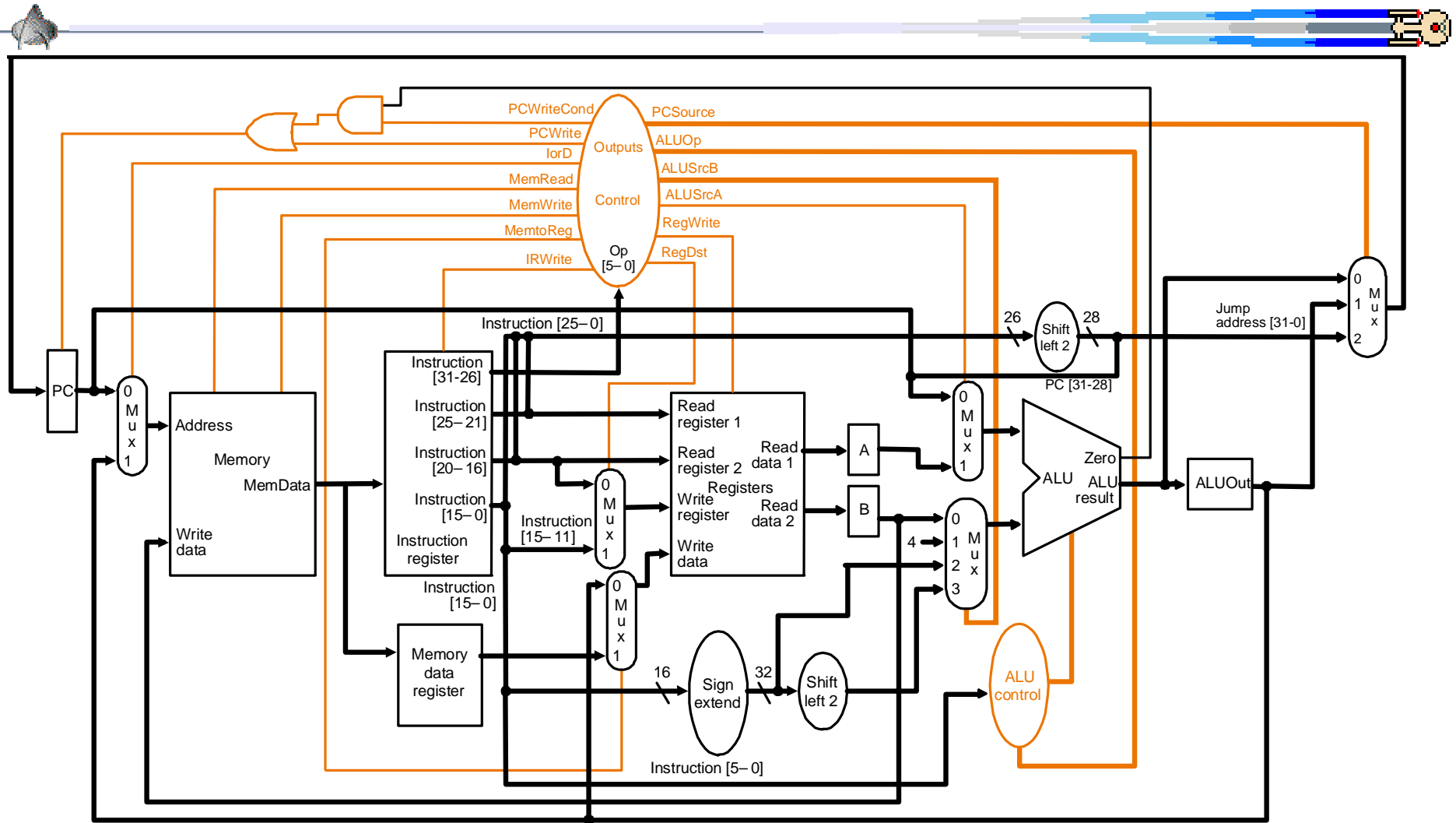
Harvard architecture was

coined to describe machines with separate memories. **Speed efficient:** Increased parallelism.

Multi-cycle Processor Datapath



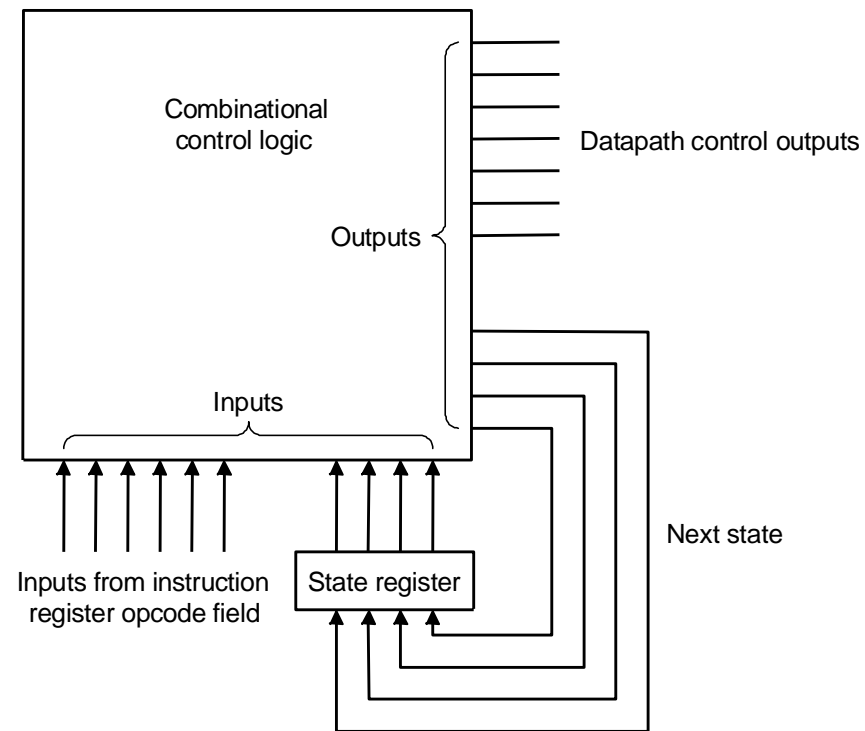
Multi-cycle Datapath: with controller



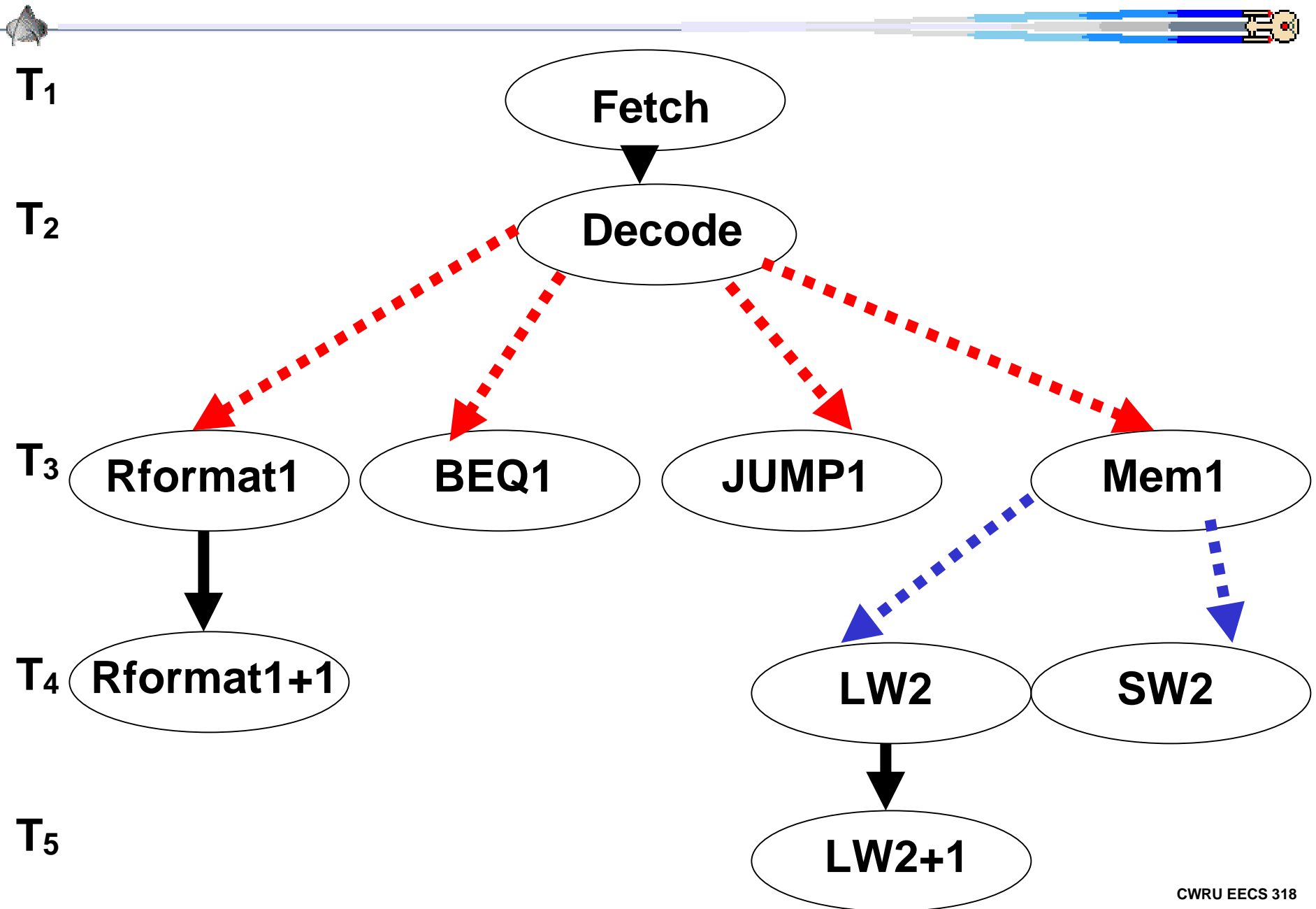
Multi-cycle using Finite State Machine



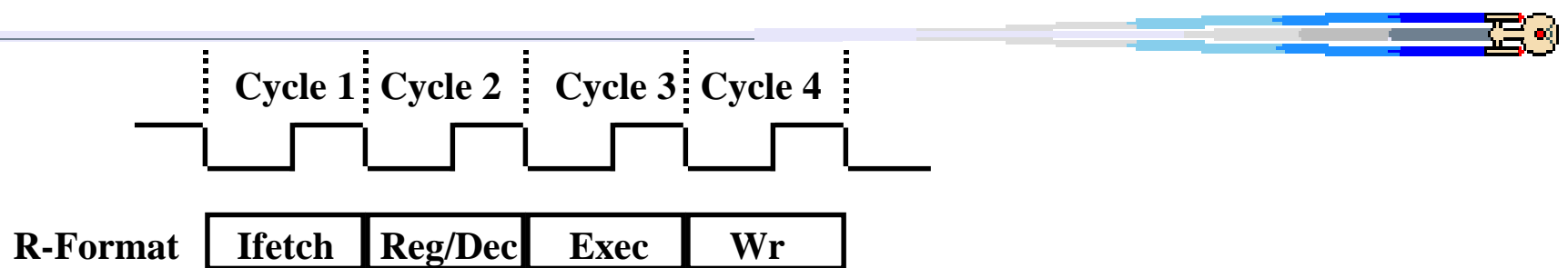
Finite State Machine (*hardwired control*)



Finite State Machine: program overview

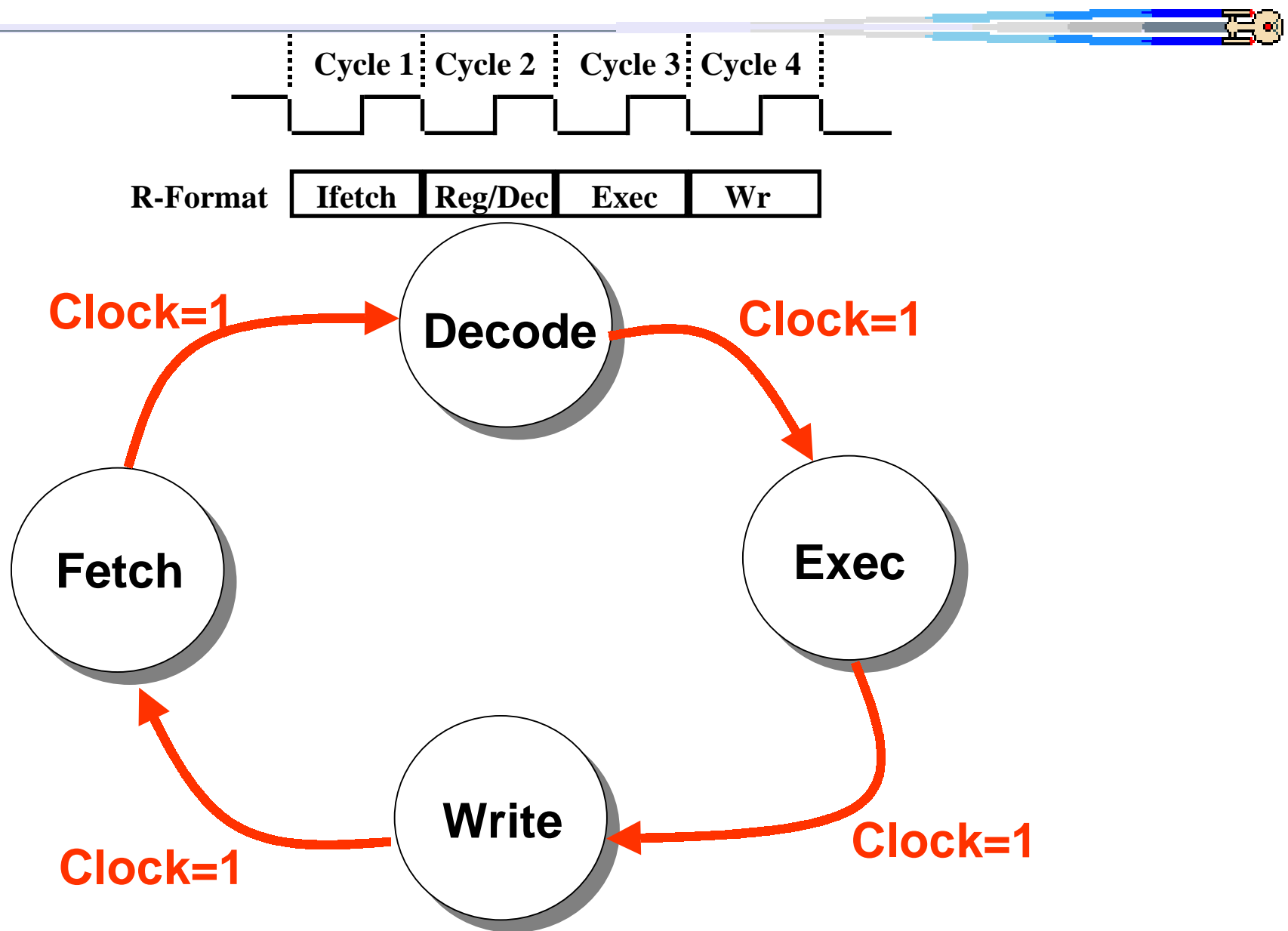


The Four Stages of R-Format

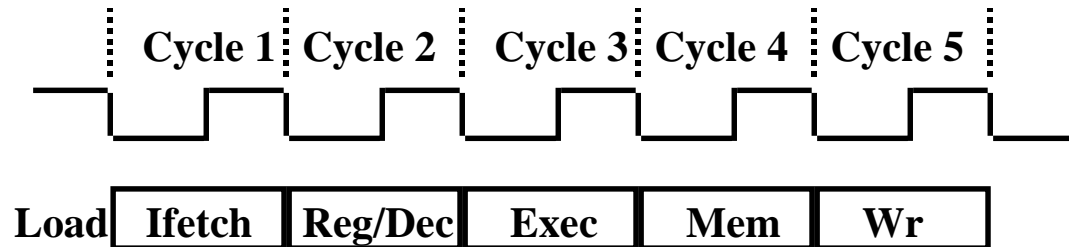


- **Fetch:**
 - Fetch the instruction from the Instruction Memory
- **Decode:**
 - Registers Fetch and Instruction Decode
- **Exec: ALU**
 - ALU operates on the two register operands
 - Update PC
- **Write: Reg**
 - Write the ALU output back to the register file

R-Format State Machine

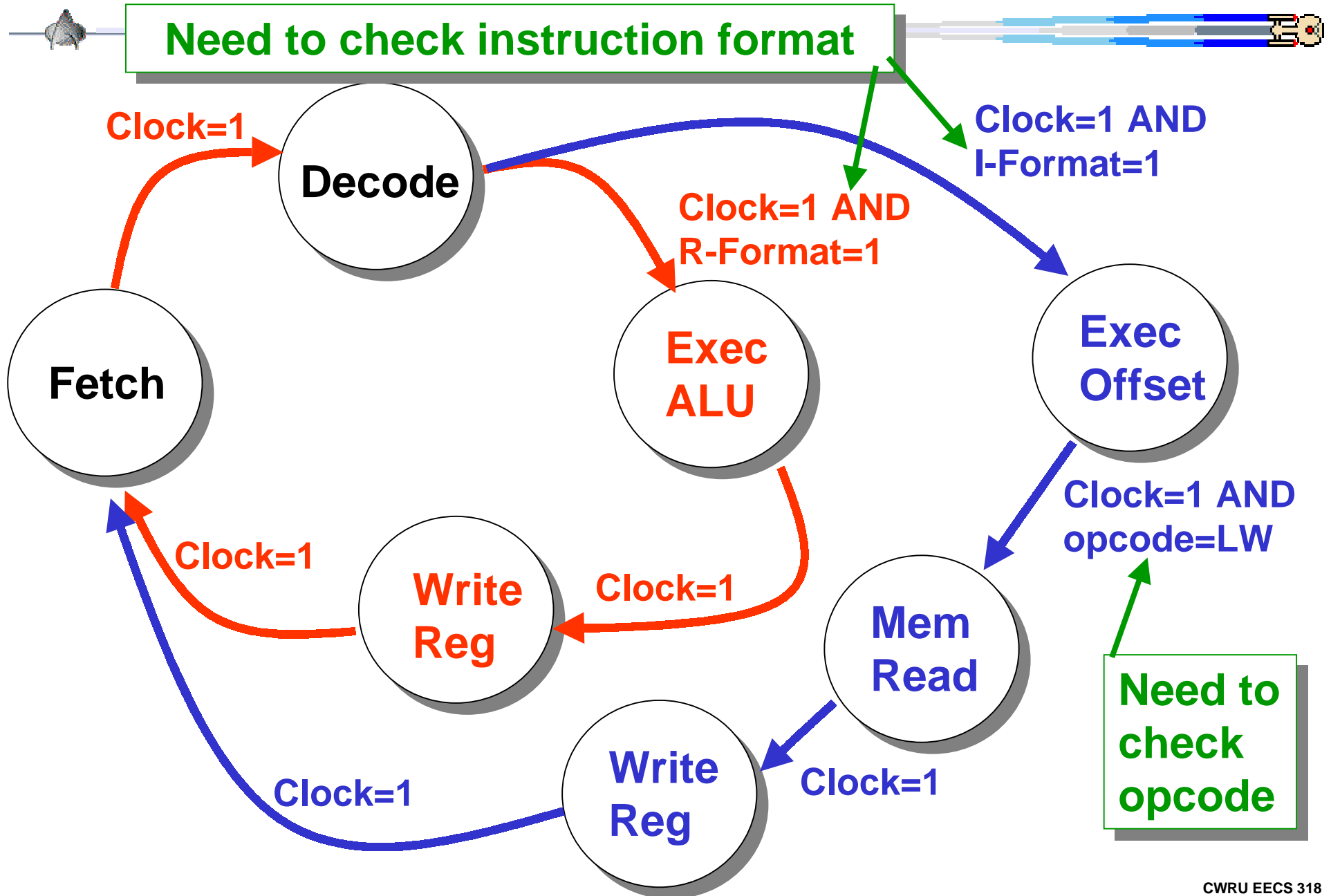


The Five Stages of Load Instruction

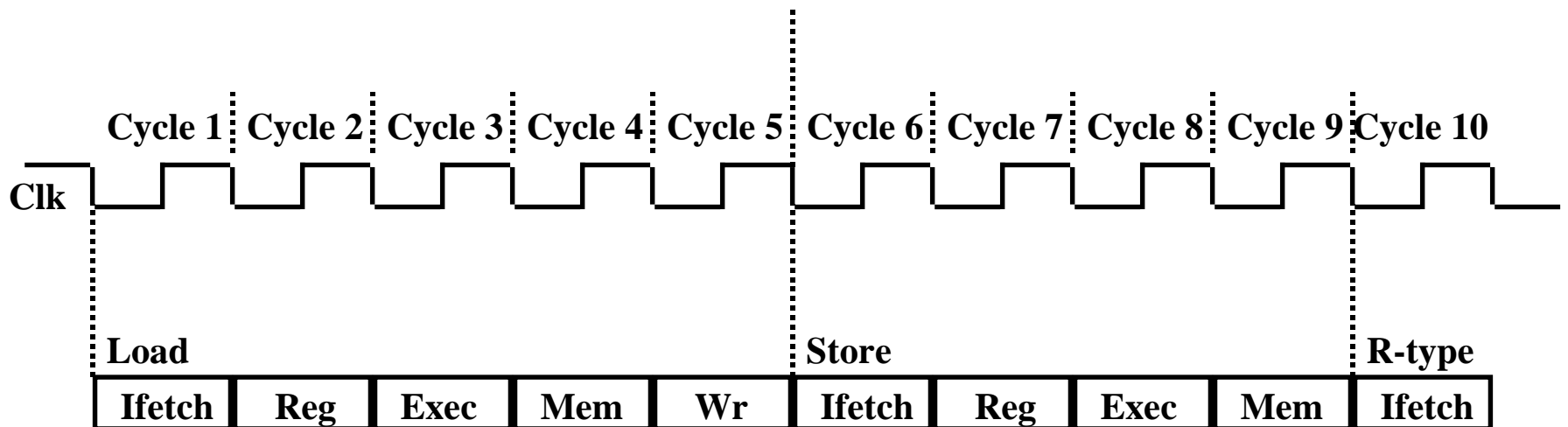


- **Fetch:**
 - Fetch the instruction from the Instruction Memory
- **Decode:**
 - Registers Fetch and Instruction Decode
- **Exec: Offset**
 - Calculate the memory offset
- **Mem:**
 - Read the data from the Data Memory
- **Wr:**
 - Write the data back to the register file

R-Format & I-Format State Machine

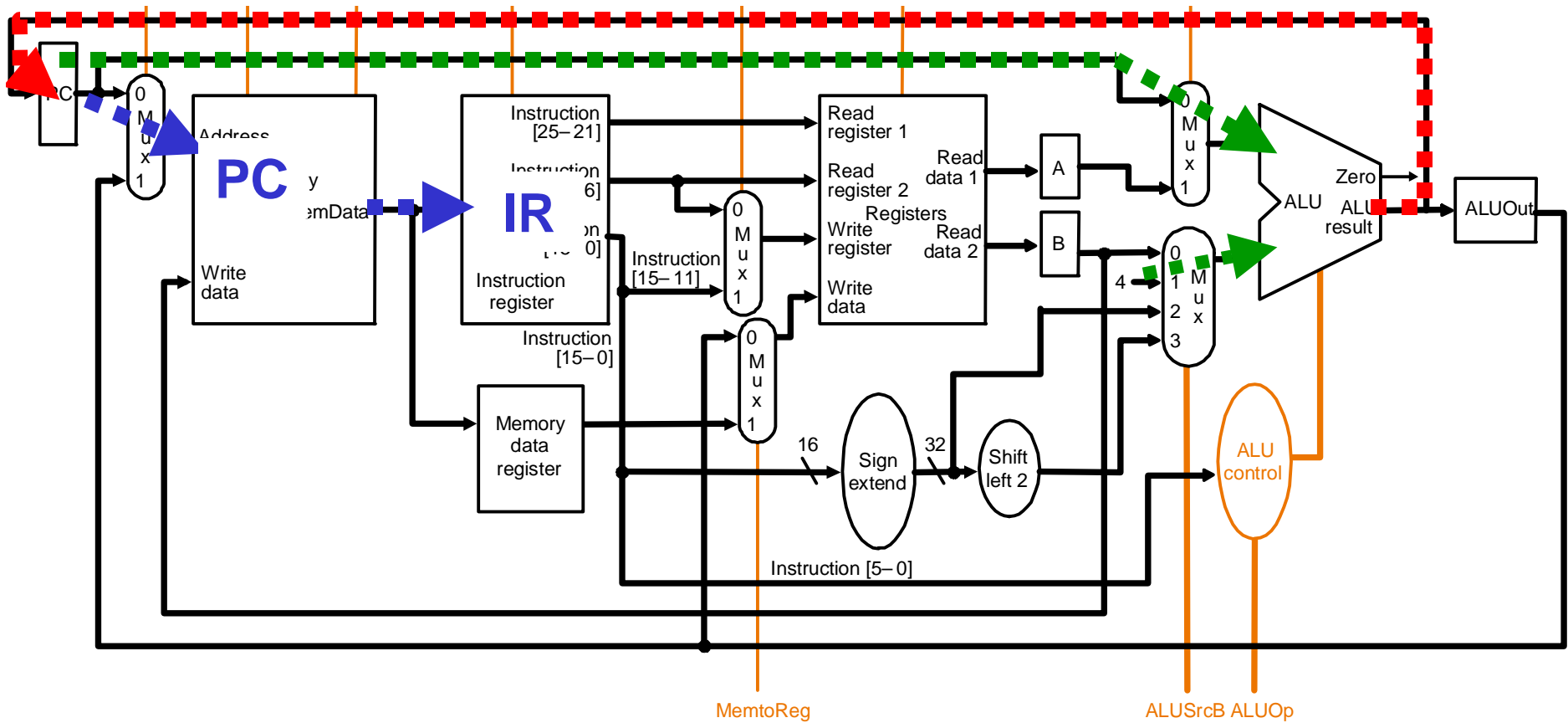


Multi-Instruction sequence

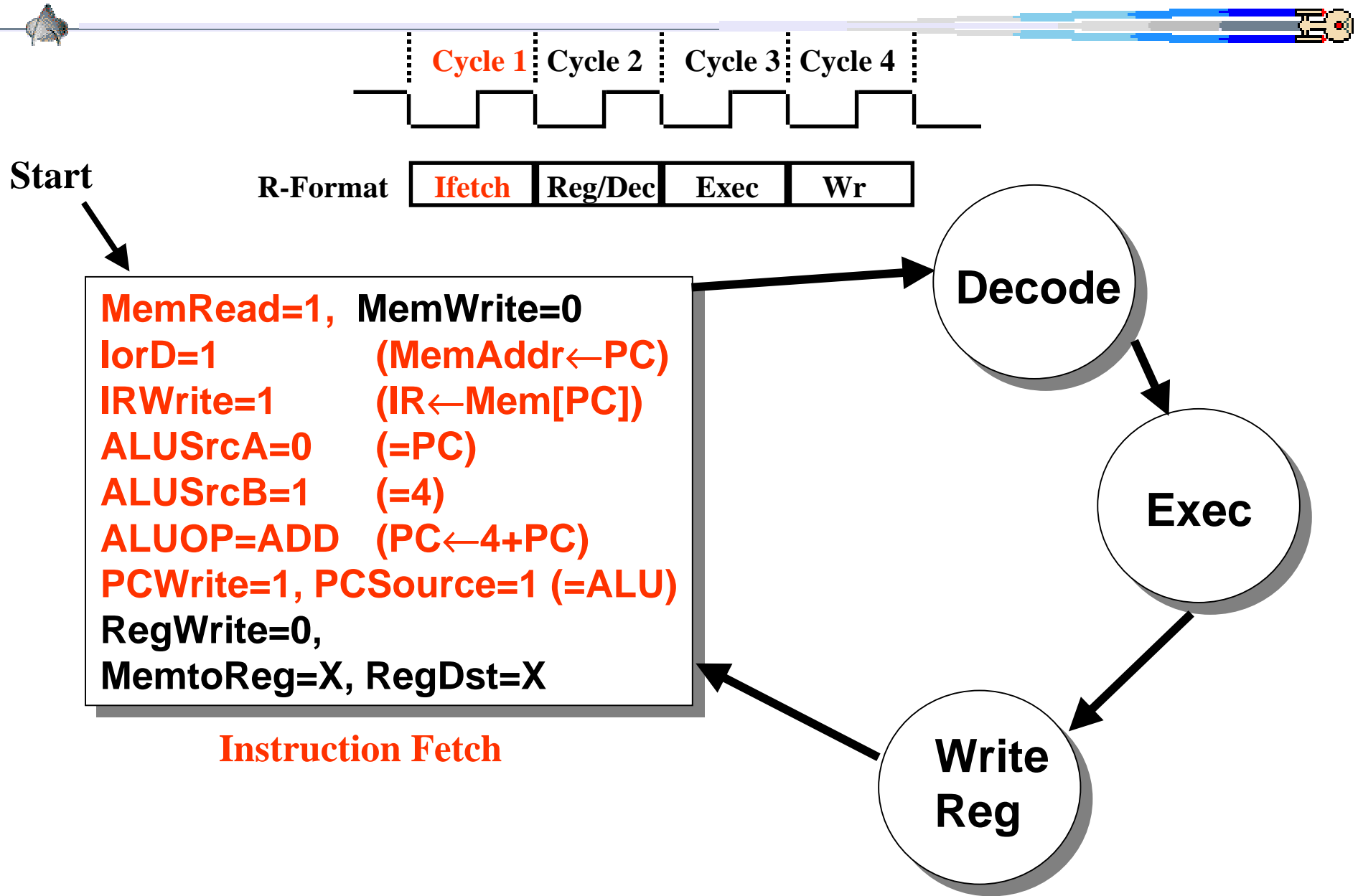


State machine stepping: T_1 Fetch

(Done in parallel) $IR \leftarrow \text{MEMORY}[PC]$ & $PC \leftarrow PC + 4$



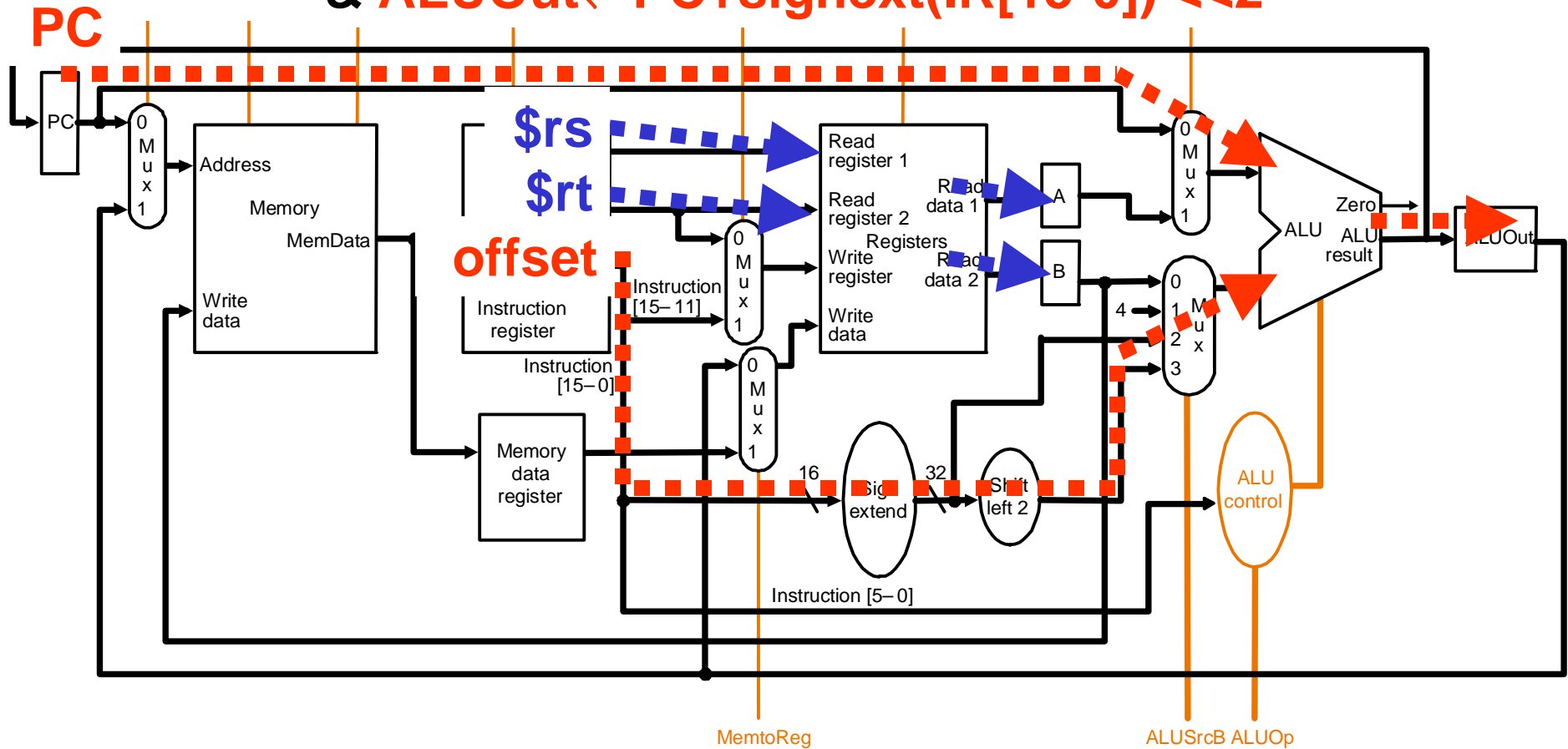
T₁ Fetch: State machine



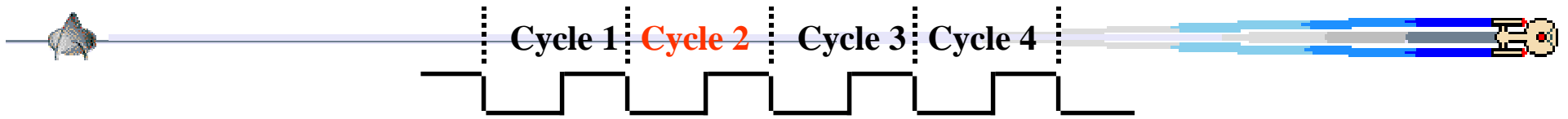
T₂ Decode (read \$rs and \$rt and offset+pc)

$A \leftarrow \text{Reg}[\text{IR}[25-21]]$ & $B \leftarrow \text{Reg}[\text{IR}[20-16]]$

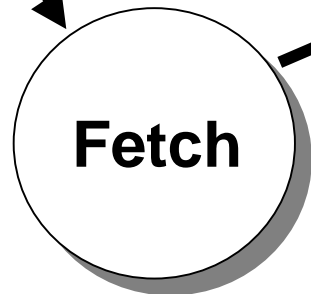
& $\text{ALUOut} \leftarrow \text{PC} + \text{signext}(\text{IR}[15-0]) \ll 2$



T₂ Decode State machine

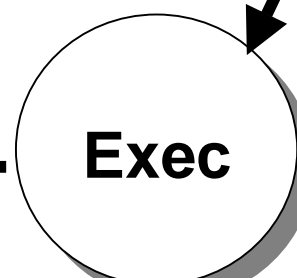
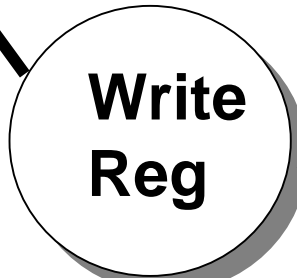


Start



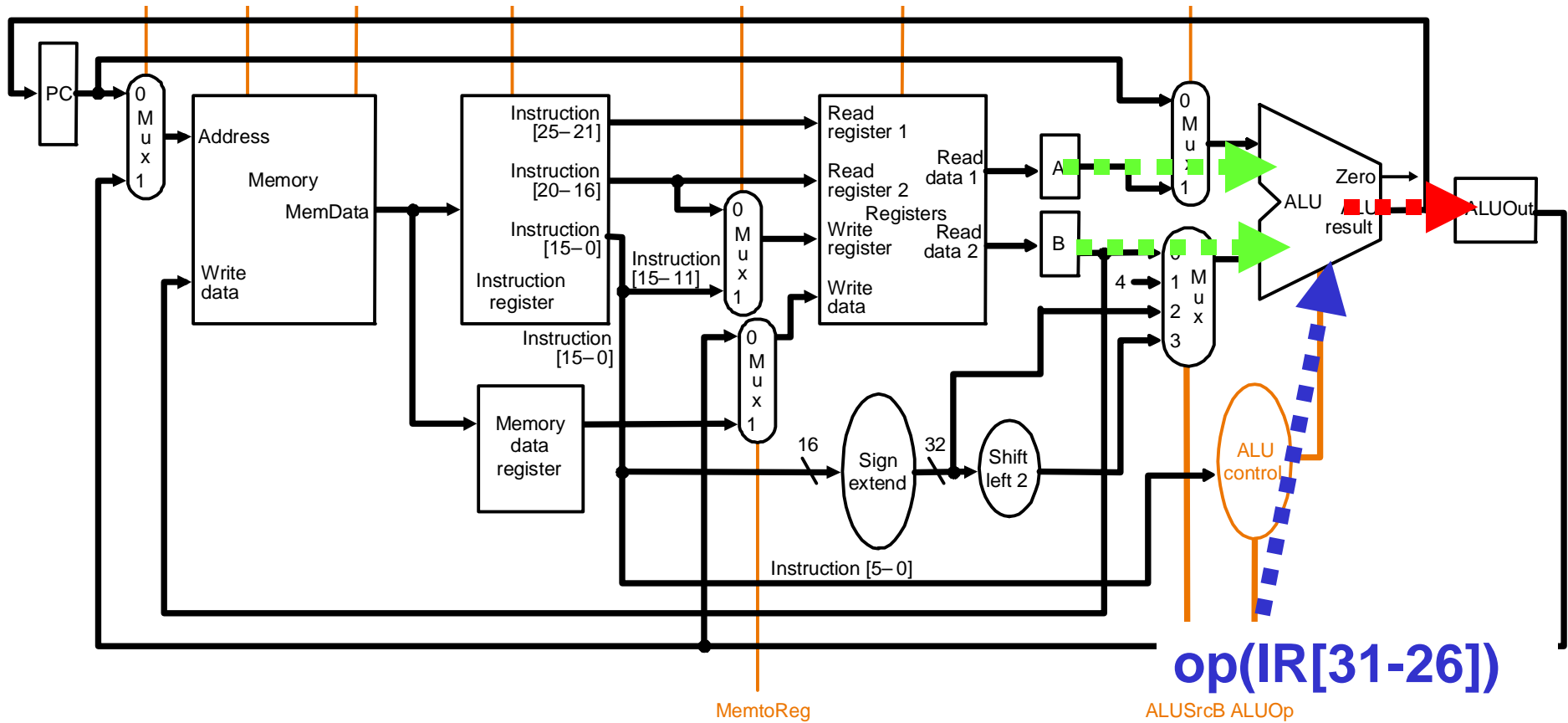
MemRead=0, MemWrite=0
lorD=X
IRWrite=0
ALUSrcA=0 (=PC)
ALUSrcB=3 (=signext(IR<<2))
ALUOP=0 (=add)
PCWrite=0, PCSource=X
RegWrite=0,
MemtoReg=X, RegDst=X

Instr. Decode & Register Fetch

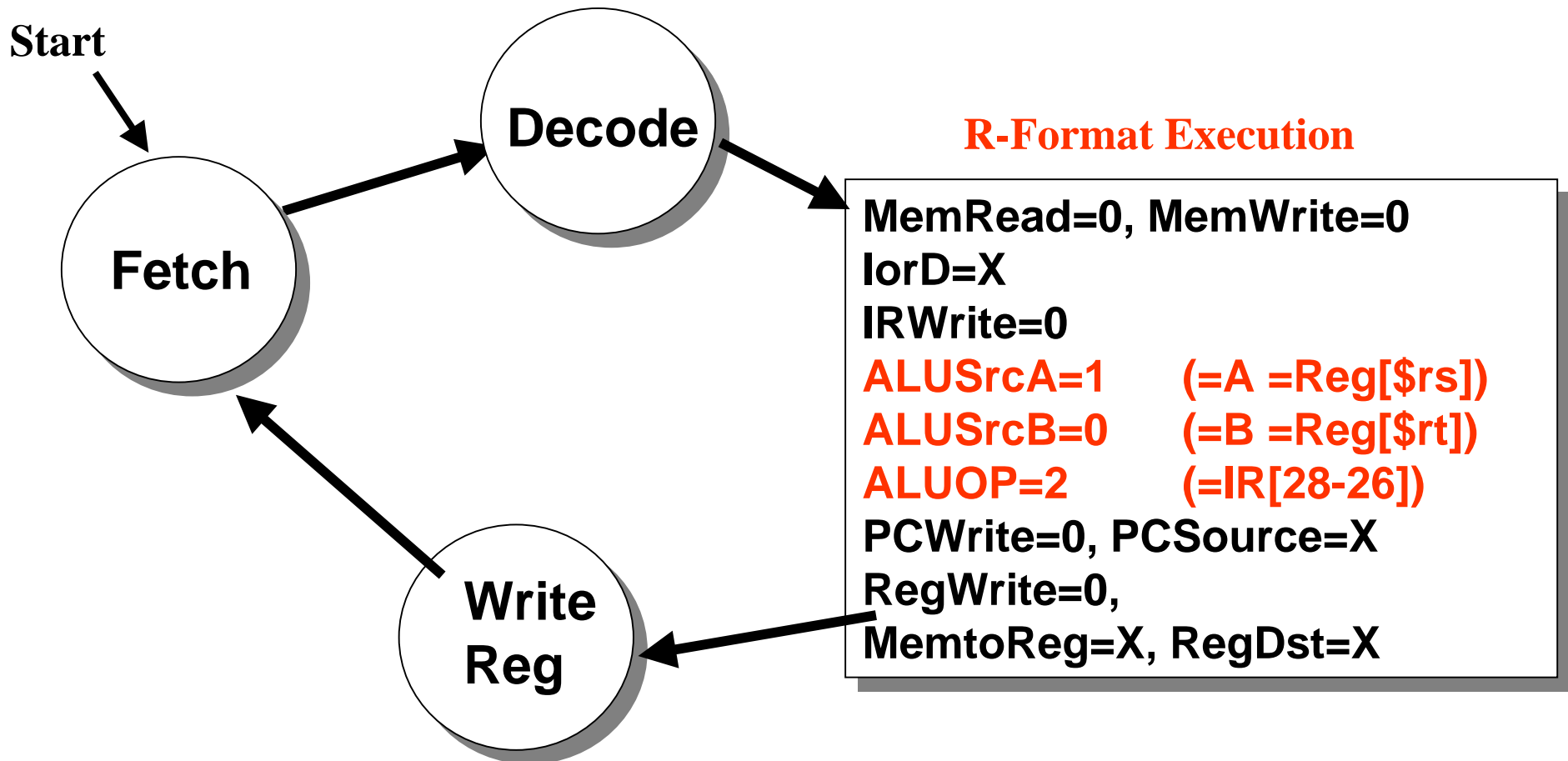
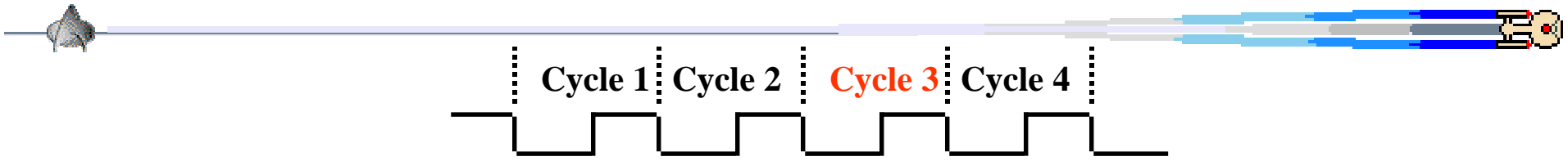


T₃ ExecALU (ALU instruction)

$$\text{ALUOut} \leftarrow A \text{ op}(\text{IR}[31-26]) B$$

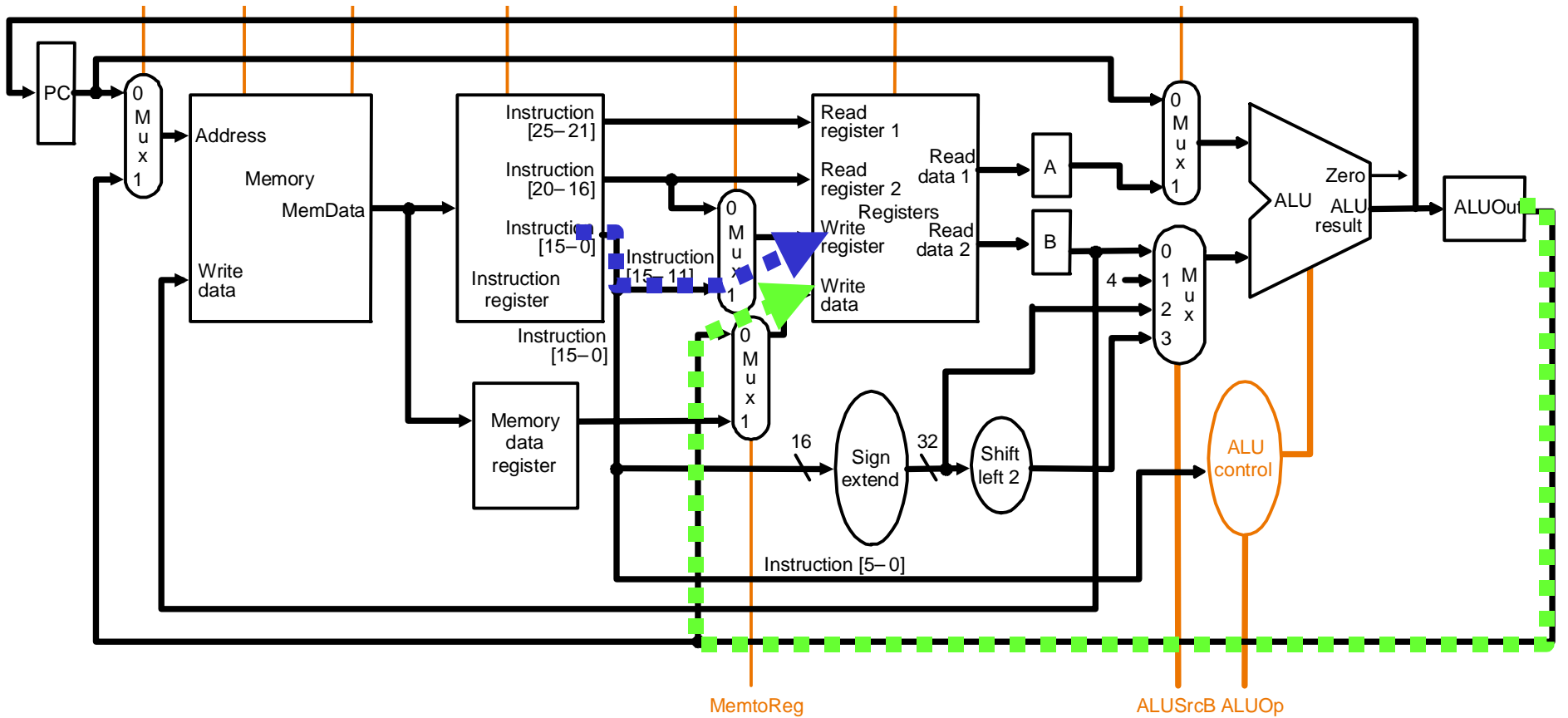


T₃ ExecALU State machine

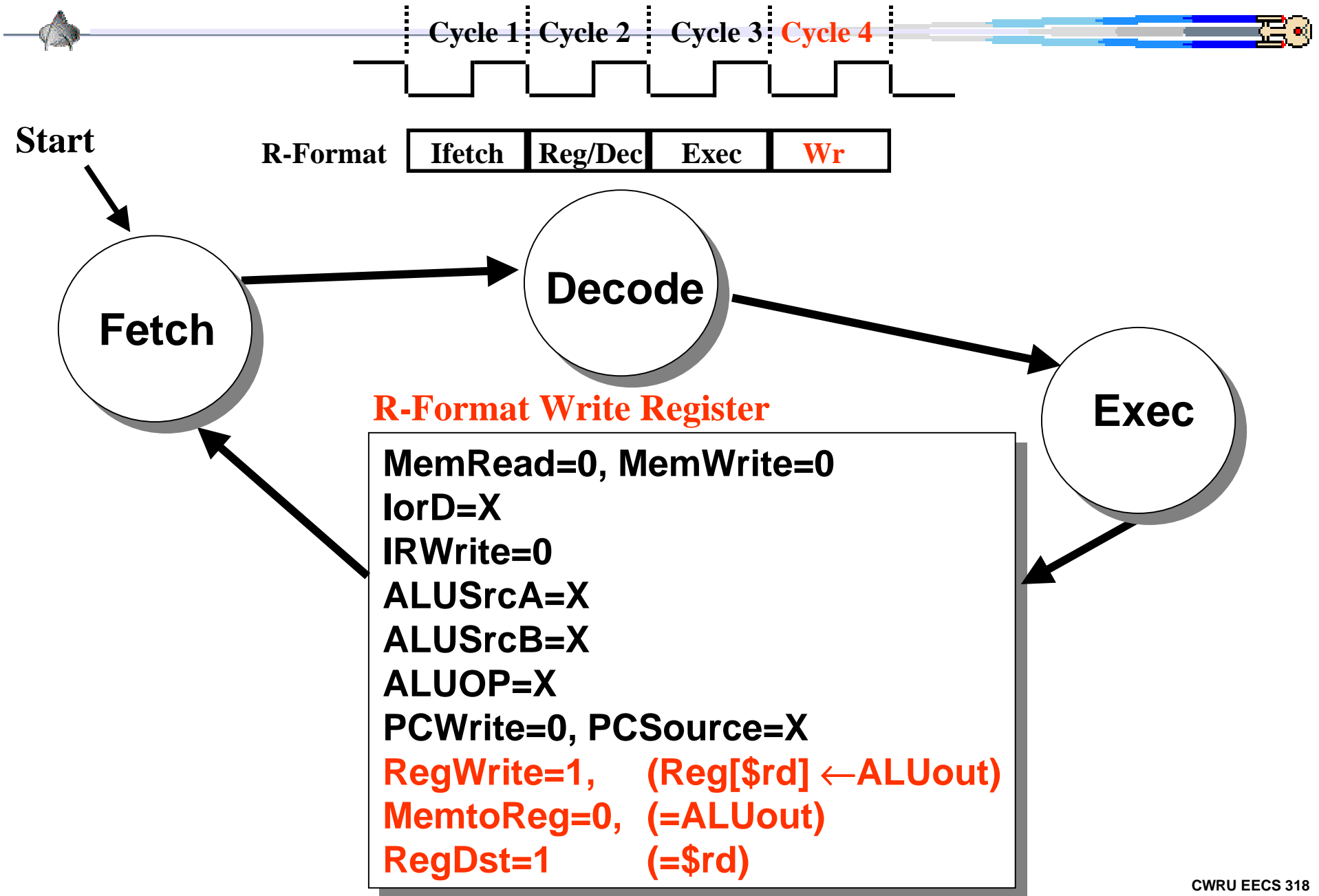


T₄ WrReg (ALU instruction)

Reg[IR[15-11]] ← ALUOut

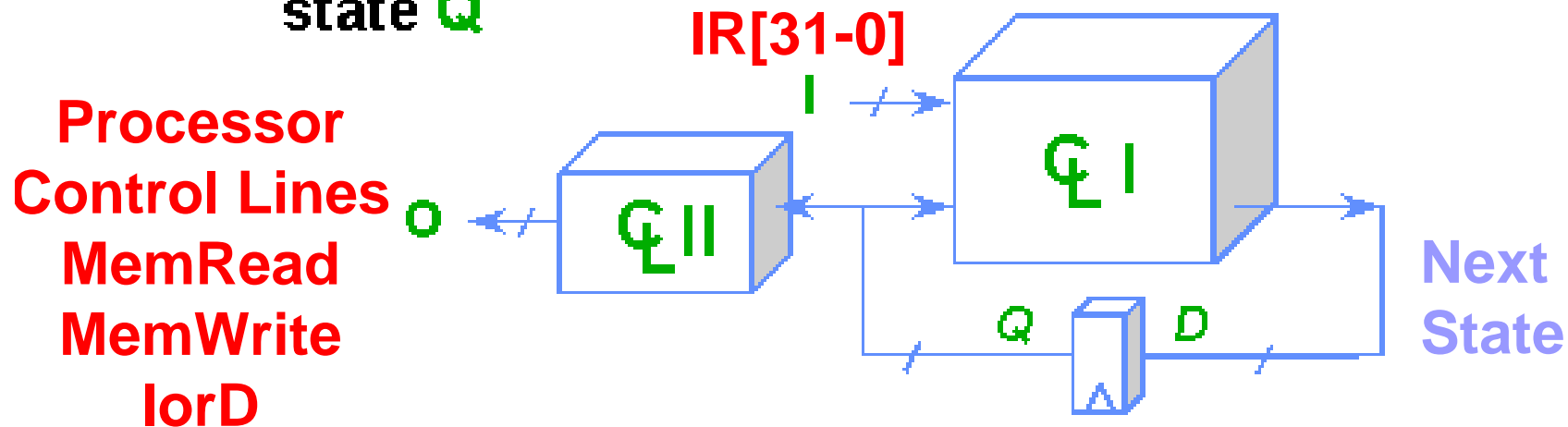


T₄ WrReg State machine



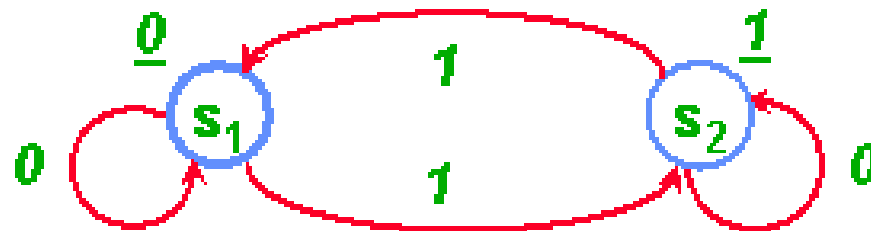
Moore Machines

- So far we considered Moore machines where the output O is a function of only the current state Q



- Moore FSM State Transition Graph

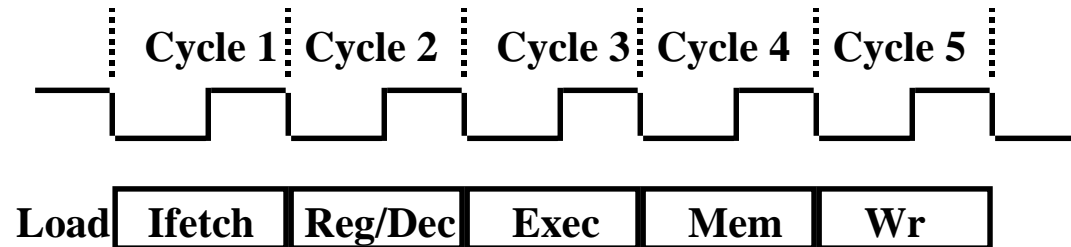
...



Moore Output State Tables: O(State)

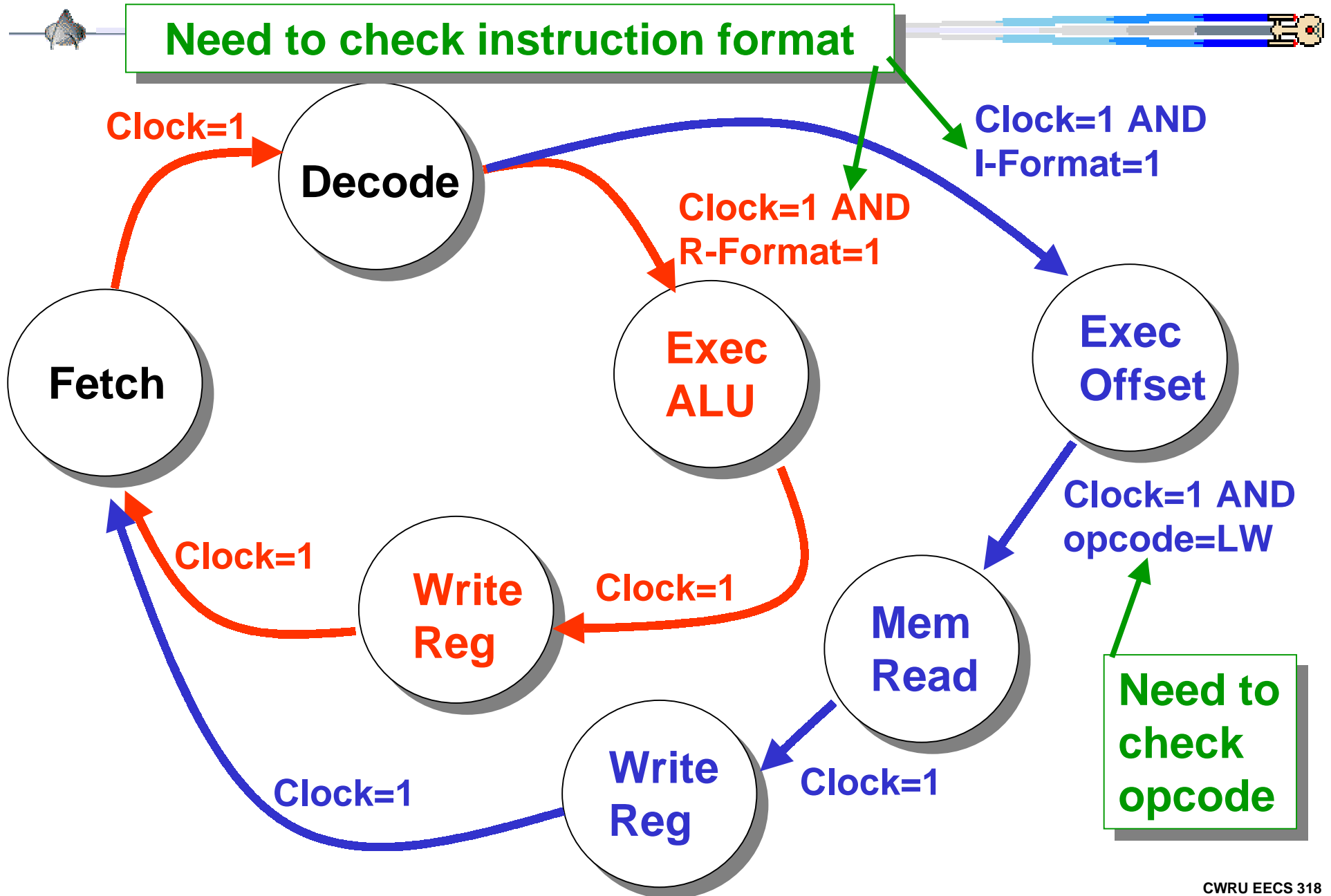
| State | T ₁ | T ₂ | T _{3-R} | T _{4-R} |
|--------------|----------------|----------------|------------------|------------------|
| MemRead | 1 | 0 | 0 | 0 |
| MemWrite | 0 | 0 | 0 | 0 |
| MUX IorD | 0 =PC | X | X | X |
| IRWrite | 1 | 0 | 0 | 0 |
| ALUOP | 0 + | 0 + | 2 =op | X |
| MUX ALUSrcA | 0 =PC | 0 =PC | 1 =A =\$rs | X |
| MUX ALUSrcB | 1 =4 | 3 =offset | 0 =B =\$rt | X |
| PCWrite | 1 | 0 | 0 | 0 |
| MUX PCSource | 0 =ALU | X | X | X |
| RegWrite | 0 | 0 | 0 | 1 |
| MUX MemtoReg | X | X | X | 0 =ALUOut |
| MUX RegDst | X | X | X | 1 =\$rd |

Review: The Five Stages of Load Instruction



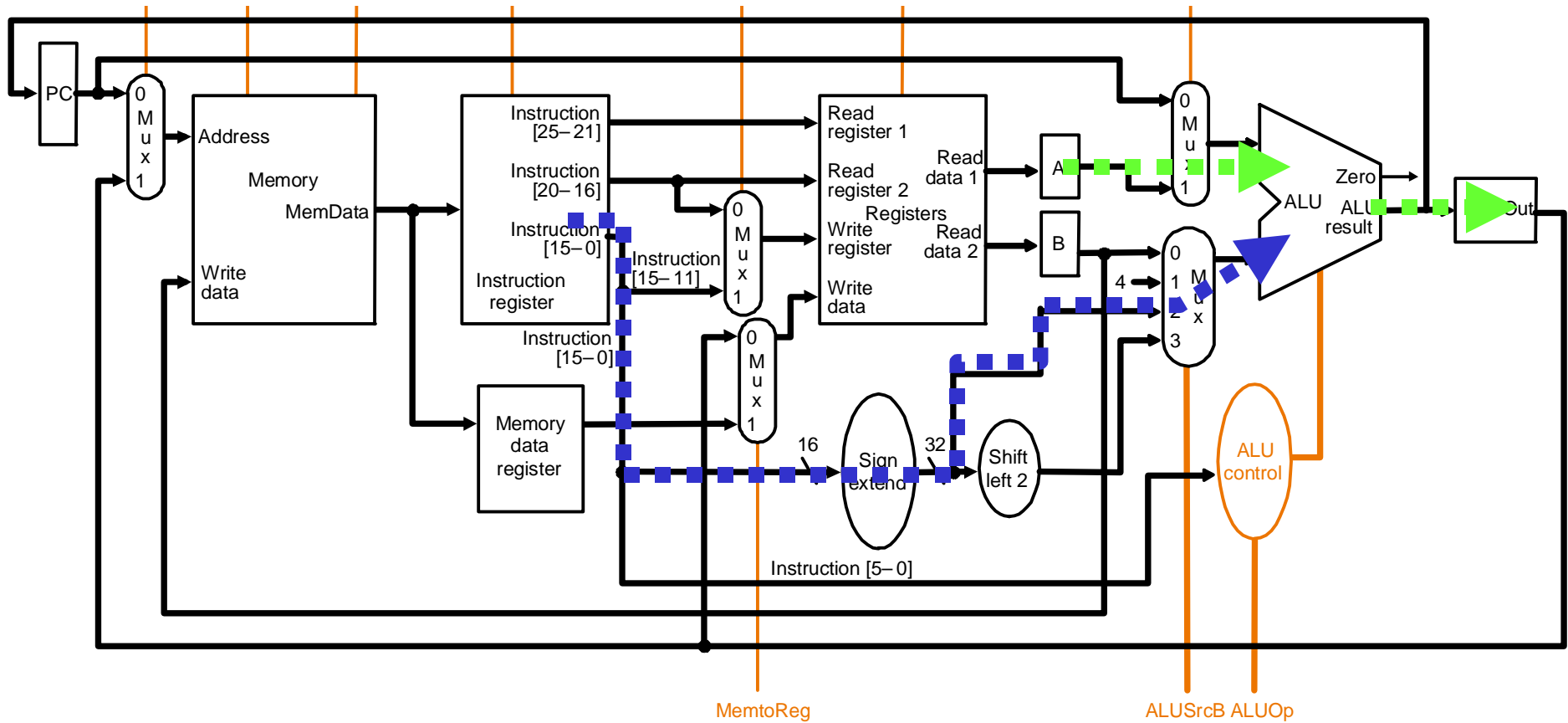
- **Fetch:**
 - Fetch the instruction from the Instruction Memory
- **Decode:**
 - Registers Fetch and Instruction Decode
- **Exec: Offset**
 - Calculate the memory offset
- **Mem:**
 - Read the data from the Data Memory
- **Wr:**
 - Write the data back to the register file

Review: R-Format & I-Format State Machine

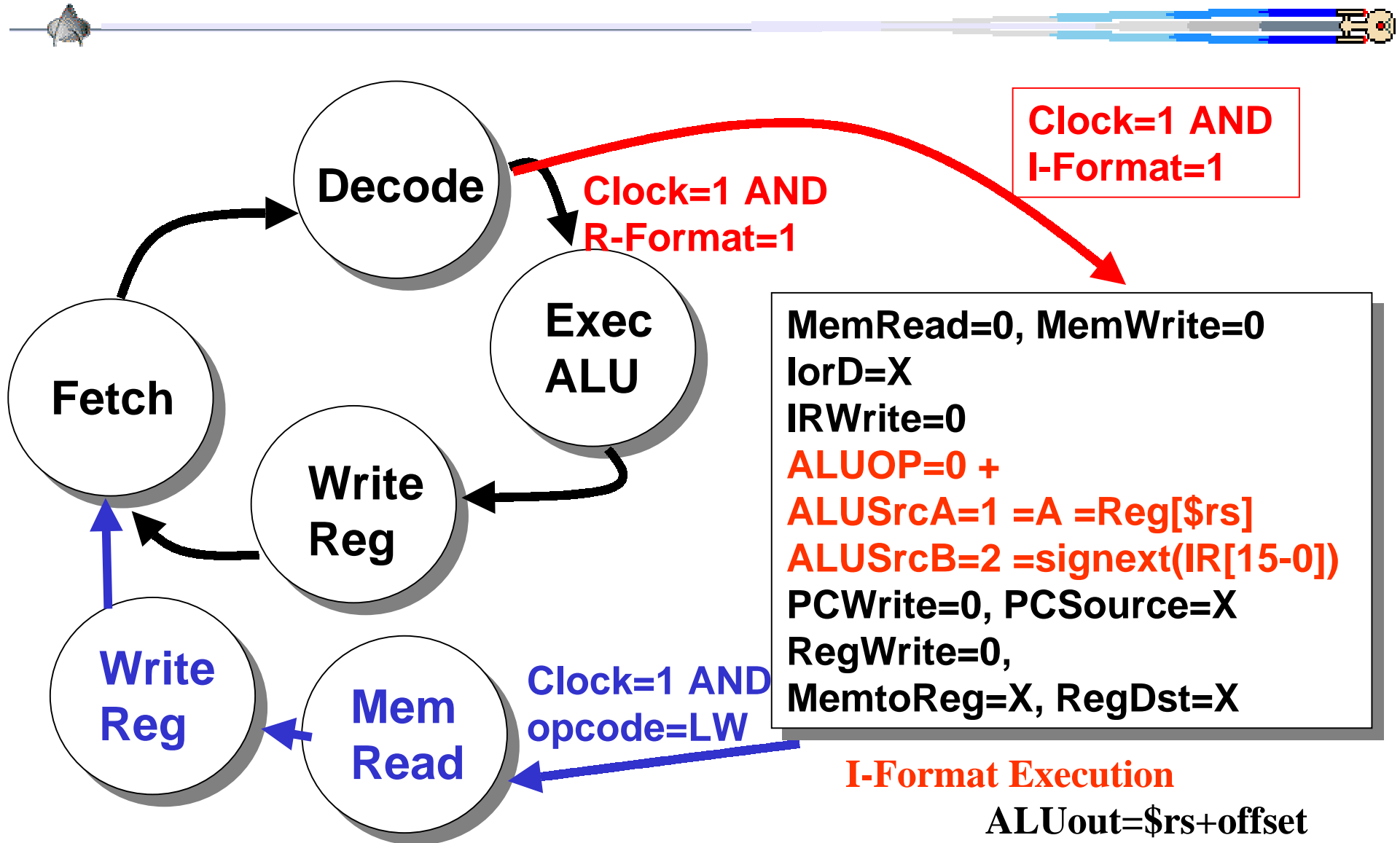


T₃-I Mem1 (common to both load & store)

$$\text{ALUOut} \leftarrow A + \text{sign_extend}(\text{IR}[15-0])$$

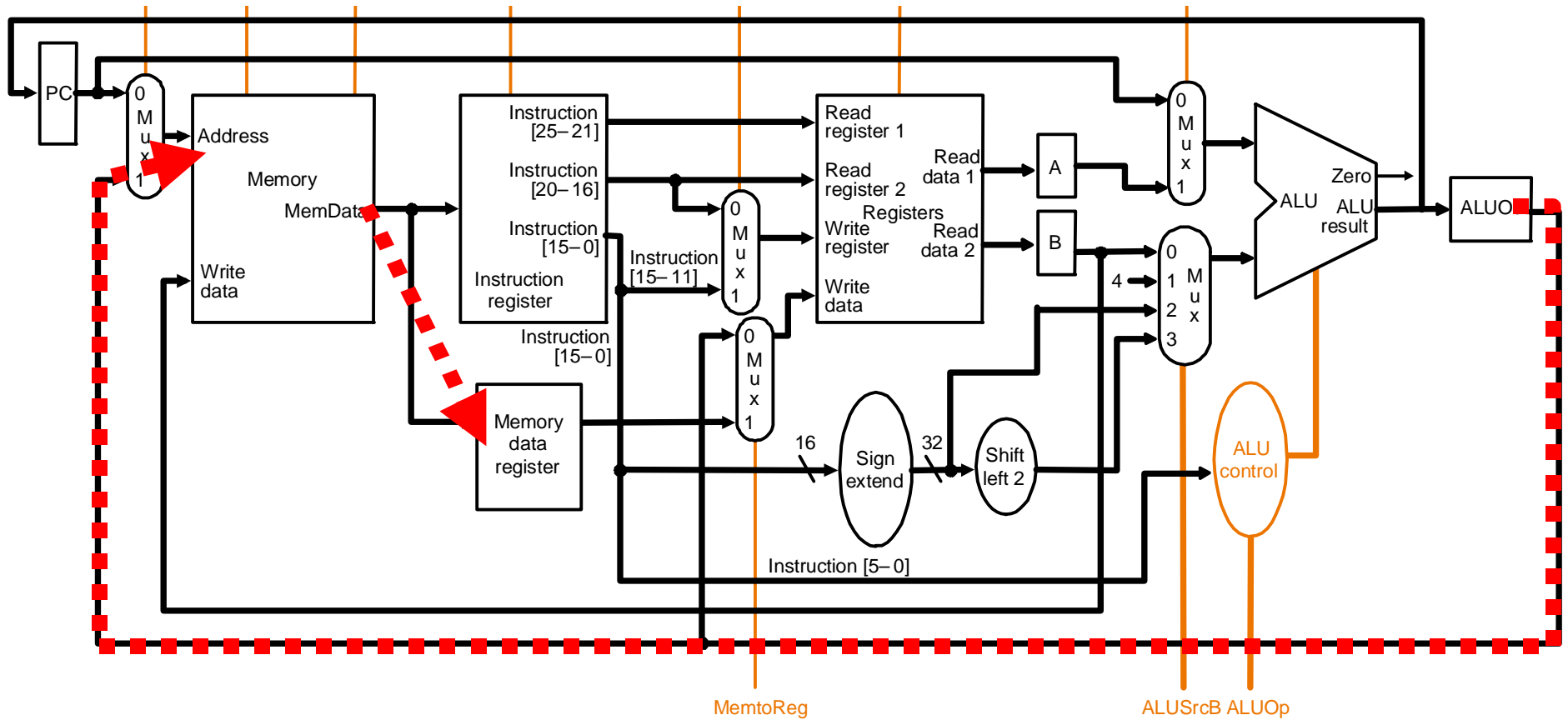


T₃ Mem1 I-Format State Machine = \$rs + offset

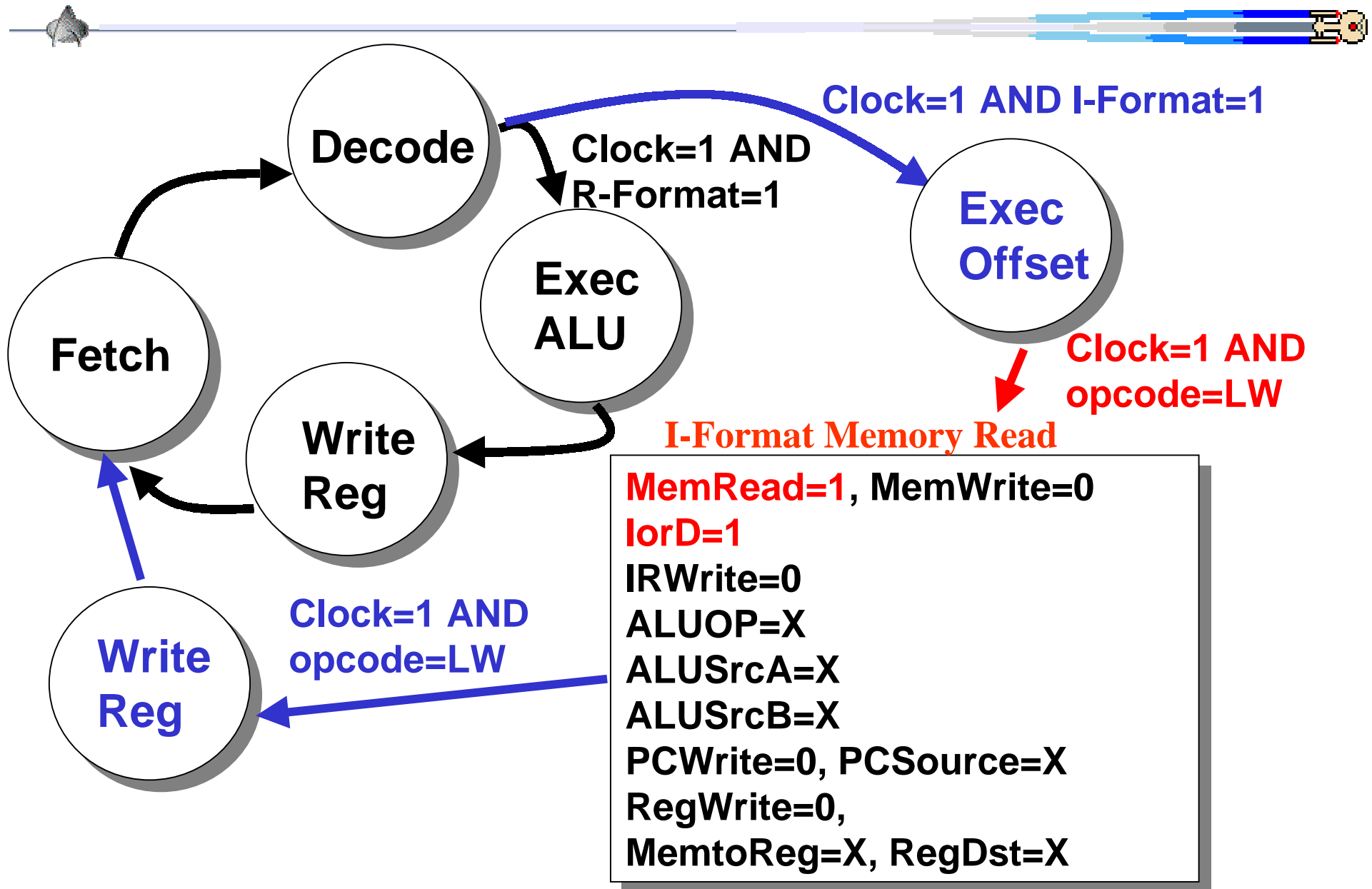


T₄ - LW1 : load instruction, read memory

MDR ← Memory[ALUOut]

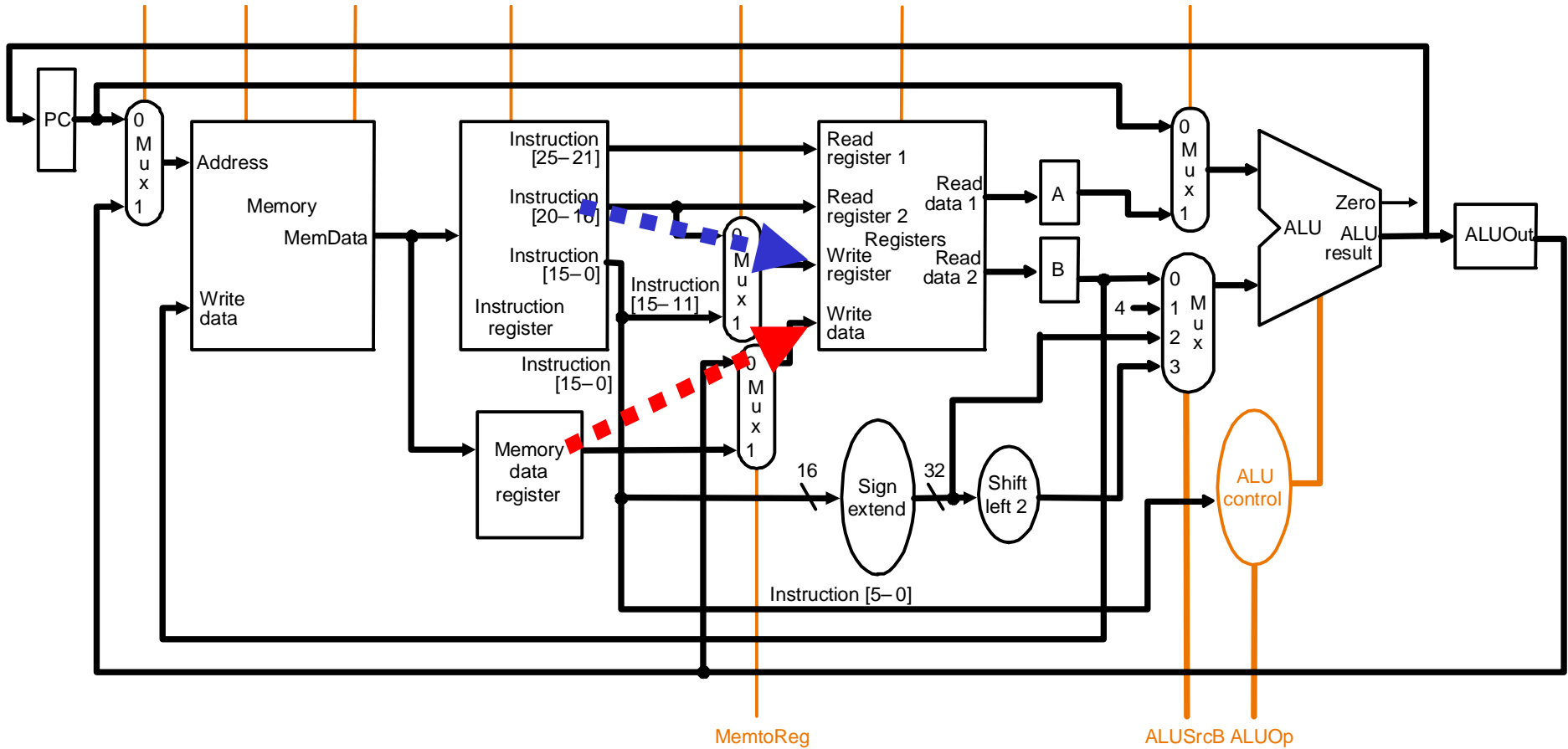


T₄ LW2 I-Format State Machine =Mem[ALU]

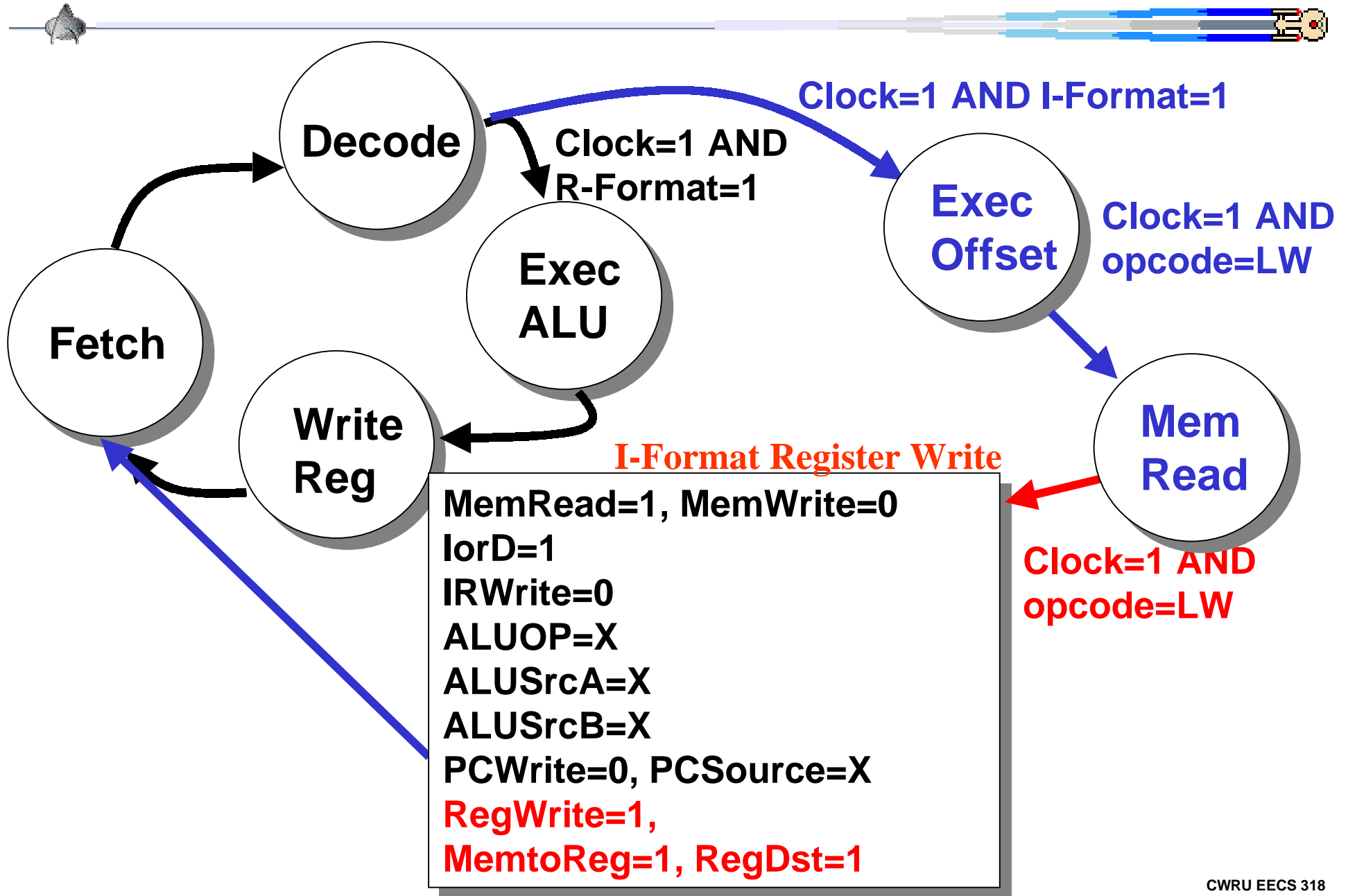


T₅ – LW2 Load instruction, write to register

Reg[IR[20-16]] ← MDR

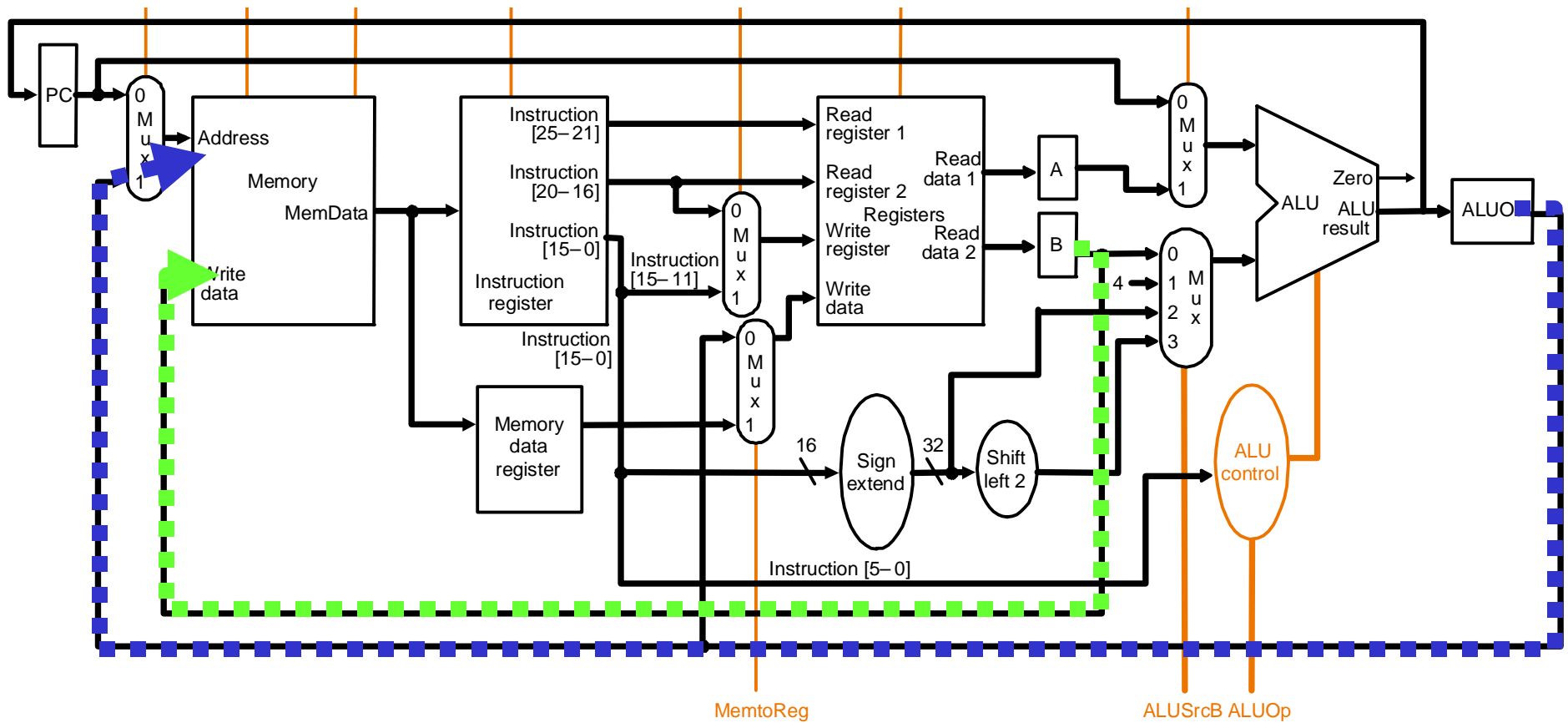


T₅ LW2 I-Format State Machine \$rt=MDR

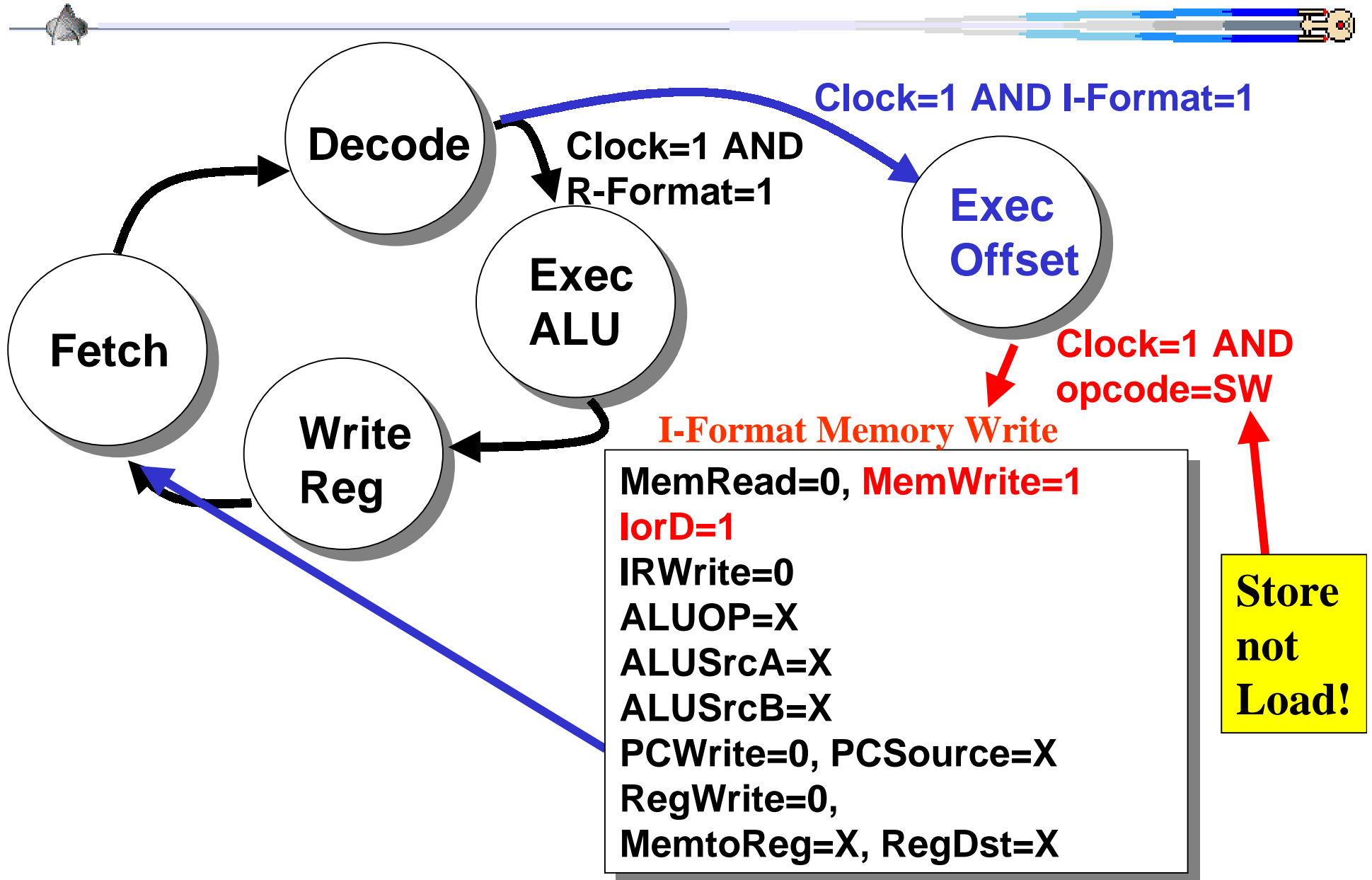


T₄-SW2 Store instruction, write to memory

Memory[ALUOut] ← B

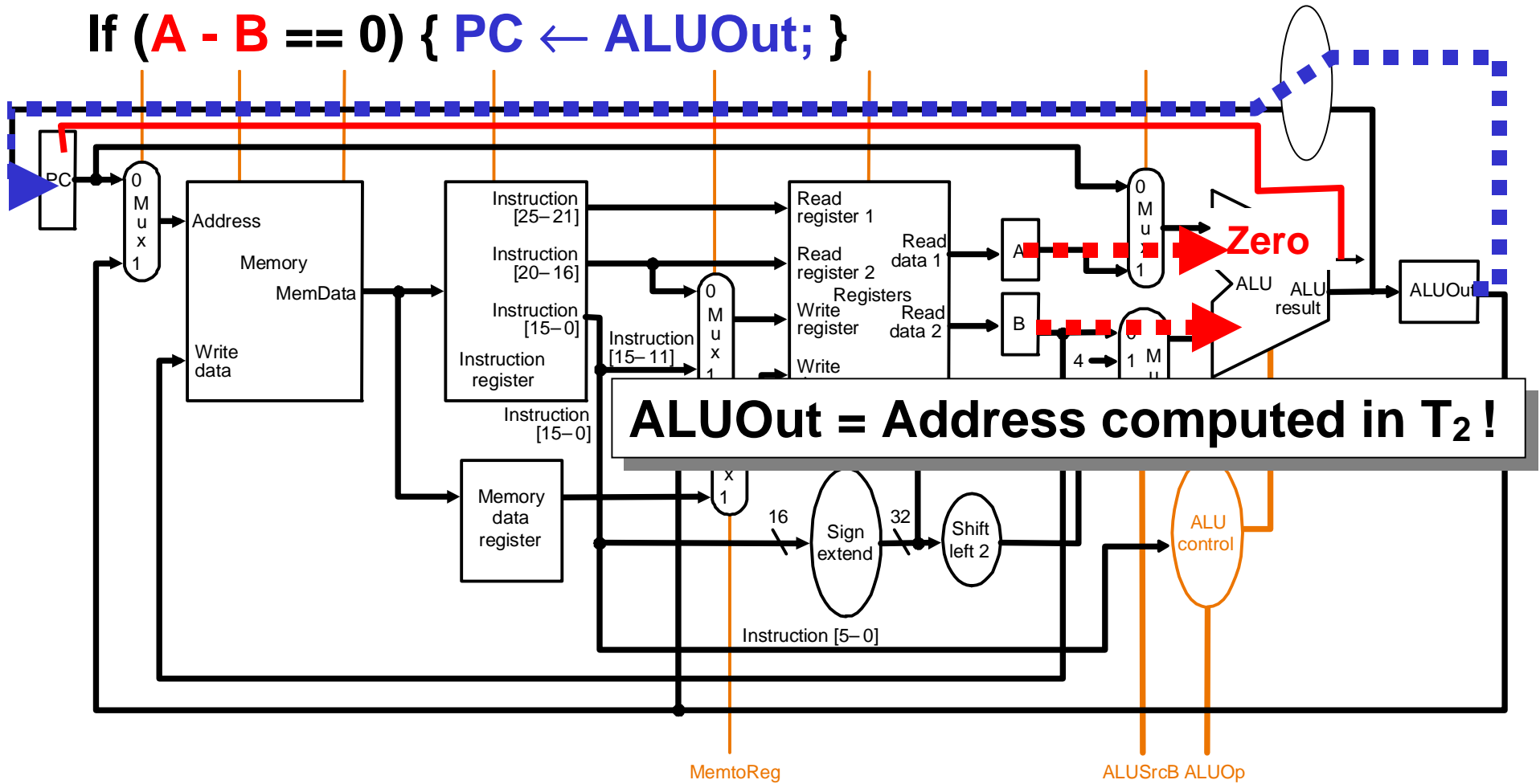


T₄ SW2 I-Format State Machine Mem[ALU]=

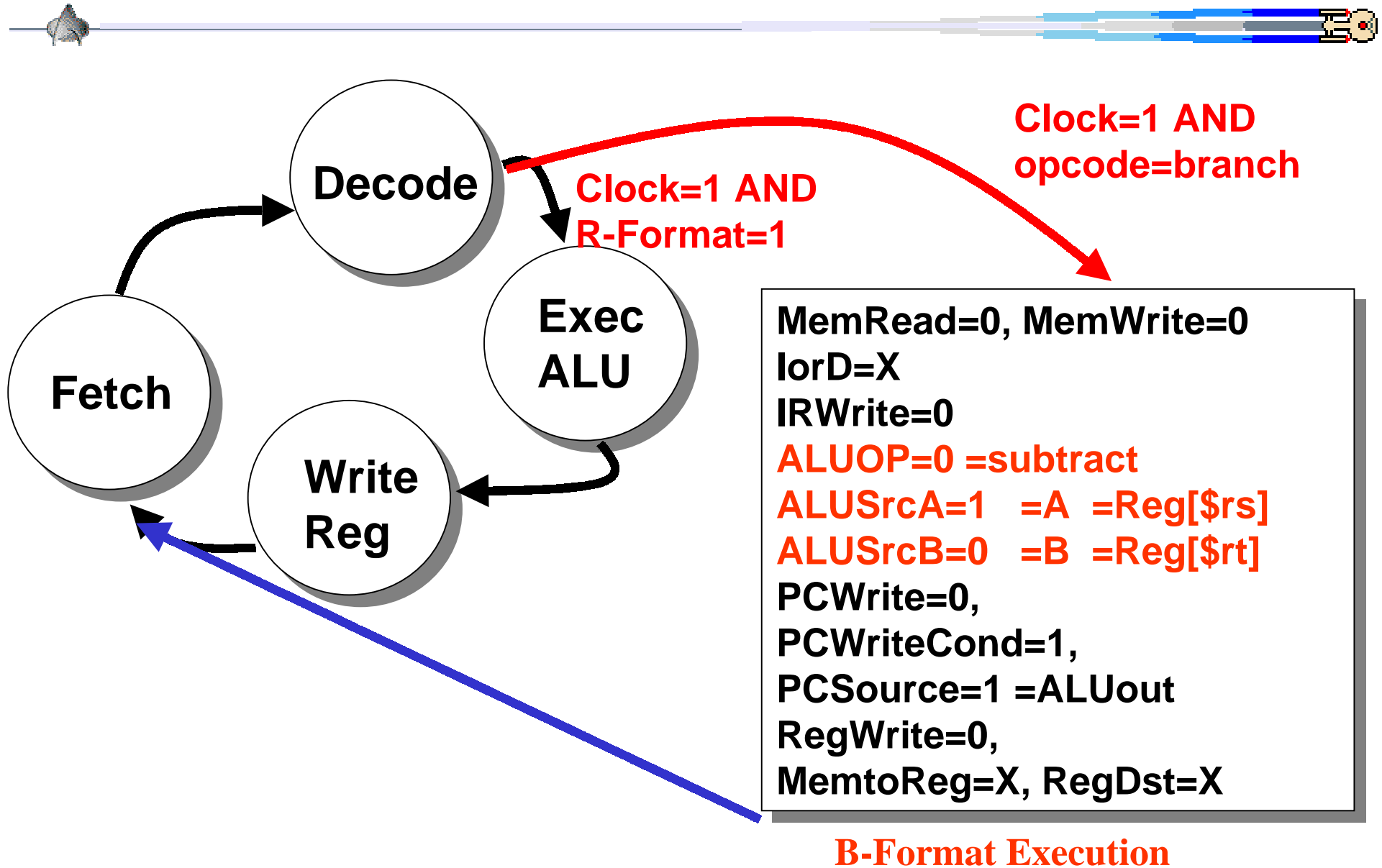


T₃ BEQ1 (Conditional branch instruction)

If (A - B == 0) { PC ← ALUOut; }

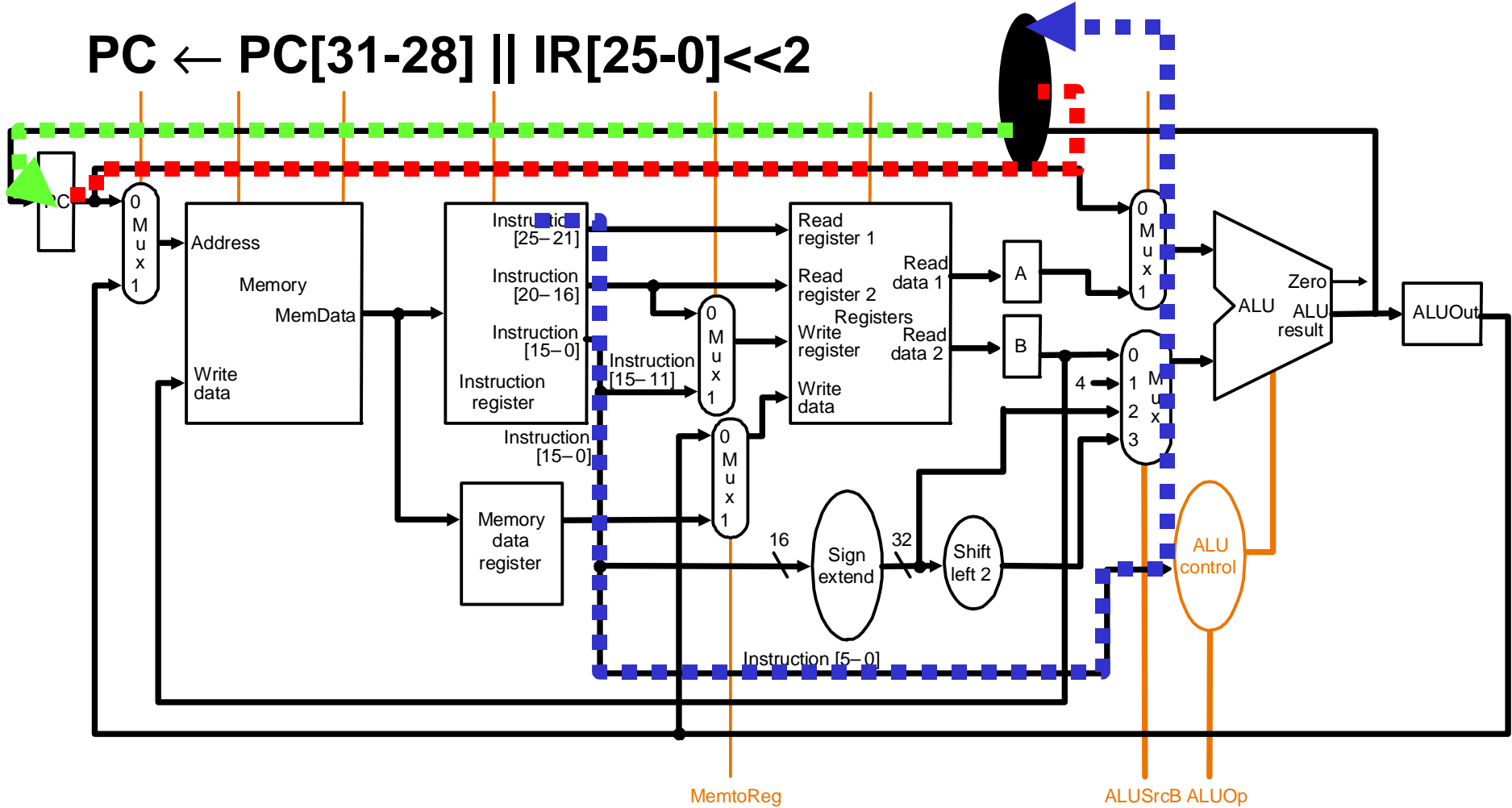


T₃ BEQ1 I-Format State Machine = \$rs + offset



T₃ Jump1 (Jump Address)

$$PC \leftarrow PC[31-28] \parallel IR[25-0] \ll 2$$



Moore Output State Tables: O(State)

| State | T ₁ | T ₂ | T _{3-R} | T _{4-R} | T _{3-I} | T _{4-SW} | T _{4-LW} | T _{5-LW} |
|--------------|----------------|----------------|------------------|------------------|------------------|-------------------|-------------------|-------------------|
| MemRead | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
| MemWrite | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |
| MUX IorD | 0=PC | X | X | X | X | 1=ALU | 1=ALU | X |
| IRWrite | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| ALUOP | 0=+ | 0 | 2=op | X | 0=add | X | X | X |
| MUX ALUSrcA | 0=PC | 0 | 1=A=\$rs | X | 1=A=\$rs | X | X | X |
| MUX ALUSrcB | 1=4 | 3 | 0=B=\$rt | X | 2=sign | X | X | X |
| PCWrite | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| MUX PCSource | 0=AL | X | X | X | X | X | X | X |
| RegWrite | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 |
| MUX MemtoReg | X | X | X | 0=ALU | X | X | X | 1=MDR |
| MUX RegDst | X | X | X | 1=\$rd | X | X | X | 1=\$rt |

Multi-cycle: 5 execution steps



- **T₁ (a,lw,sw,beq,j) Instruction Fetch**
- **T₂ (a,lw,sw,beq,j) Instruction Decode and Register Fetch**
- **T₃ (a,lw,sw,beq,j) Execution, Memory Address Calculation, or Branch Completion**
- **T₄ (a,lw,sw) Memory Access or R-type instruction completion**
- **T₅ (a,lw) Write-back step**

INSTRUCTIONS TAKE FROM 3 - 5 CYCLES!

Multi-cycle Approach

All operations in each clock cycle T_i are done in parallel not sequential!

For example, T_1 , $IR = \text{Memory}[PC]$ and $PC = PC + 4$ are done simultaneously!

| | Step name | Action for R-type instructions | Action for memory-reference instructions | Action for branches | Action for jumps |
|-------|--|---|--|-------------------------------------|--|
| T_1 | Instruction fetch | $IR = \text{Memory}[PC]$ $PC = PC + 4$ | | | |
| T_2 | Instruction decode/register fetch | $A = \text{Reg} [IR[25-21]]$ $B = \text{Reg} [IR[20-16]]$ $ALUOut = PC + (\text{sign-extend} (IR[15-0]) \ll 2)$ | | | |
| T_3 | Execution, address computation, branch/jump completion | $ALUOut = A \text{ op } B$ | $ALUOut = A + \text{sign-extend} (IR[15-0])$ | if $(A == B)$ then $PC = ALUOut$ | $PC = PC [31-28] \parallel (IR[25-0] \ll 2)$ |
| T_4 | Memory access or R-type completion | $\text{Reg} [IR[15-11]] = ALUOut$ | Load: $MDR = \text{Memory}[ALUOut]$ or Store: $\text{Memory} [ALUOut] = B$ | | |
| T_5 | Memory read completion | | Load: $\text{Reg}[IR[20-16]] = MDR$ | | |

Between Clock T_2 and T_3 the microcode sequencer will do a dispatch 1