EECS 322: Computer Architecture

Single-cycle
Multi-cycle FSM controller
Multi-cycle microcontroller

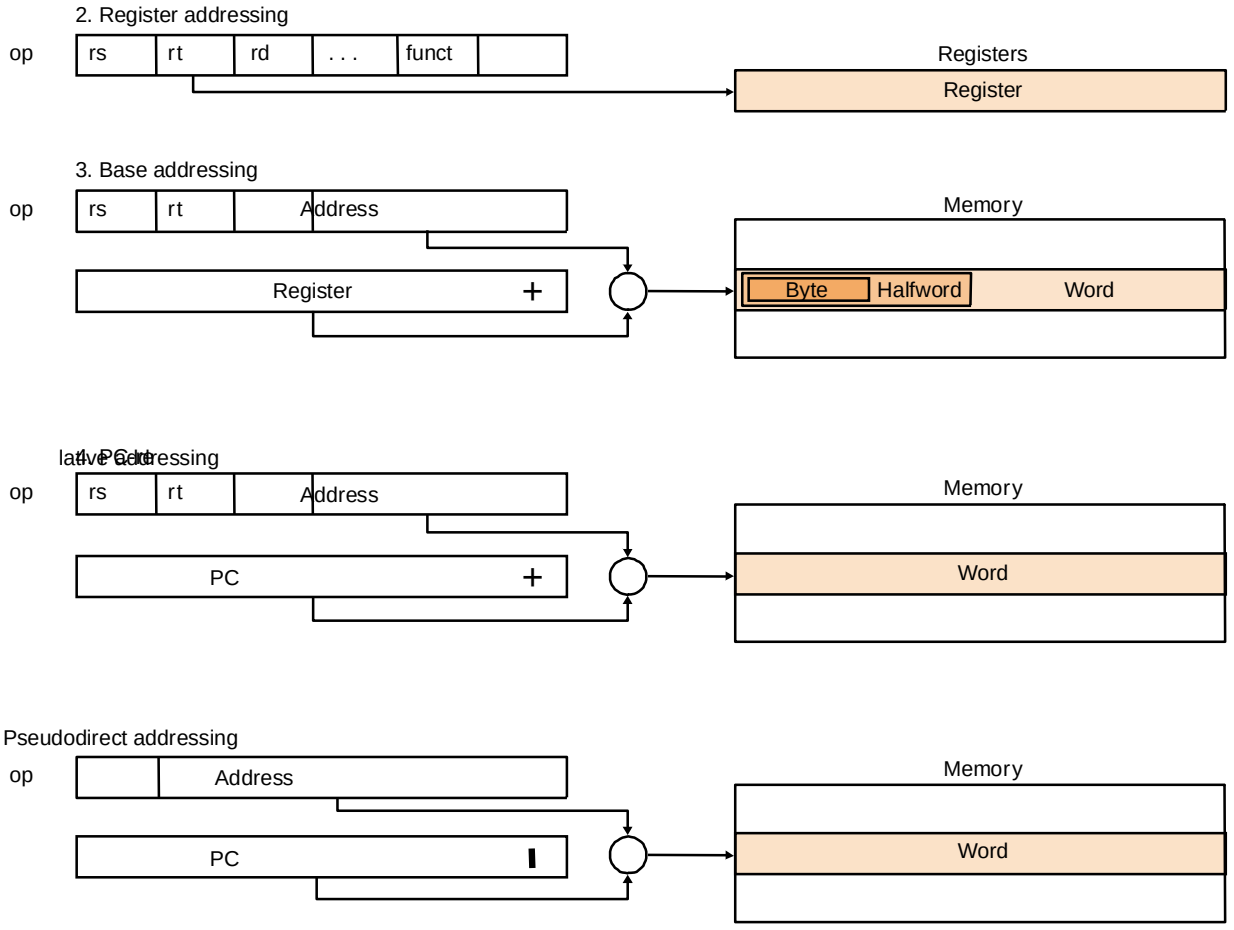
MIPS instruction formats

Arithmetic
add $rd,$rs,$rt

Data Transfer
lw $rd,offset($rs)
sw $rd,offset($rs)

Conditional branch
beq $rd,$rs,raddr

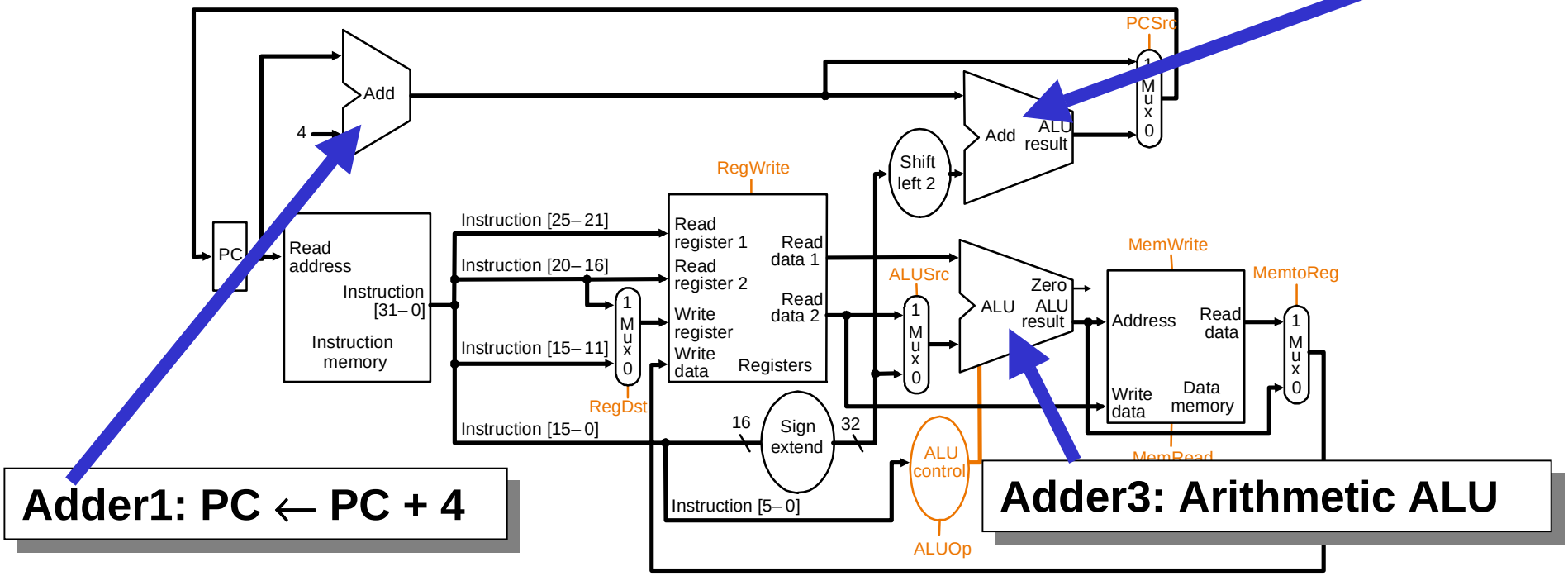
Unconditional jump
j addr



Single Cycle Implementation

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

Adder2: PC←PC+signext(IR[15-0]) <<2



Adder1: PC ← PC + 4

Adder3: Arithmetic ALU

Single Cycle = 2 adders + 1 ALU

Single/Multi-Clock Comparison

add = 6ns = Fetch(2ns)+RegR(1ns)+ALU(2ns)+RegW(2ns)

lw = 8ns = Fetch(2ns)+RegR(1ns)+ALU(2ns)+MemR(2ns)+RegW(2ns)

sw = 7ns = Fetch(2ns)+RegR(1ns)+ALU(2ns)+MemW(2ns)

beq = 5ns = Fetch(2ns)+RegR(1ns)+ALU(2ns)

j = 2ns = Fetch(2ns)

$$\frac{\text{CPU single}-\text{cycle clock}}{\text{CPU multi}-\text{cycle clock}} = \frac{8ns}{6.3ns} = 1.27 \text{ times faster}$$

Architectural improved performance without speeding up the clock!

Some Design Trade-offs

High level design techniques

Algorithms: change instruction usage

minimize $\sum n_{instruction} * t_{instruction}$

Architecture: Datapath, FSM, Microprogramming

adders: ripple versus carry lookahead

multiplier types, …

Lower level design techniques (closer to physical design)

clocking: single verus multi clock

technology: layout tools: better place and route

process technology: 0.5 micron to .18 micron

Single-cycle problems

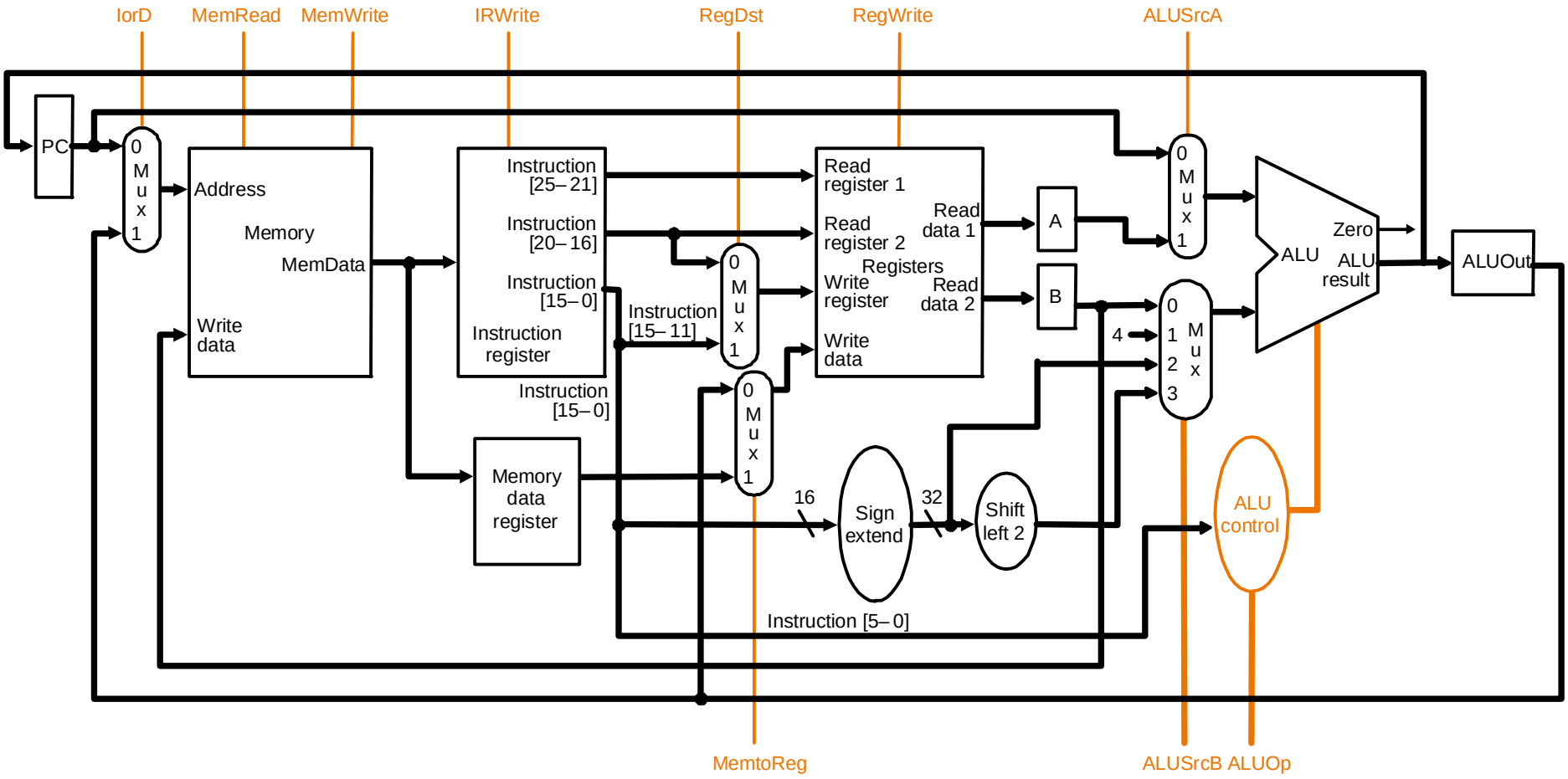
- Single Cycle Problems:
  - what if we had a more complicated instruction like floating point? (fadd = 30ns, fmul=100ns)
  - wasteful of area (2 adders + 1 ALU)
- One Solution:
  - use a “smaller” cycle time (if the technology can do it)
  - have different instructions take different numbers of cycles
  - a “multicycle” datapath (1 ALU)

- Multi-cycle approach
  - We will be reusing functional units:
    ALU used to increment PC (Adder1)
    and to compute address (Adder2)
  - Memory used for instruction and data

Reality Check: Intel 8086 clock cycles

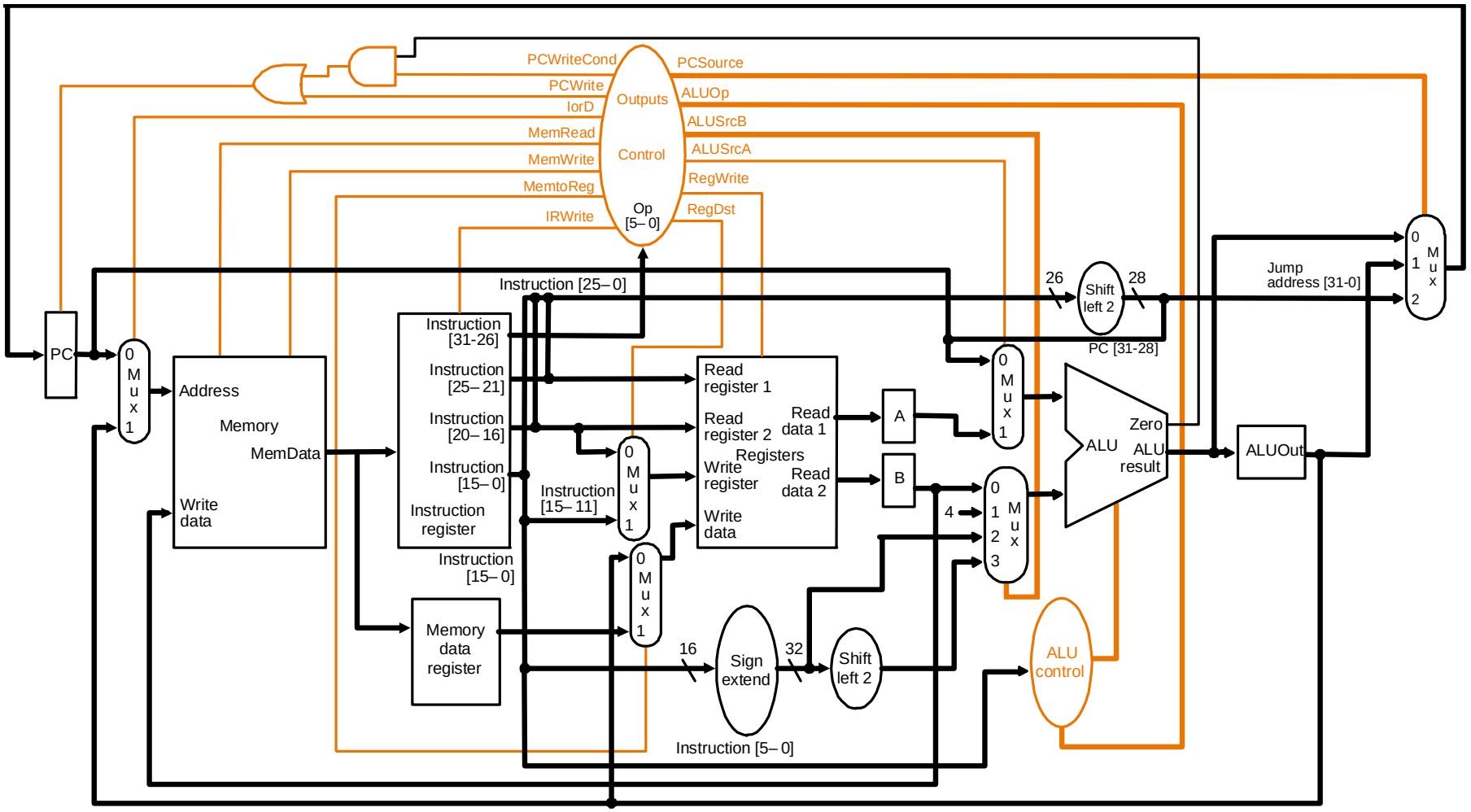
Arithmetic			
3	add	reg16, reg16	
118-133	mul	dx:ax, reg16	very slow!!
128-154	imul	dx:ax, reg16	
114-162	div	dx:ax, reg16	
165-184	idiv	dx:ax, reg16	
Data Transfer			
14	mov	reg16, mem16	
15	mov	mem16, reg16	
Conditional Branch			
4/16	je	displacement8	
Unconditional Jump			
15	jmp	segment:offset16	

Multi-cycle Datapath

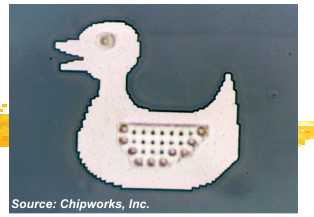


Multi-cycle = 1 ALU + Controller

Multi-cycle Datapath: with controller



Multi-cycle: 5 execution steps



From: Siemens SAB80C535-N
8-bit Microcontroller

CHIPWORKS

- T_1 (a,lw,sw,beq,j) Instruction Fetch
- T_2 (a,lw,sw,beq,j) Instruction Decode and Register Fetch
- T_3 (a,lw,sw,beq,j) Execution, Memory Address Calculation, or Branch Completion
- T_4 (a,lw,sw) Memory Access or R-type instruction completion
- T_5 (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!

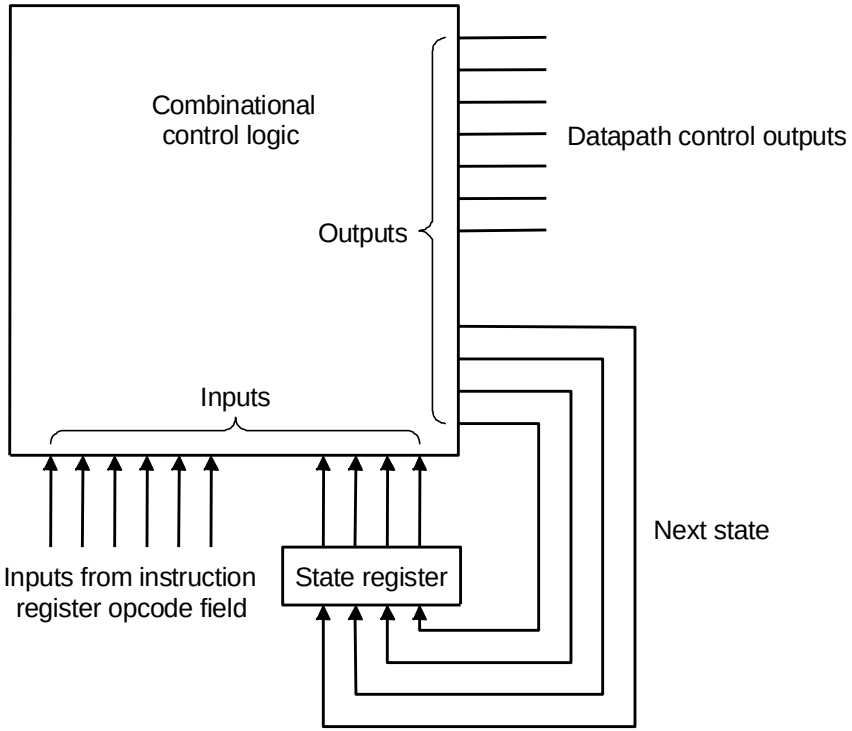
For example, T_1, IR = Memory[PC] and PC=PC+4 are done simultaneously!

	Step name	Action for R-type instructions	Action for memory-reference instructions	Action for branches	Action for jumps
T_1	Instruction fetch	IR = Memory[PC] PC = PC + 4			
T_2	Instruction decode/register fetch	A = Reg [IR[25-21]] B = Reg [IR[20-16]] ALUOut = PC + (sign-extend (IR[15-0]) << 2)			
T_3	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] II (IR[25-0]<<2)
T_4	Memory access or R-type completion	Reg [IR[15-11]] = ALUOut	Load: MDR = Memory[ALUOut] or Store: Memory [ALUOut] = B		
T_5	Memory read completion		Load: Reg[IR[20-16]] = MDR		

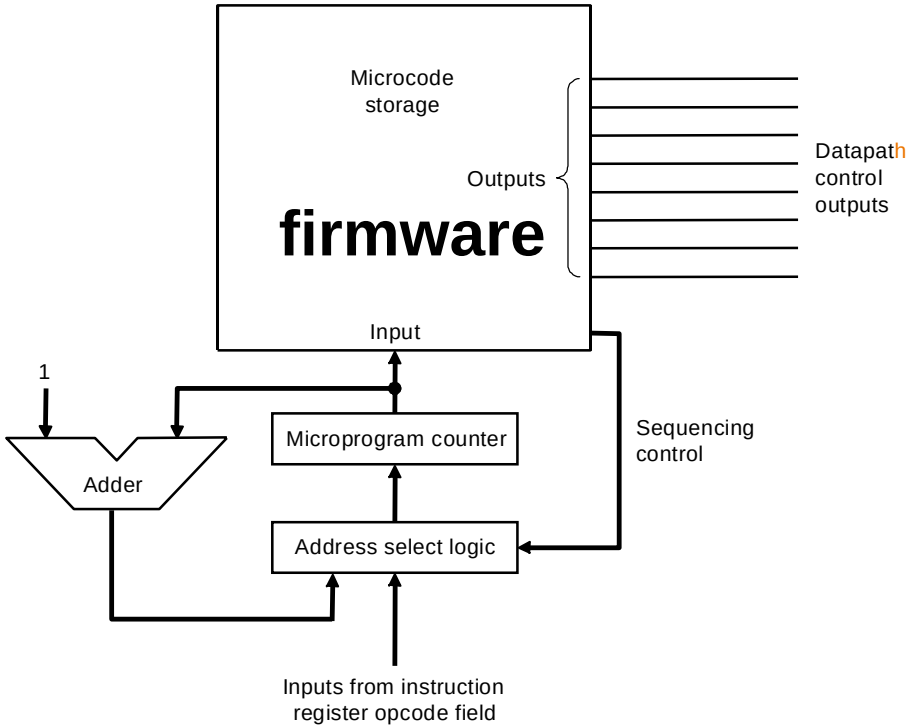
Between Clock T_2 and T_3 the microcode sequencer will do a dispatch 1

Multi-cycle using Microprogramming

Finite State Machine (*hardwired control*)



Microcode controller



Requires microcode memory to be faster than main memory

Microcode: Trade-offs

- Distinction between specification and implementation is sometimes blurred
- Specification Advantages:
  - Easy to design and write (maintenance)
  - Design architecture and microcode in parallel
- Implementation (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:
  - Control is implemented on same chip as processor
  - ROM is no longer faster than RAM
  - No need to go back and make changes

Microinstruction format

Field name	Value	Signals active	Comment
ALU control	Add	ALUOp = 00	Cause the ALU to add.
	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.
SRC2	B	ALUSrcB = 00	Register B is the second ALU input.
	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.
Register control	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.
	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.
Memory	Read PC	MemRead, IorD = 0	Read memory using the PC as address; write result into IR (and the MDR).
	Read ALU	MemRead, IorD = 1	Read memory using the ALUOut as address; write result into MDR.
	Write ALU	MemWrite, IorD = 1	Write memory using the ALUOut as address, contents of B as the data.
PC write control	ALU	PCSource = 00 PCWrite	Write the output of the ALU into the PC.
	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.
Sequencing	Seq	AddrCtl = 11	Choose the next microinstruction sequentially.
	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.

Microinstruction format: Maximally vs. Minimally Encoded

- **No encoding:**
  - 1 bit for each datapath operation
  - faster, requires more memory (logic)
  - used for Vax 780 — an astonishing 400K of memory!

- **Lots of encoding:**
  - send the microinstructions through logic to get control signals
  - uses less memory, slower

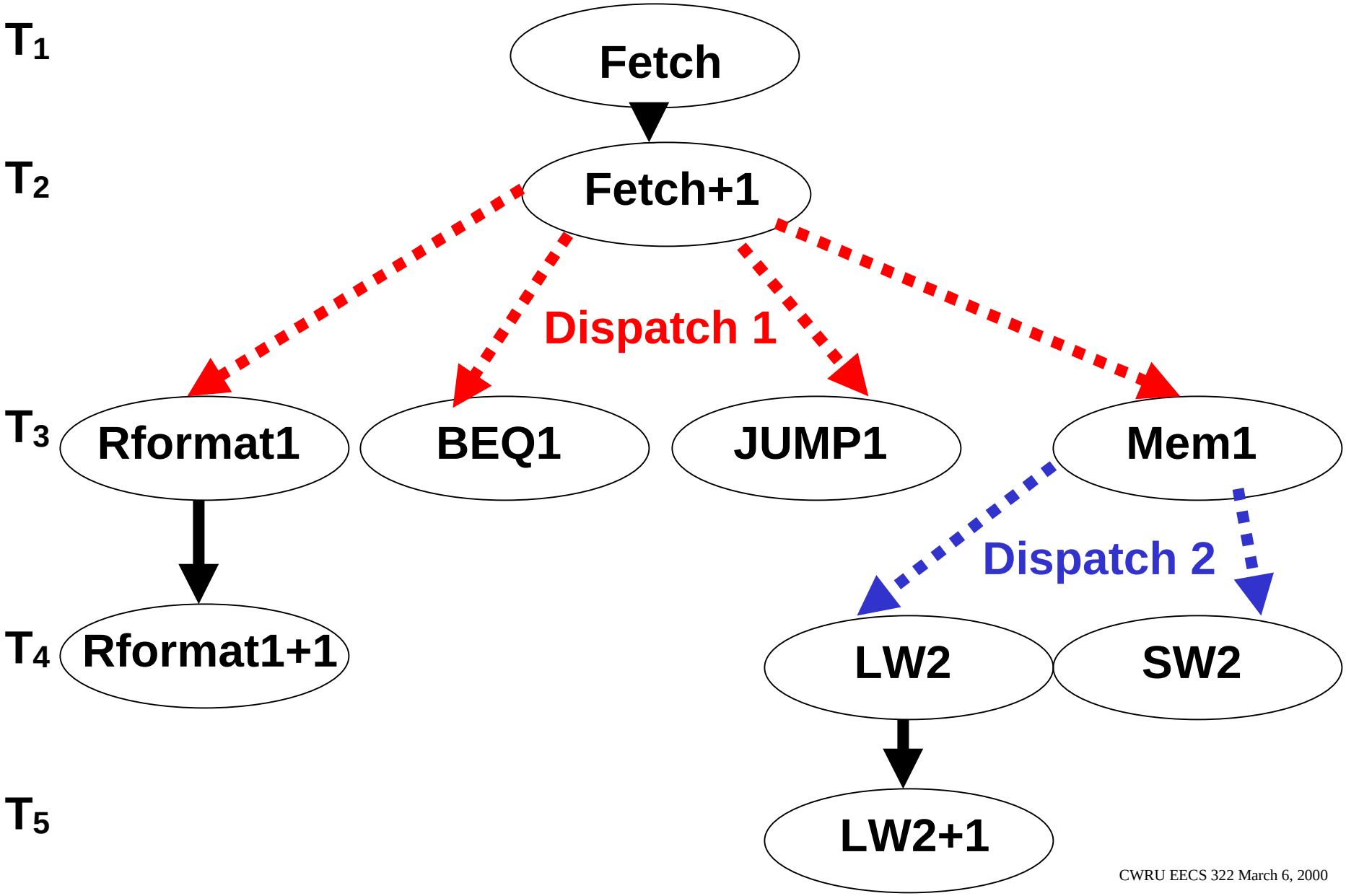
- **Historical context of CISC:**
  - Too much logic to put on a single chip with everything else
  - Use a ROM (or even RAM) to hold the microcode
  - It's easy to add new instructions

Microprogramming: program

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

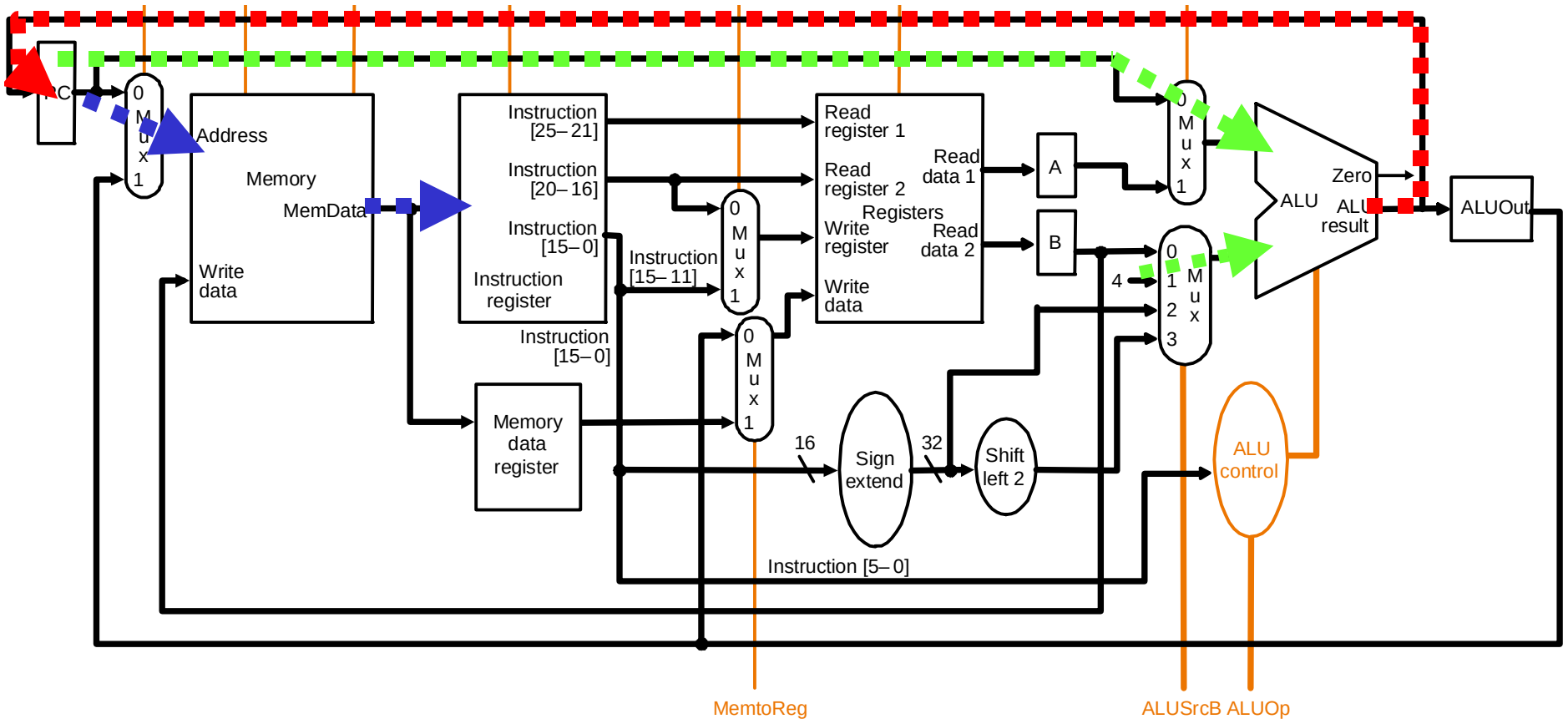
Step name	Action for R-type instructions	Action for memory-reference instructions	Action for branches	Action for jumps
Instruction fetch	IR = Memory[PC] PC = PC + 4			
Instruction decode/register fetch	A = Reg [IR[25-21]] B = Reg [IR[20-16]] ALUOut = PC + (sign-extend (IR[15-0]) << 2)			
Execution, address 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] II (IR[25-0]<<2)
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		

Microprogramming: program overview



Microprogram steping: T_1 Fetch

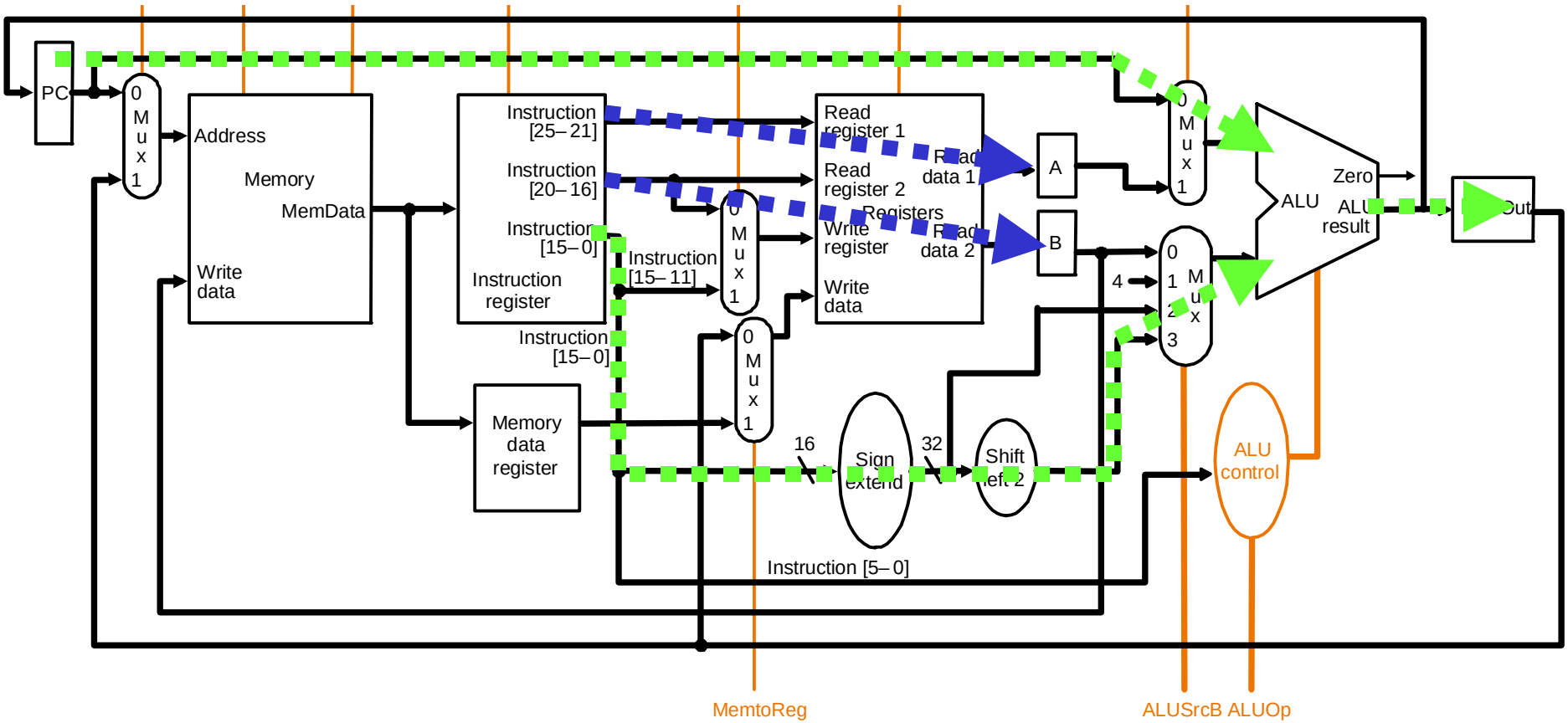
(Done in parallel) IR←MEMORY[PC] & PC ← PC + 4



Label	ALU	SRC1	SRC2	RCntl	Memory	PCwrite	Seq
Fetch	add	pc	4		ReadPC	ALU	Seq

T_2 Fetch + 1

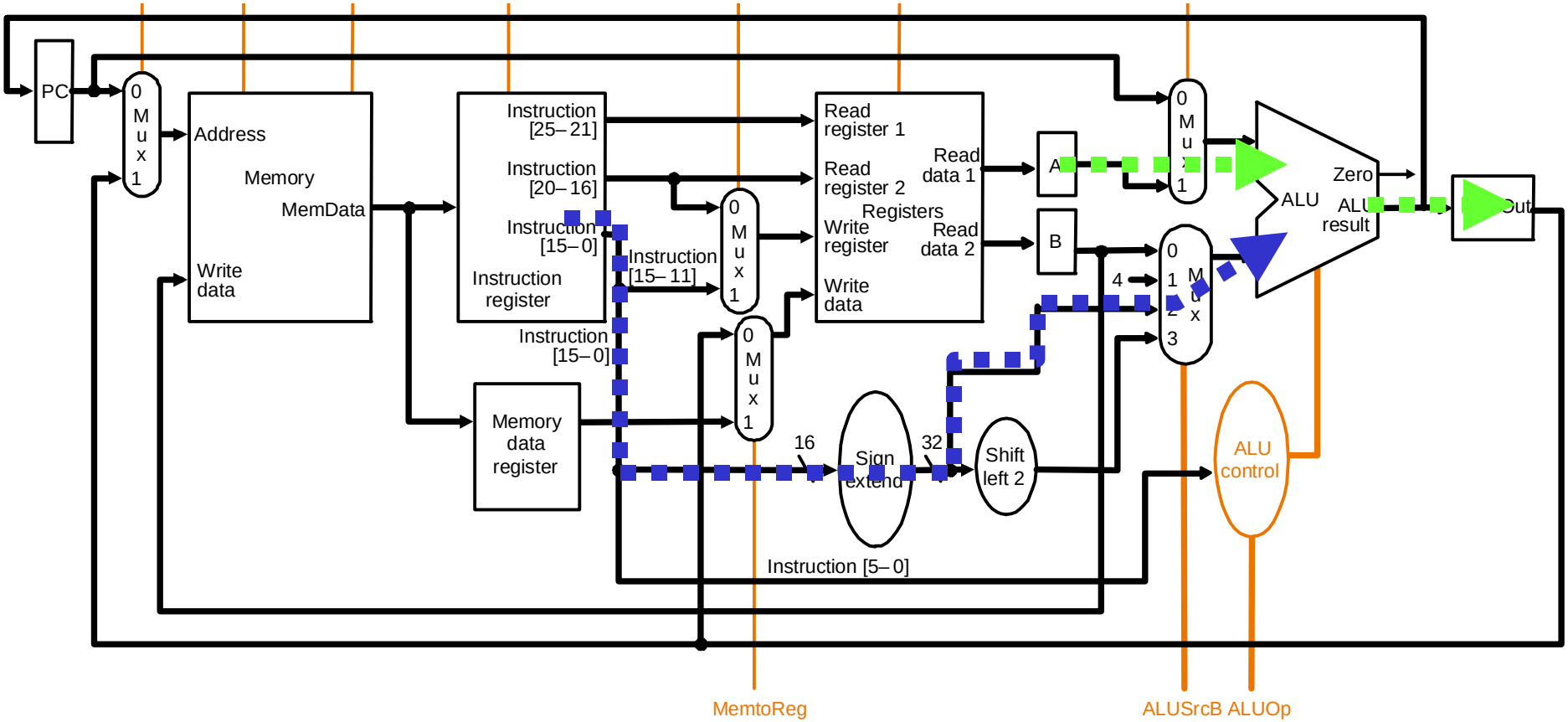
A←Reg[IR[25-21]] & B←Reg[IR[20-16]] & ALUOut←PC+signext(IR[15-0]) <<2



Label	ALU	SRC1	SRC2	RCntl	Memory	PCwrite	Seq
	add	pc	ExtSh	Read			D#1

T_3 Dispatch 1: Mem1

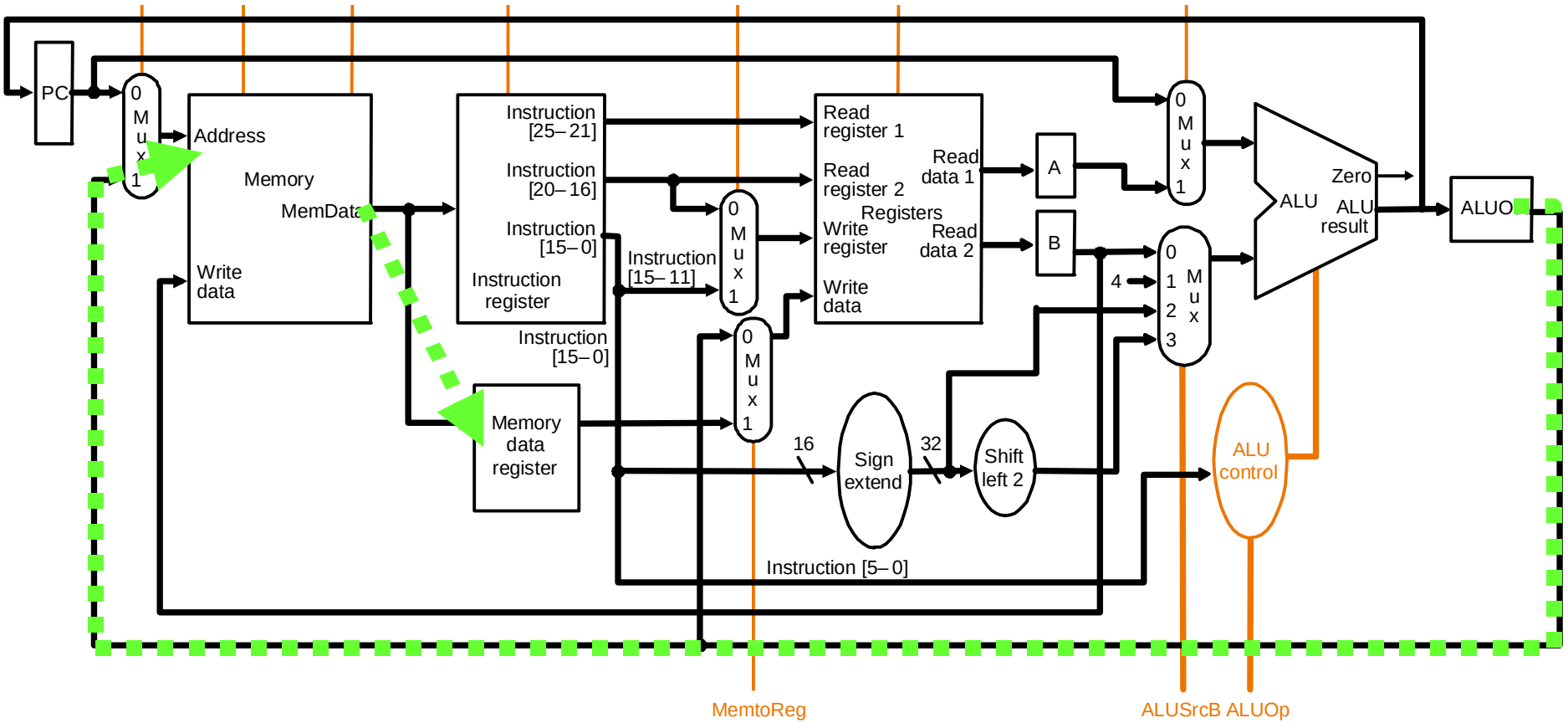
ALUOut ← A + sign_extend(IR[15-0])



Label	ALU	SRC1	SRC2	RCntl	Memory	PCwrite	Seq
Mem1	add	A	ExtSh				D#2

T_4 Dispatch 2: LW2

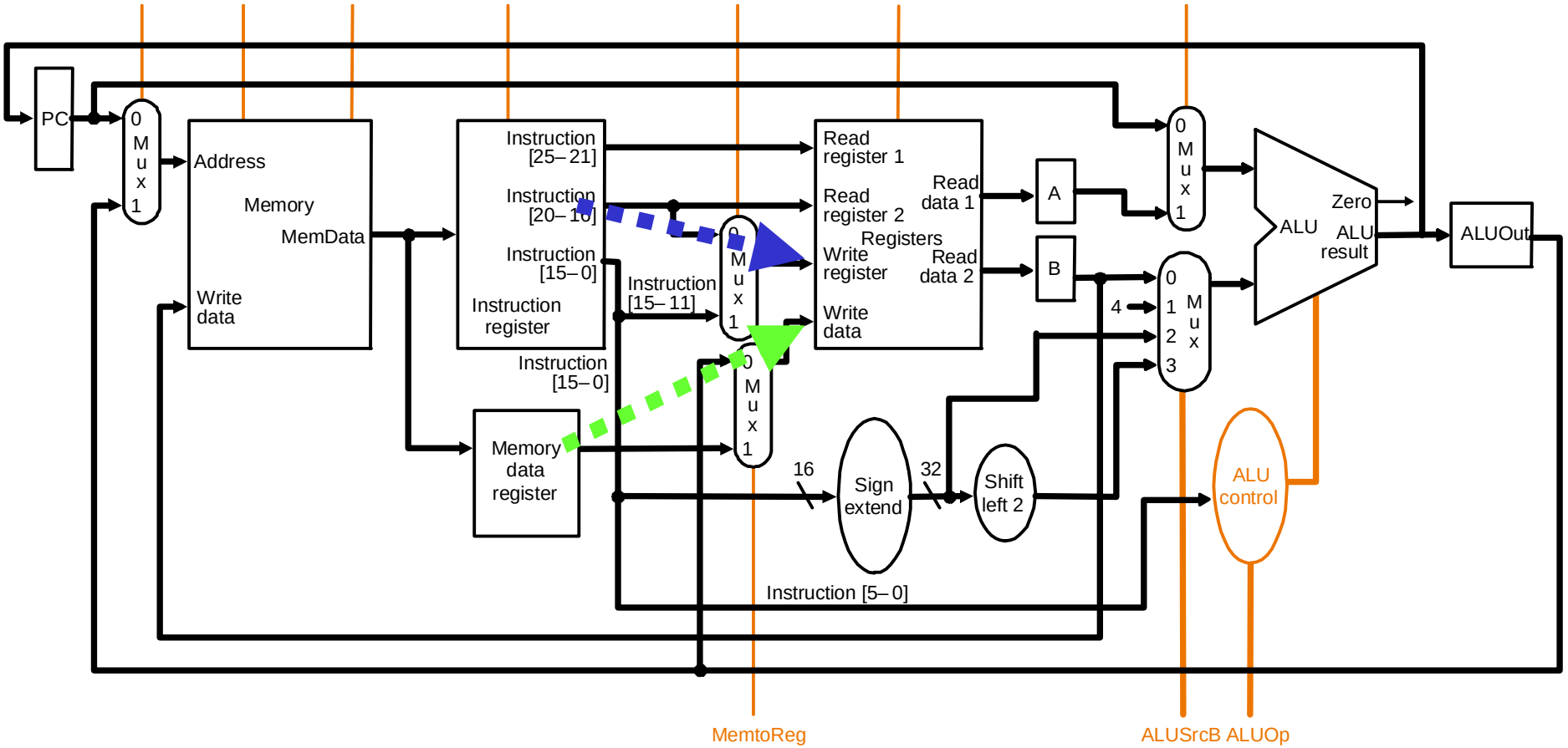
MDR ← Memory[ALUOut]



Label	ALU	SRC1	SRC2	RCntl	Memory	PCwrite	Seq
LW2					ReadALU		Seq

T_5 LW2+1

Reg[IR[20-16]] ← MDR

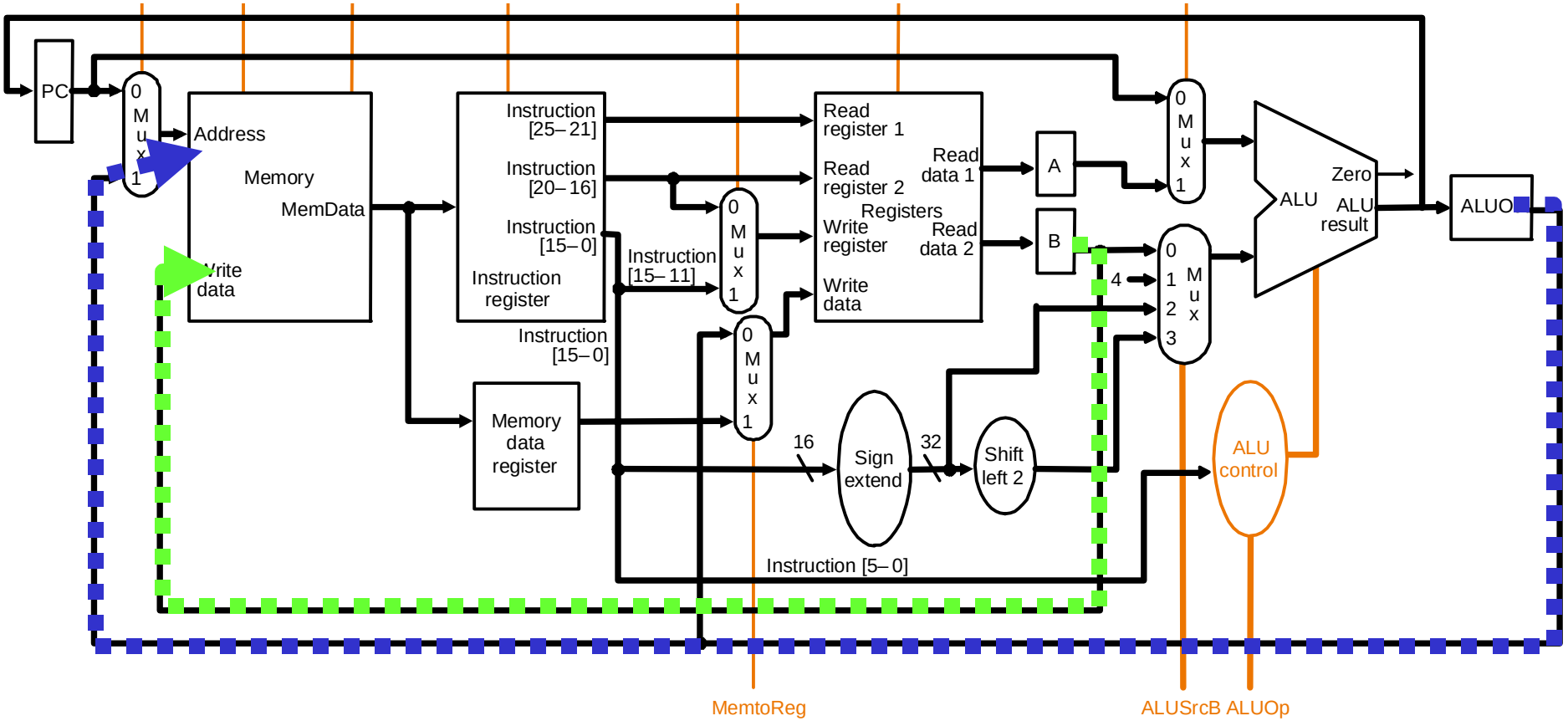


Label ALU SRC1 SRC2 RCntl Memory PCwrite Seq

WMDR Fetch

T_4 Dispatch 2: SW2

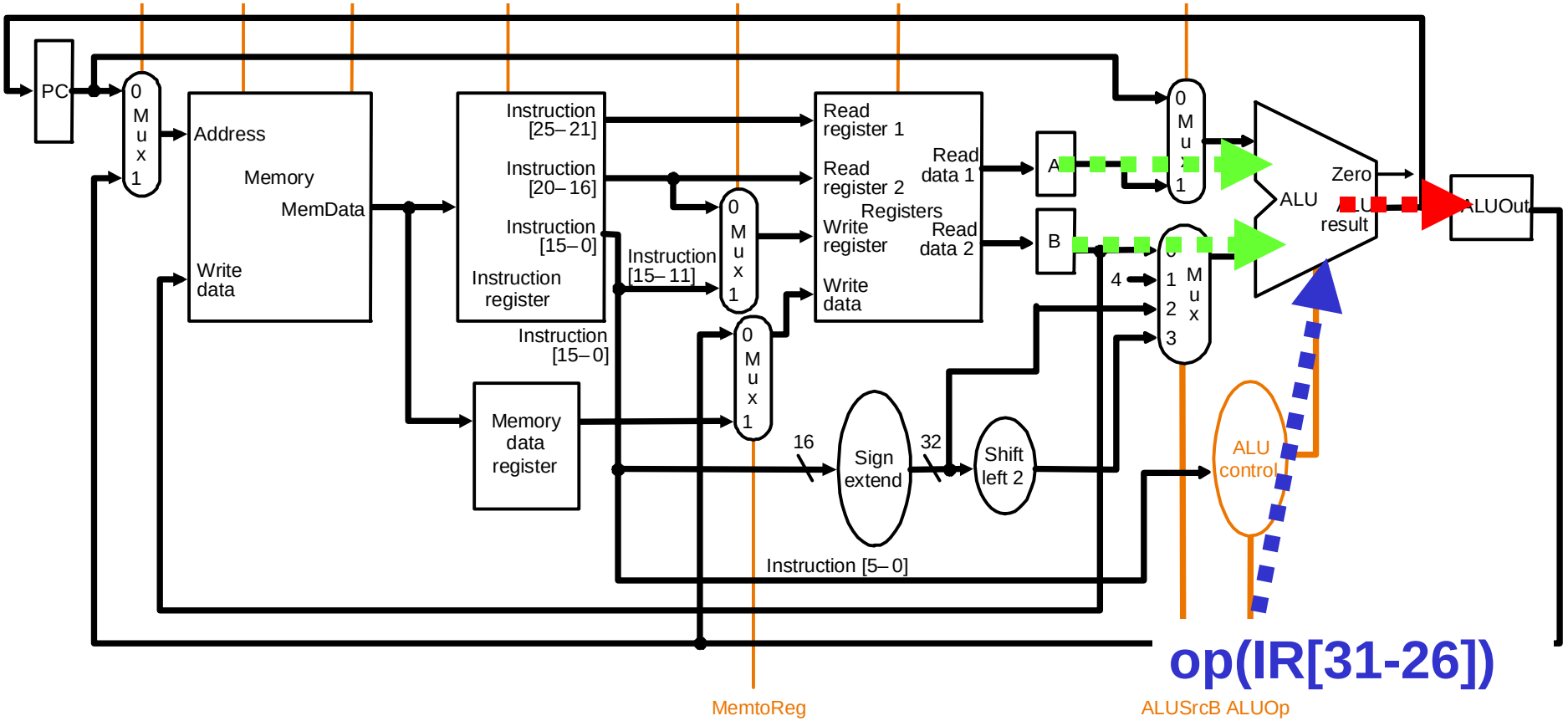
Memory[ALUOut] ← B



Label	ALU	SRC1	SRC2	RCntl	Memory	PCwrite	Seq
SW2					WriteALU		Fetch

T_3 Dispatch 1: Rformat1

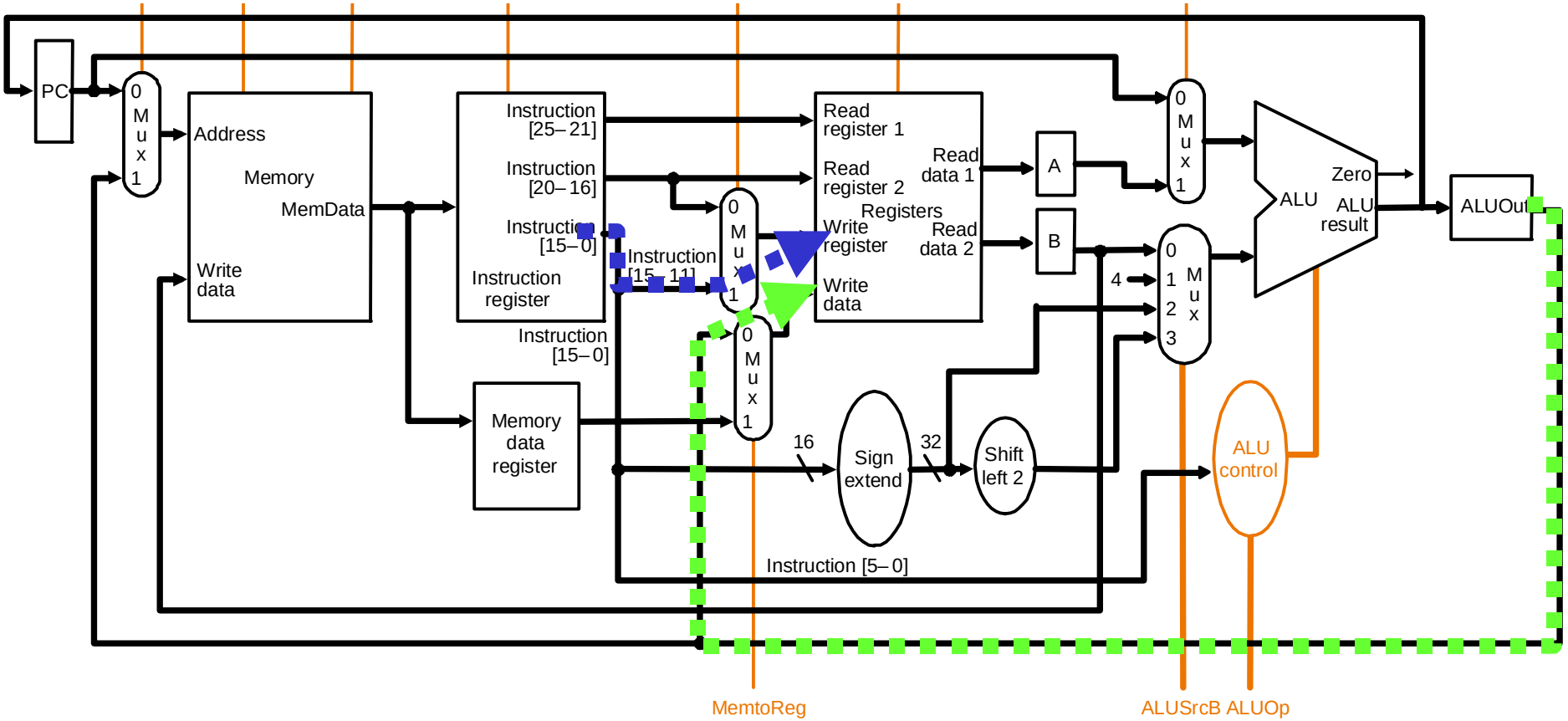
ALUOut ← A op(IR[31-26]) B



Label	ALU	SRC1	SRC2	RCntl	Memory	PCwrite	Seq
Rf...1	op	A	B				Seq

T_4 Dispatch 1: Rformat1+1

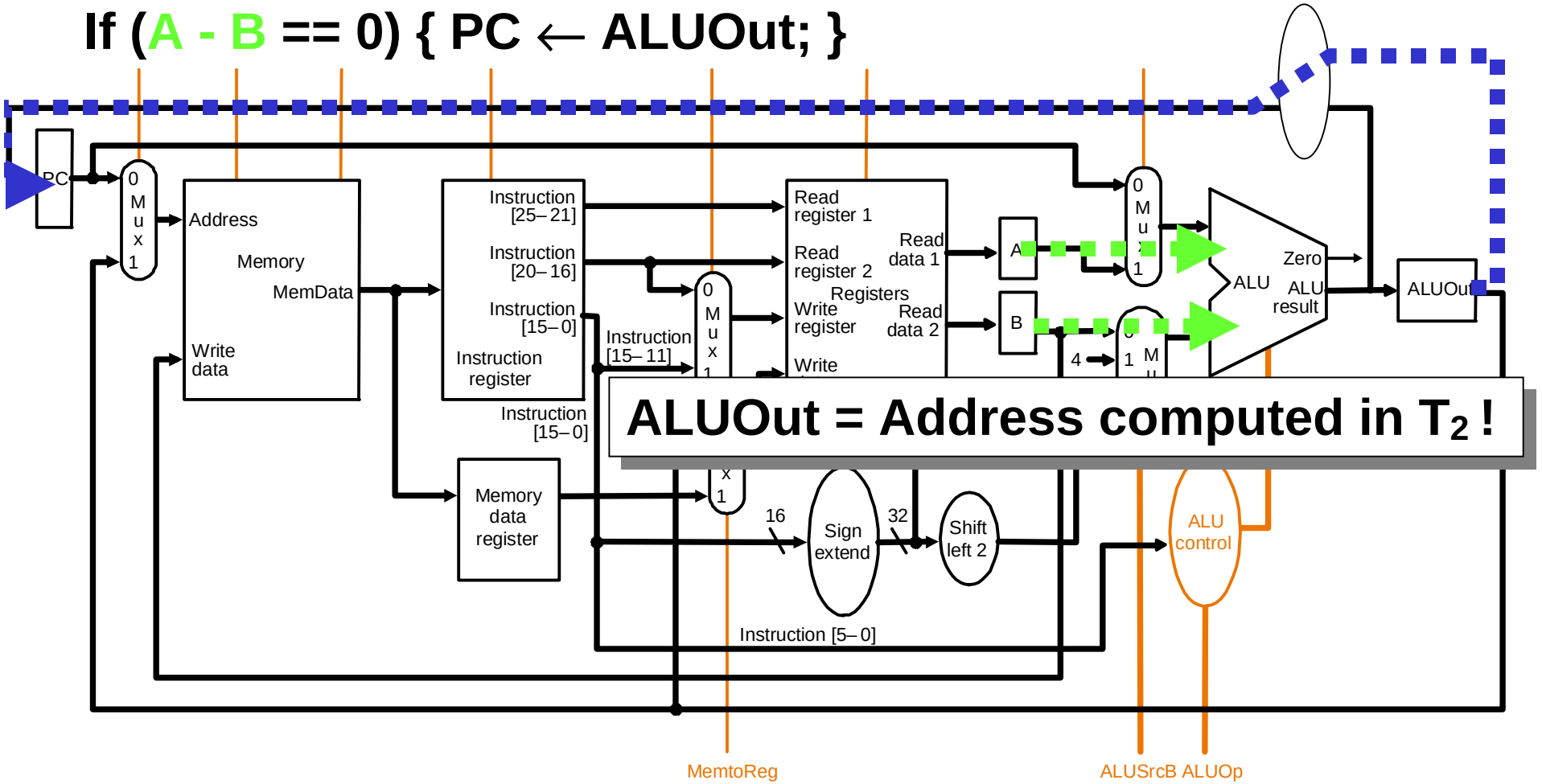
Reg[IR[15-11]] ← ALUOut



Label	ALU	SRC1	SRC2	RCntl	Memory	PCwrite	Seq
				WALU			Fetch

T_3 Dispatch 1: BEQ1

If (A - B == 0) { PC ← ALUOut; }

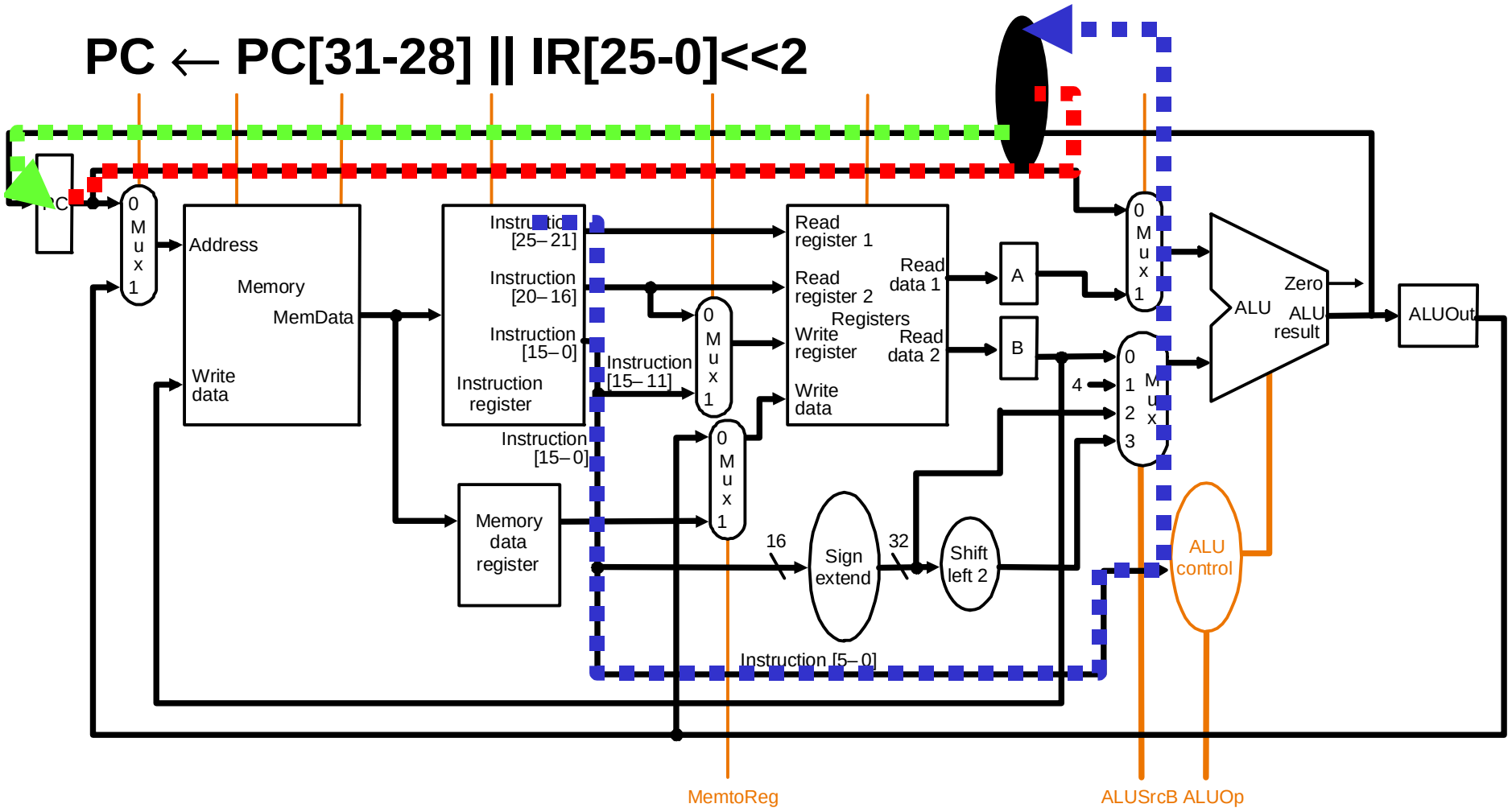


ALUOut = Address computed in T_2 !

Label	ALU	SRC1	SRC2	RCntl	Memory	PCwrite	Seq
BEQ1	subt	A	B			ALUOut-0	Fetch

T_3 Dispatch 1: Jump1

PC ← PC[31-28] || IR[25-0]<<2



Label	ALU	SRC1	SRC2	RCntl	Memory	PCwrite	Seq
Jump1						Jaddr	Fetch

The Big Picture