A. Pipelining: Basic Concepts

What is pipelining?

How is the pipelining Implemented?

What makes pipelining hard to implement?

What is Pipelining ?

- Pipelining:
  - "*A technique designed into some computers to increase speed by starting the execution of one instruction before completing the previous one.*"

    ----*Modern English-Chinese Dictionary*
  - implementation technique whereby different instructions are overlapped in execution at the same time.
  - implementation technique to make fast CPUs

It likes Auto Assembly line

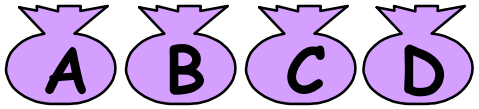
- An arrangement of workers, machines, and equipment in which the **product being assembled passes consecutively from operation to operation until completed.**

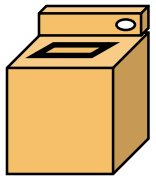
- **Ford installs first moving assembly line in 1913.** The right picture shows the moving assembly line at Ford Motor Company's michigan plant.

(84 distinct steps)

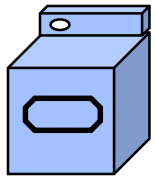
Why Pipelining: Its Natural

- Laundry
  - Ann, Brian, Cathy, Dave each have one load of clothes to wash, dry, and fold

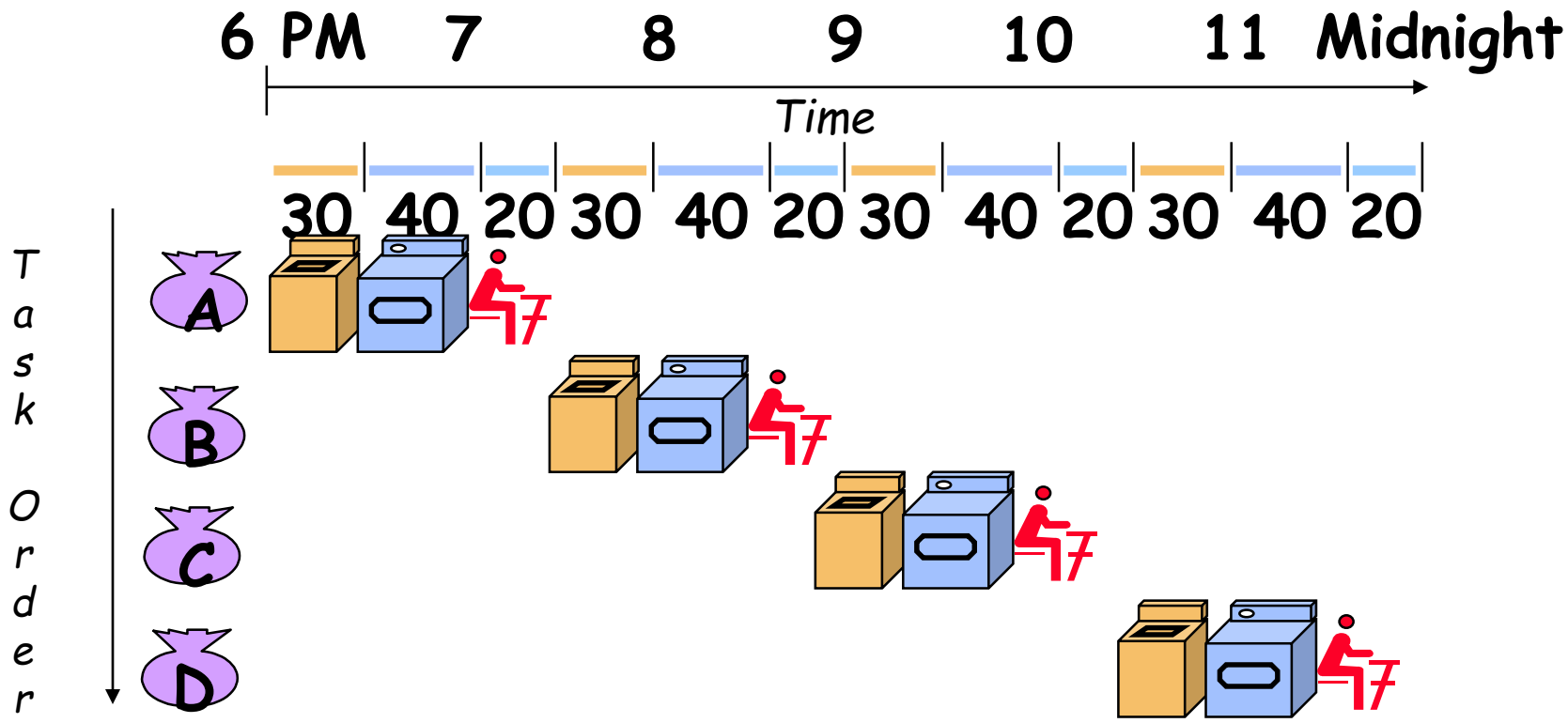




  - Washer takes 30 minutes
  - Dryer takes 40 minutes
  - "Folder" takes 20 minutes

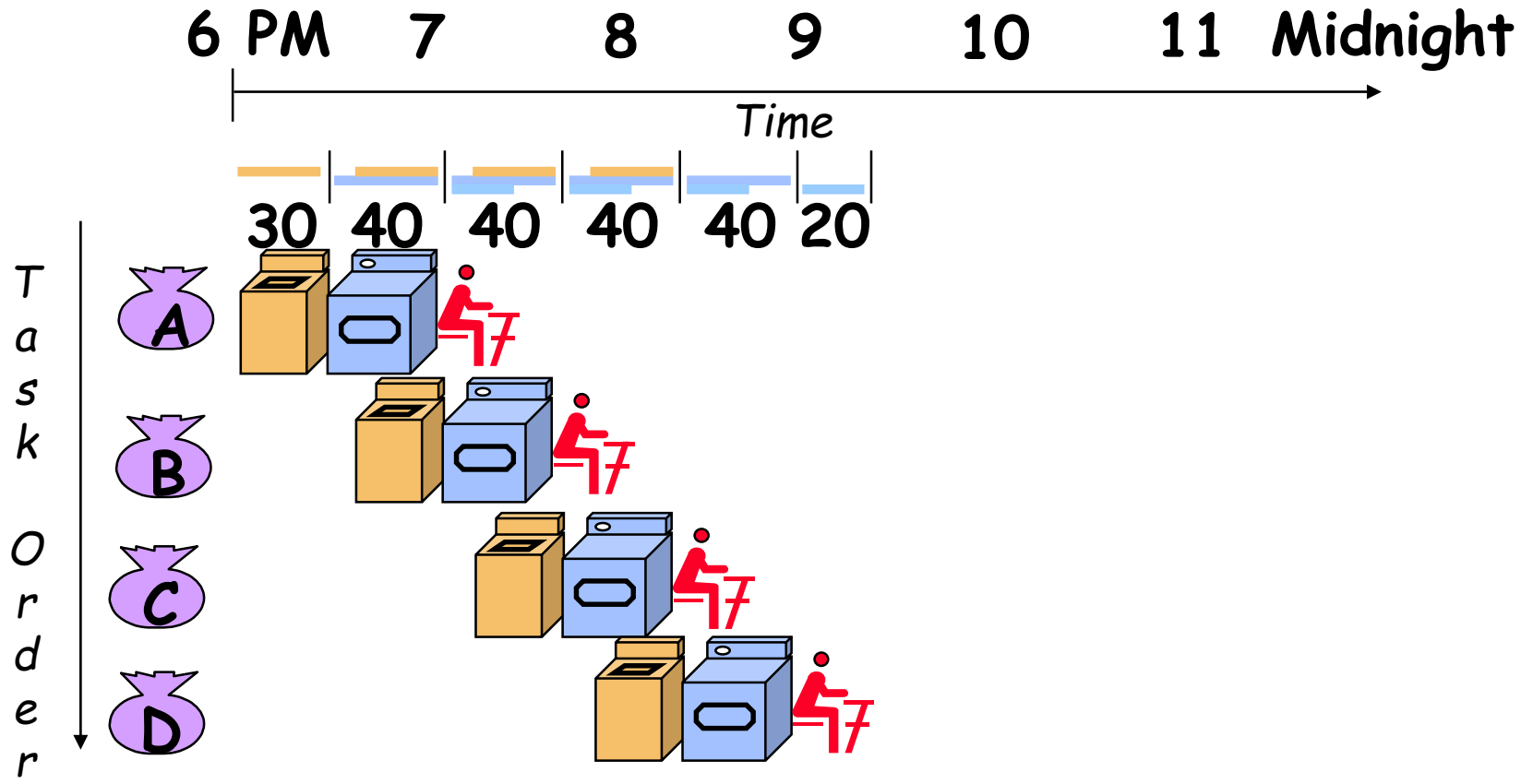


Sequential Laundry



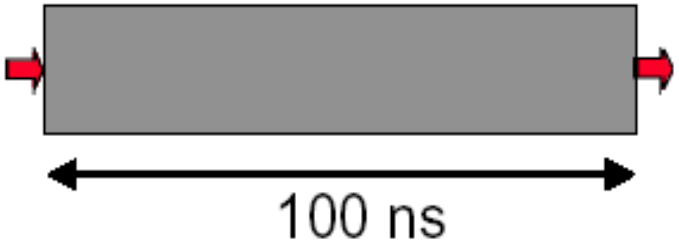
- Sequential laundry takes 6 hours for 4 loads
- If they learned pipelining, how long would laundry take?

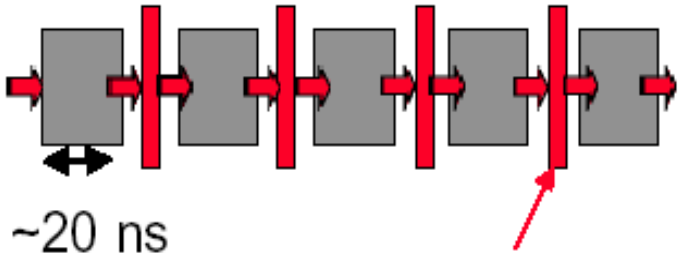
Pipelined Laundry--Start work ASAP



- Pipelined laundry takes 3.5 hours for 4 loads

Why pipelining : overlapped

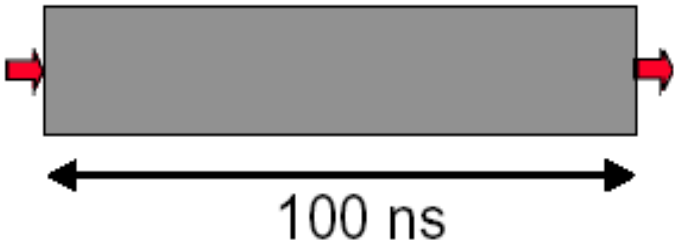


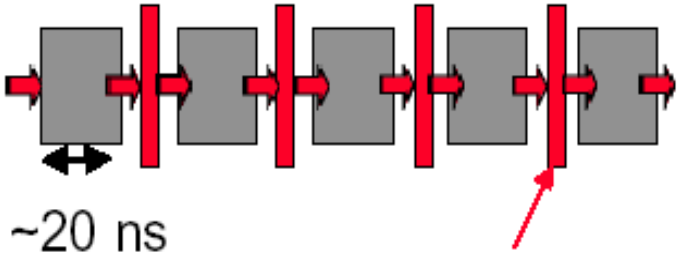


- Only deal one task each time.
- This task takes " such a long time"

- Latches, called pipeline registers' break up computation into 5 stages
- Deal 5 tasks at the same time.

Why pipelining: more faster





- Can "launch" a new computation every 100ns in this structure
- Can finish 10^7 computations per second

- Can launch a new computation every 20ns in pipelined structure
- Can finish 5×10^7 computations per second

Why pipelining : conclusion

- The key implementation technique used to Make fast CPU: decrease CPUtime.
- Improving of Throughput (rather than individual execution time)
- Improving of efficiency for resources (functional unit)

What is a pipeline ?

- A pipeline is like an auto assemble line
- A pipeline has many stages
- Each stage carries out a different part of instruction or operation
- The stages, which cooperates at a synchronized clock, are connected to form a pipe
- An instruction or operation enters through one end and progresses through the stages and exit through the other end
- Pipelining is an implementation technique that exploits parallelism among the instructions in a sequential instruction stream

Pipeline Characteristics --latency vs. Throughput

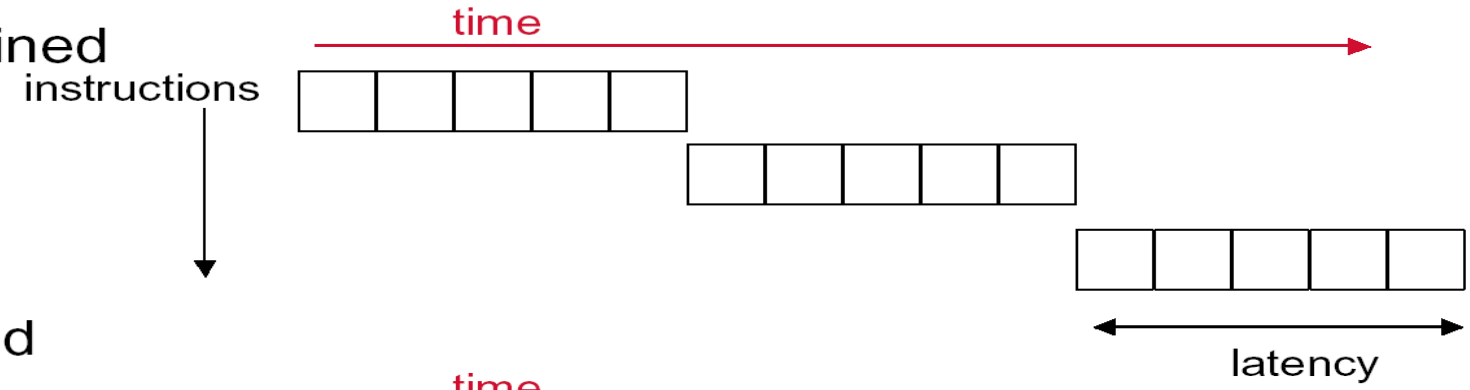
- **Latency**
  - Each instruction takes a certain time to complete. This is the latency for that operation.
  - It's the amount of time between when the instruction is issued and when it completes.
- **Throughput**
  - Number of items (cars, instructions) exiting the pipeline per unit time.
  - The throughput of the assembly line is the number of products completed per hour.
  - The throughput of a CPU pipeline is the number of instructions completed per second.

Pipeline Characteristics-clock cycle vs. Machine cycle

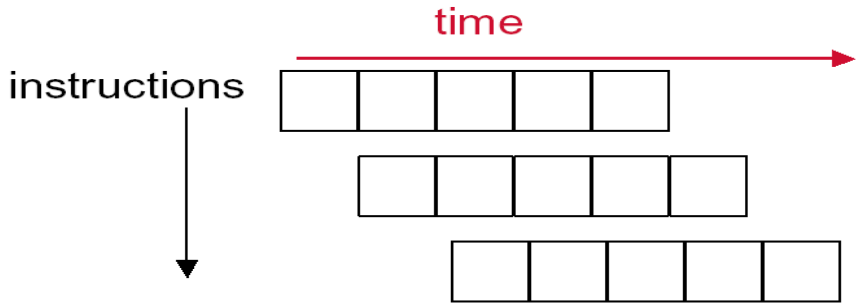
- **Clock cycle**
  - Everything in a CPU moves in lockstep, synchronized by the clock (The Heartbeat of CPU)
- **Machine cycle (Processor cycle, Stage time)**
  - time required to complete a single pipeline stage.
  - The pipeline designer's goal is to balance the length of each pipeline stage.
  - In many instances, machine cycle = max (times for all stages).
  - A machine cycle is usually one, sometimes two, clock cycles long, but rarely more.

Performance for Pipelining

- Unpipelined



- Pipelined



- Ideally, $\text{Speedup}_{\text{pipeline}} = \dfrac{\text{Time}_{\text{sequential}}}{\text{Pipeline Depth}}$

Ideal Performance for Pipelining

- If the stages are perfectly balanced, The time per instruction on the pipelined processor equal to:

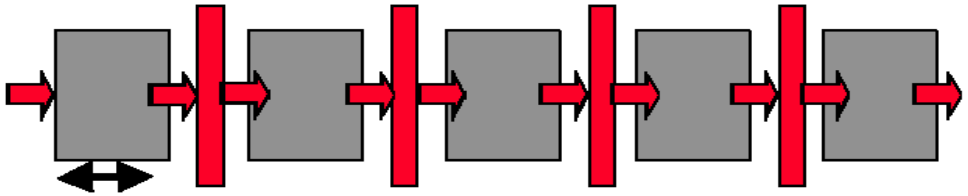
$$\frac{\textit{Time per instruction on unpipelined machine}}{\textit{Number of pipe stages}}$$

- So, **Ideal speedup equal to**

  Number of pipe stages.

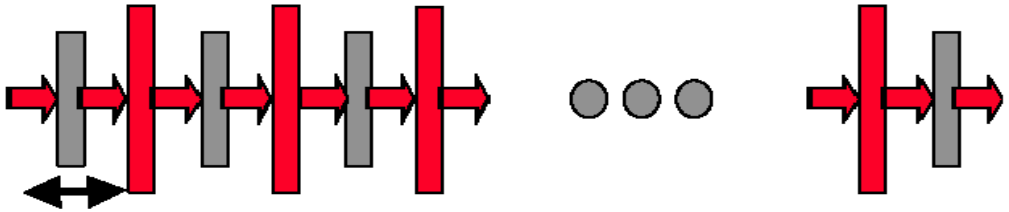
Why not just make a 50-stage pipeline?

- Some computations just won't divide into any finer (shorter in time) logical implementation.



~20 ns

5 stages OK

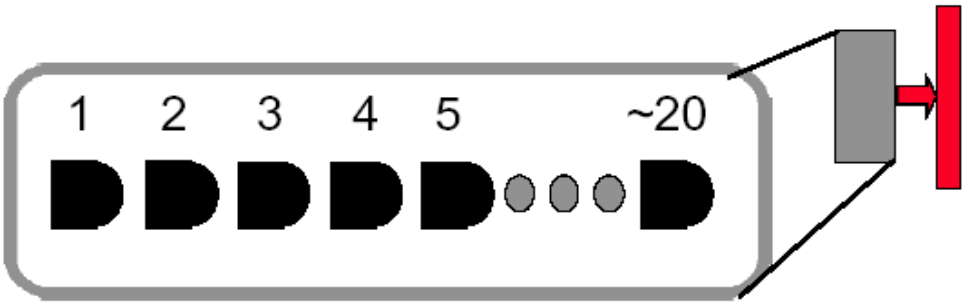


~2 ns

50 stages NO. Sorry!

Why not just make a 50-stage pipeline ?

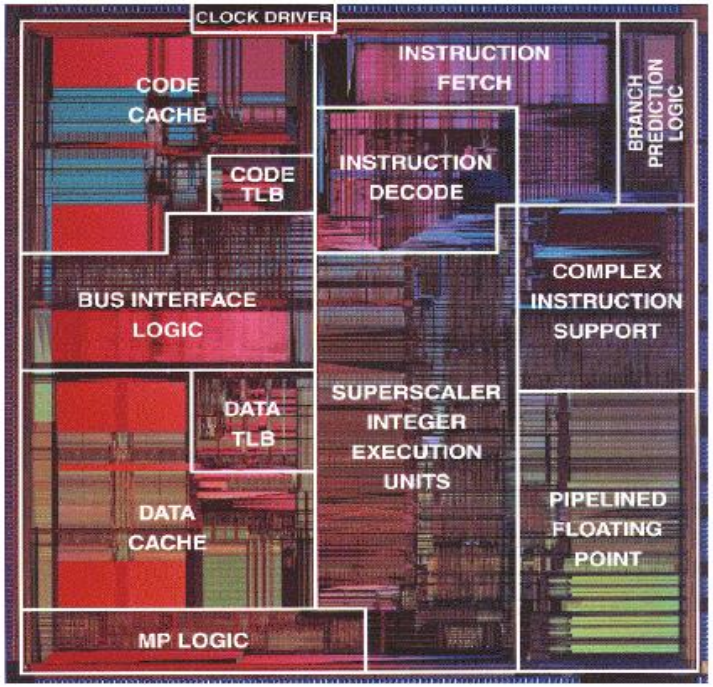
- Those latches are NOT free, they take up area, and there is a real delay to go THRU the latch itself.
  - Machine cycle > latch latency + clock skew
- In modern, deep pipeline (10-20 stages), this is a real effect
- Typically see logic "depths" in one pipe stage of 10-20 "gates".



At these speeds, and with this few levels of logic, latch delay is important

How Many Pipeline Stages?

- E.g., Intel
  - Pentium III, Pentium 4: 20+ stages
  - More than 20 instructions in flight
  - High clock frequency (>1GHz)
  - High IPC



- Too many stages:
  - Lots of complications
  - Should take care of possible depend instructions
  - Control logic is huge

Simple implementation of a RISC (MIPS)

- Start with Implementation without pipelining
  - single-cycle implementation
  - multi-cycle implementation
- Pipelining the RISC Instruction Set
- Pipelining performance issues
- How can we do it efficiently ?
- Examples

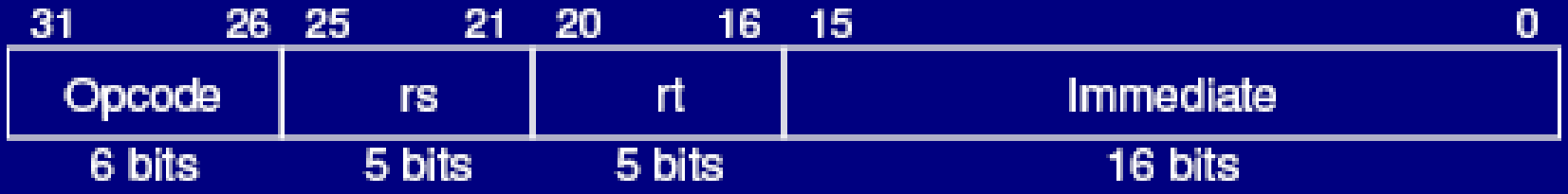
MIPS ISA

MIPS assembly language

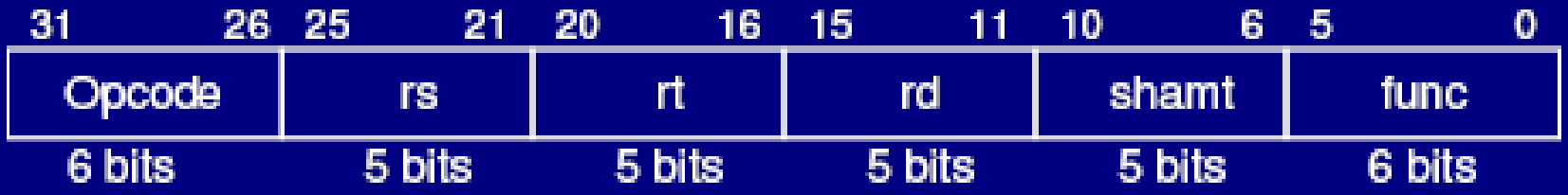
Category	Instruction	Example	Meaning	Comments
Arithmetic	add	`add $s1, $s2, $s3`	`$s1 = $s2 + $s3`	Three operands; data in registers
	subtract	`sub $s1, $s2, $s3`	`$s1 = $s2 - $s3`	Three operands; data in registers
	add immediate	`addi $s1, $s2, 100`	`$s1 = $s2 + 100`	Used to add constants
Data transfer	load word	`lw $s1, 100($s2)`	`$s1 = Memory[$s2 + 100]`	Word from memory to register
	store word	`sw $s1, 100($s2)`	`Memory[$s2 + 100] = $s1`	Word from register to memory
	load byte	`lb $s1, 100($s2)`	`$s1 = Memory[$s2 + 100]`	Byte from memory to register
	store byte	`sb $s1, 100($s2)`	`Memory[$s2 + 100] = $s1`	Byte from register to memory
	load upper immediate	`lui $s1, 100`	$s1 = 100 * 2^{16}	Loads constant in upper 16 bits
Conditional branch	branch on equal	`beq $s1, $s2, 25`	if (`$s1 == $s2`) go to PC + 4 + 100	Equal test; PC-relative branch
	branch on not equal	`bne $s1, $s2, 25`	if (`$s1 != $s2`) go to PC + 4 + 100	Not equal test; PC-relative
	set on less than	`slt $s1, $s2, $s3`	if (`$s2 < $s3`) `$s1 = 1`; else `$s1 = 0`	Compare less than; for beq, bne
	set less than immediate	`slti $s1, $s2, 100`	if (`$s2 < 100`) `$s1 = 1`; else `$s1 = 0`	Compare less than constant
Uncondi-tional jump	jump	`j 2500`	go to 10000	Jump to target address
	jump register	`jr $ra`	go to `$ra`	For switch, procedure return
	jump and link	`jal 2500`	`$ra = PC + 4`; go to 10000	For procedure call

MIPS instruction format

I-type instruction



R-type instruction



J-type instruction

9 Instructions

	Instruction bit number					
	31..26	25..21	20..16	15..11	10..6	5..0
add	000000	rs	rt	rd	00000	100000
sub	000000	rs	rt	rd	00000	100010
and	000000	rs	rt	rd	00000	100100
or	000000	rs	rt	rd	00000	100101
slt	000000	rs	rt	rd	00000	101010
lw	100011	rs	rt	immediate		
sw	101011	rs	rt	immediate		
beq	000100	rs	rt	immediate		
j	000010	address				

R-instr. Add / Sub

- `add $17, $18, $19 # $17=$18+$19`

6-bit	5-bit	5-bit	5-bit	5-bit	6-bit
op	rs	rt	rd	shamt	funct
0	18	19	17	0	32
000000	10010	10011	10001	00000	100000

- `sub $17, $18, $19 # $17=$18−$19`

6-bit	5-bit	5-bit	5-bit	5-bit	6-bit
op	rs	rt	rd	shamt	funct
0	18	19	17	0	34
000000	10010	10011	10001	00000	100010

R-Instruction: And/Or

- `and $17, $18, $19 # $17=$18 AND $19`

6-bit	5-bit	5-bit	5-bit	5-bit	6-bit
op	rs	rt	rd	shamt	funct
0	18	19	17	0	36
000000	10010	10011	10001	00000	100100

- `or $17, $18, $19 # $17=$18 OR $19`

6-bit	5-bit	5-bit	5-bit	5-bit	6-bit
op	rs	rt	rd	shamt	funct
0	18	19	17	0	37
000000	10010	10011	10001	00000	100101

I-Instruction: LW/SW

- `lw $17, 100($18) # $17=Memory[$18+100]`

6-bit	5-bit	5-bit	16-bit
op	rs	rt	Address
35	18	17	100

`100011 10010 10001 0000 0000 0110 0100`

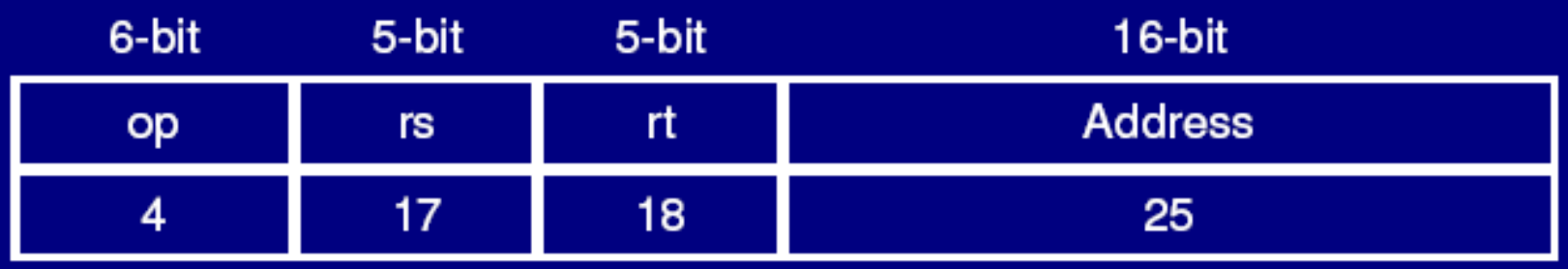
- `sw $17, 100($18) # $Memory[$18+100]=$17`

6-bit	5-bit	5-bit	16-bit
op	rs	rt	Address
43	18	17	100

`101011 10010 10001 0000 0000 0110 0100`

Branch / Jump

- `beq $17, $18, 25 # if $17=$18, goto PC+4+25*4`



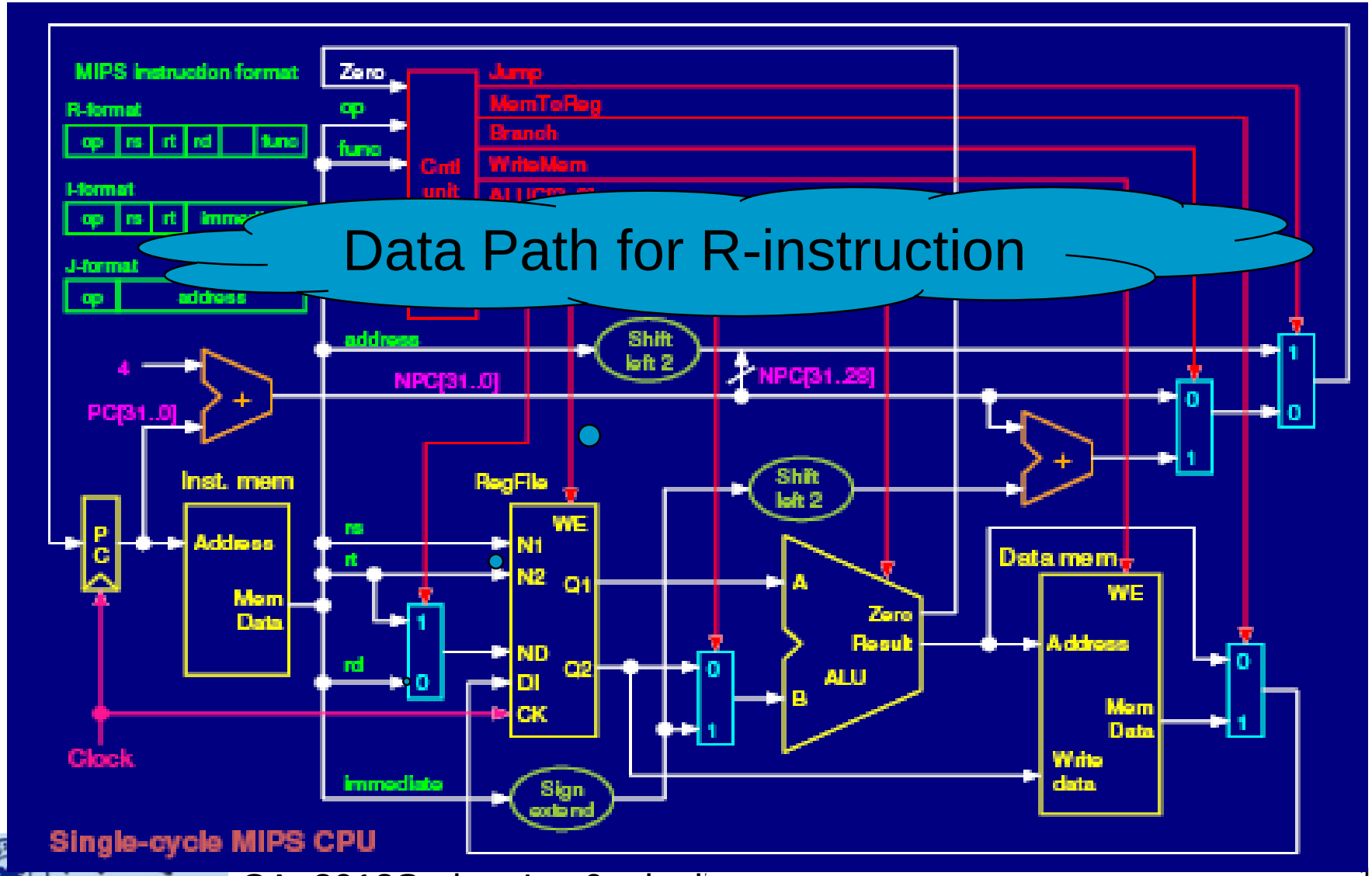
- `j 10000 # jump to 10000`

SLT: set when less than

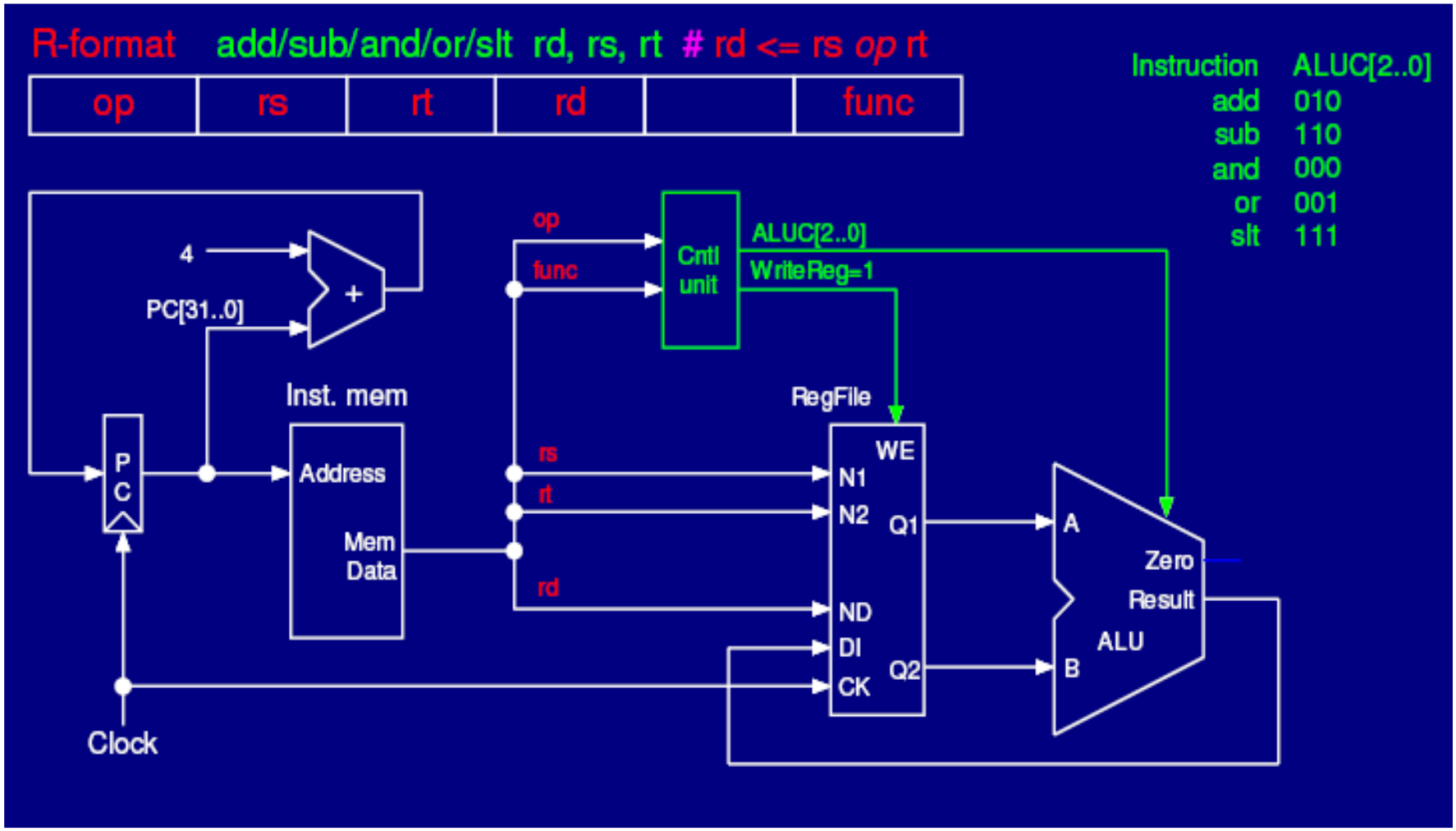
- `slt $17, $18, $19 # if $18<$19, $17=1; else $17=0`

6-bit	5-bit	5-bit	5-bit	5-bit	6-bit
op	rs	rt	rd	shamt	funct
0	18	19	17	0	42
000000	10010	10011	10001	00000	101010

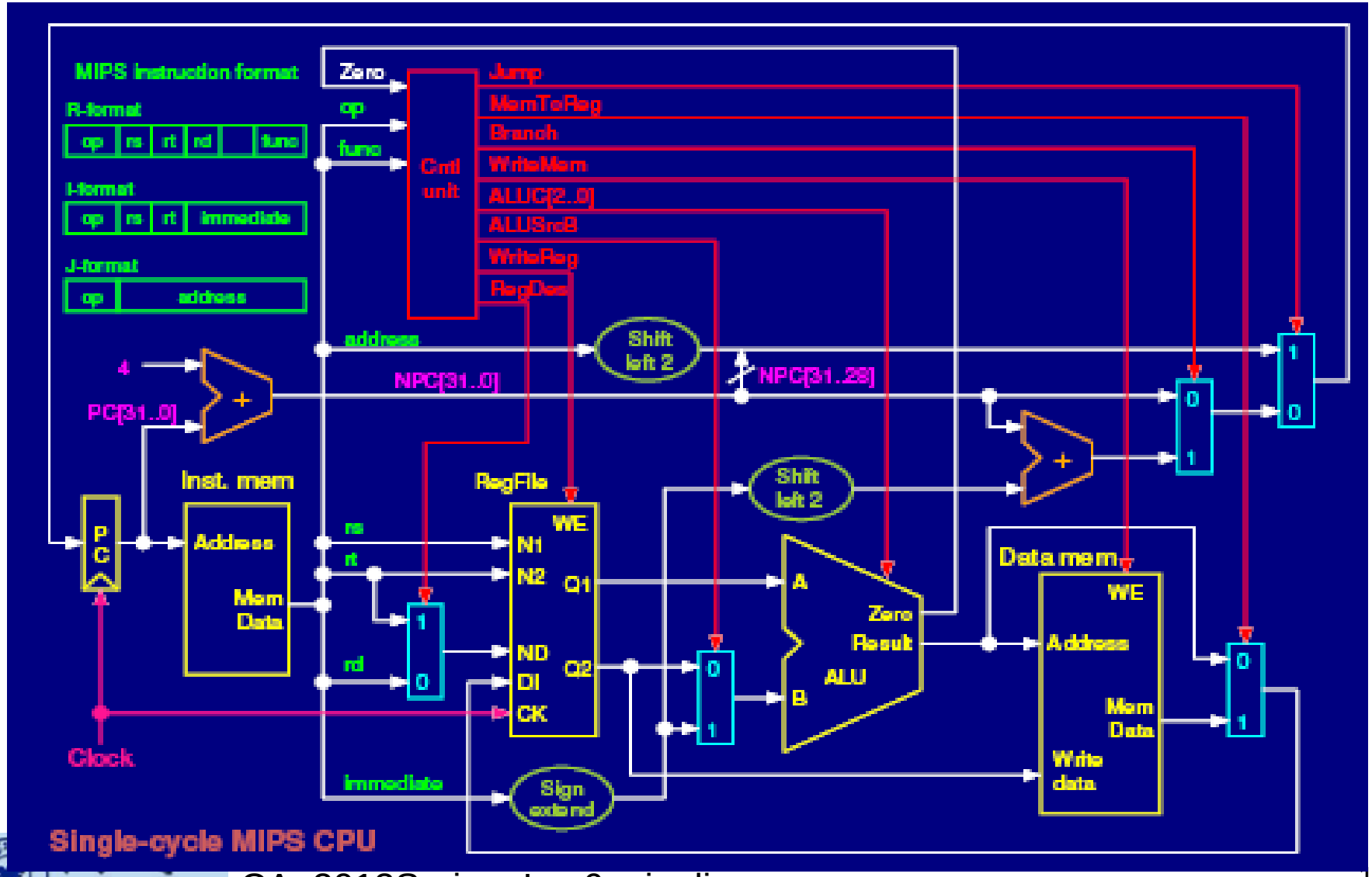
Single-cycle implementation



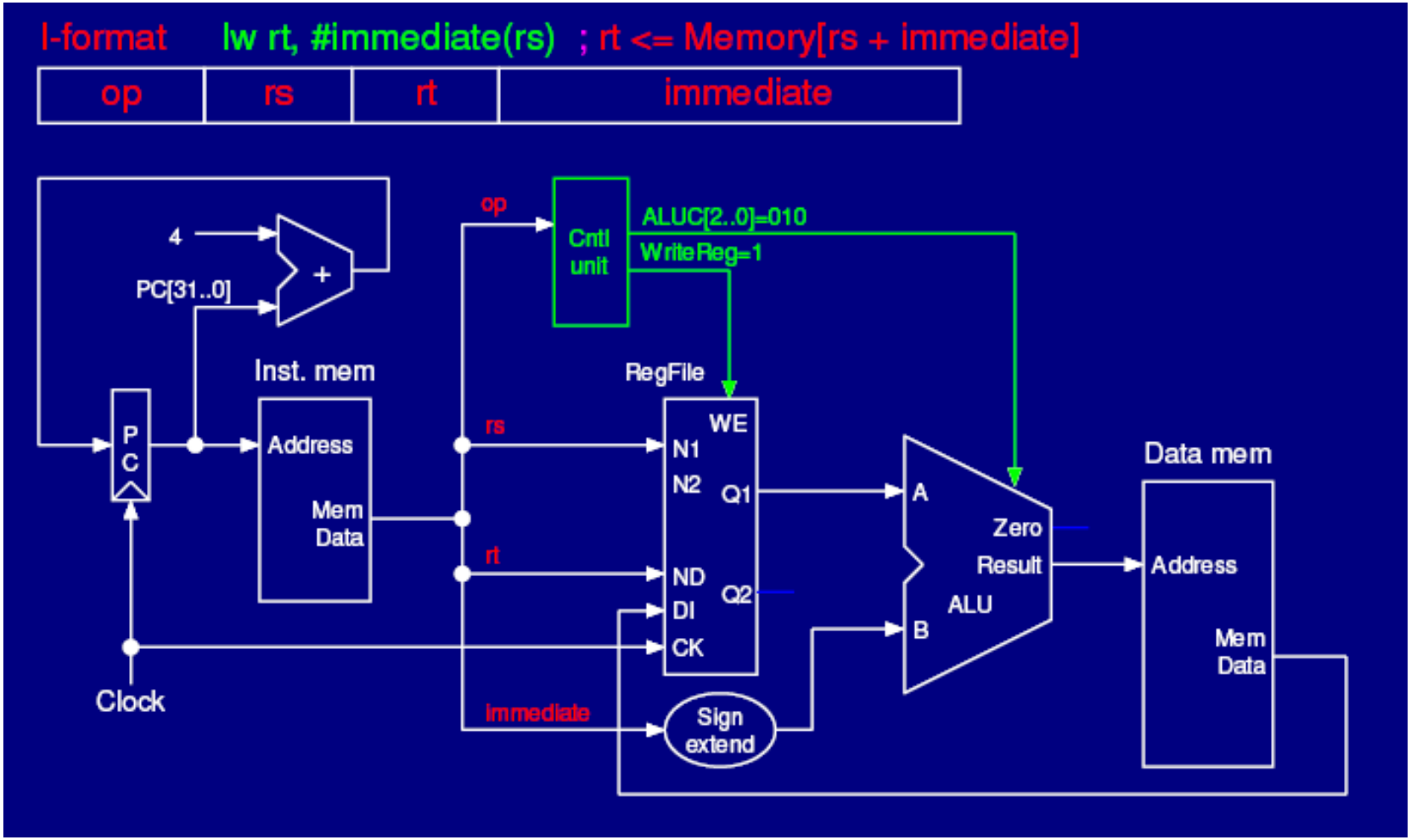
R-instruction Data Path



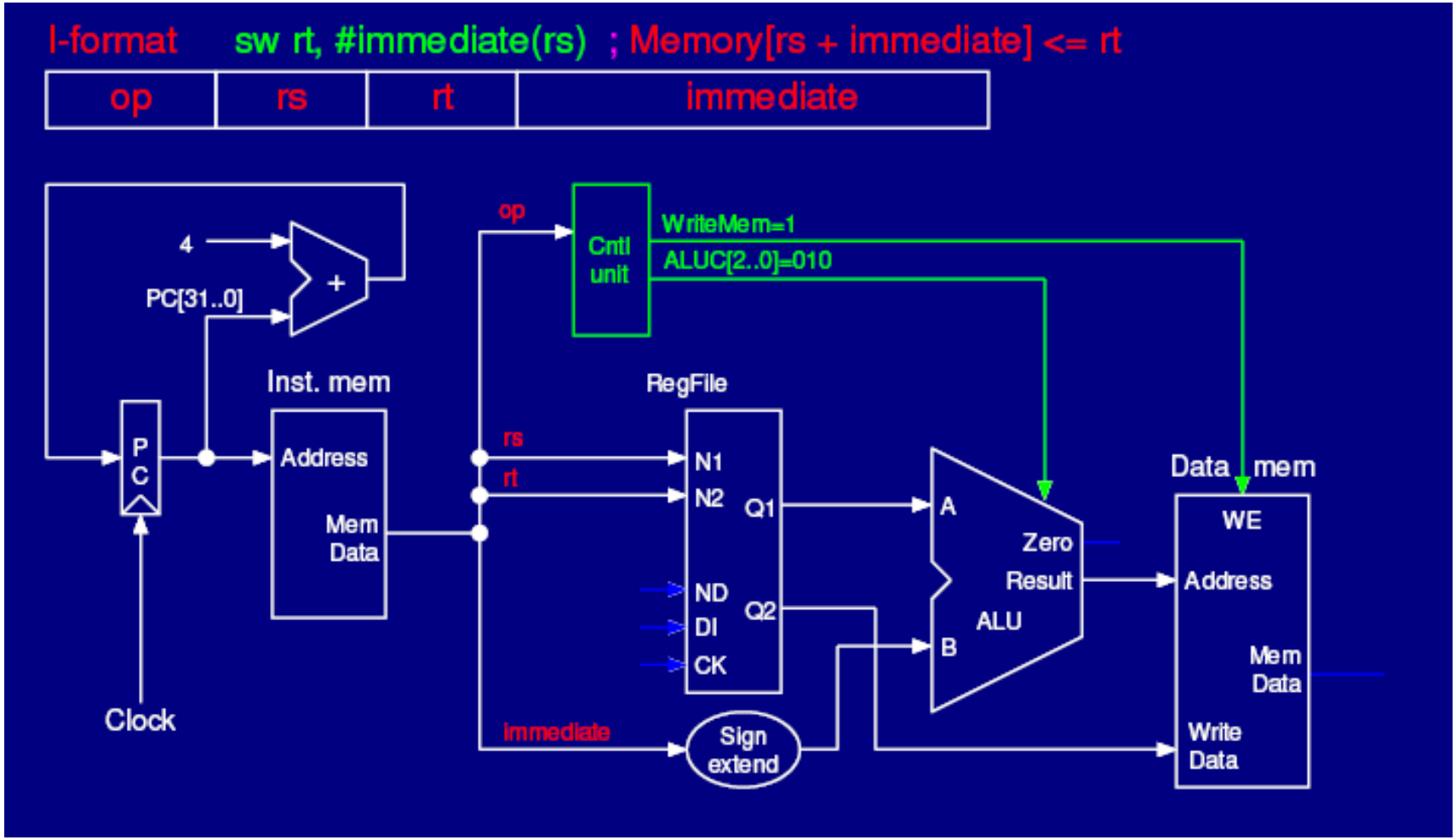
What's the data path for I-instruction (R-I, LW/SW/BEQ)?



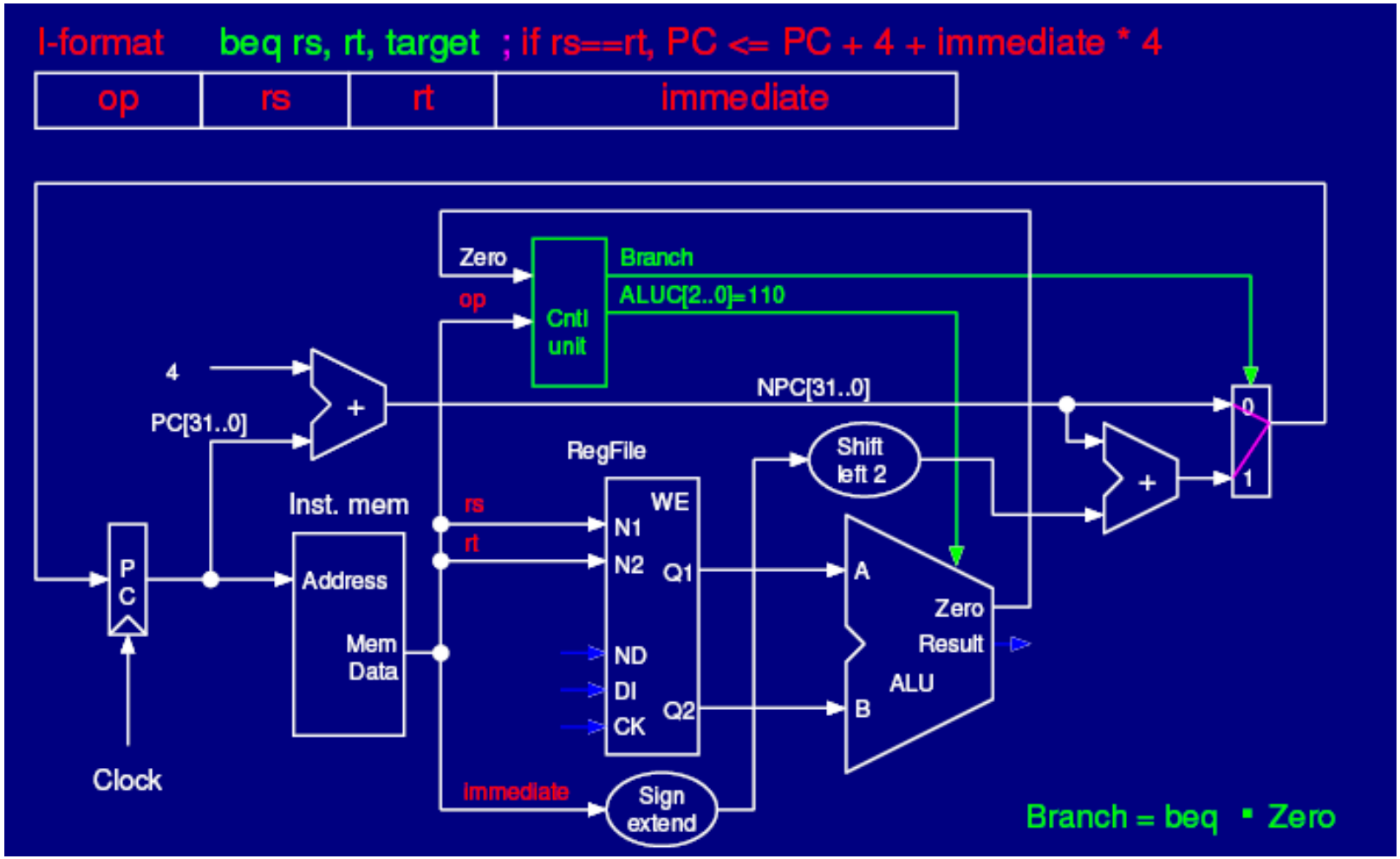
Data Path for LW



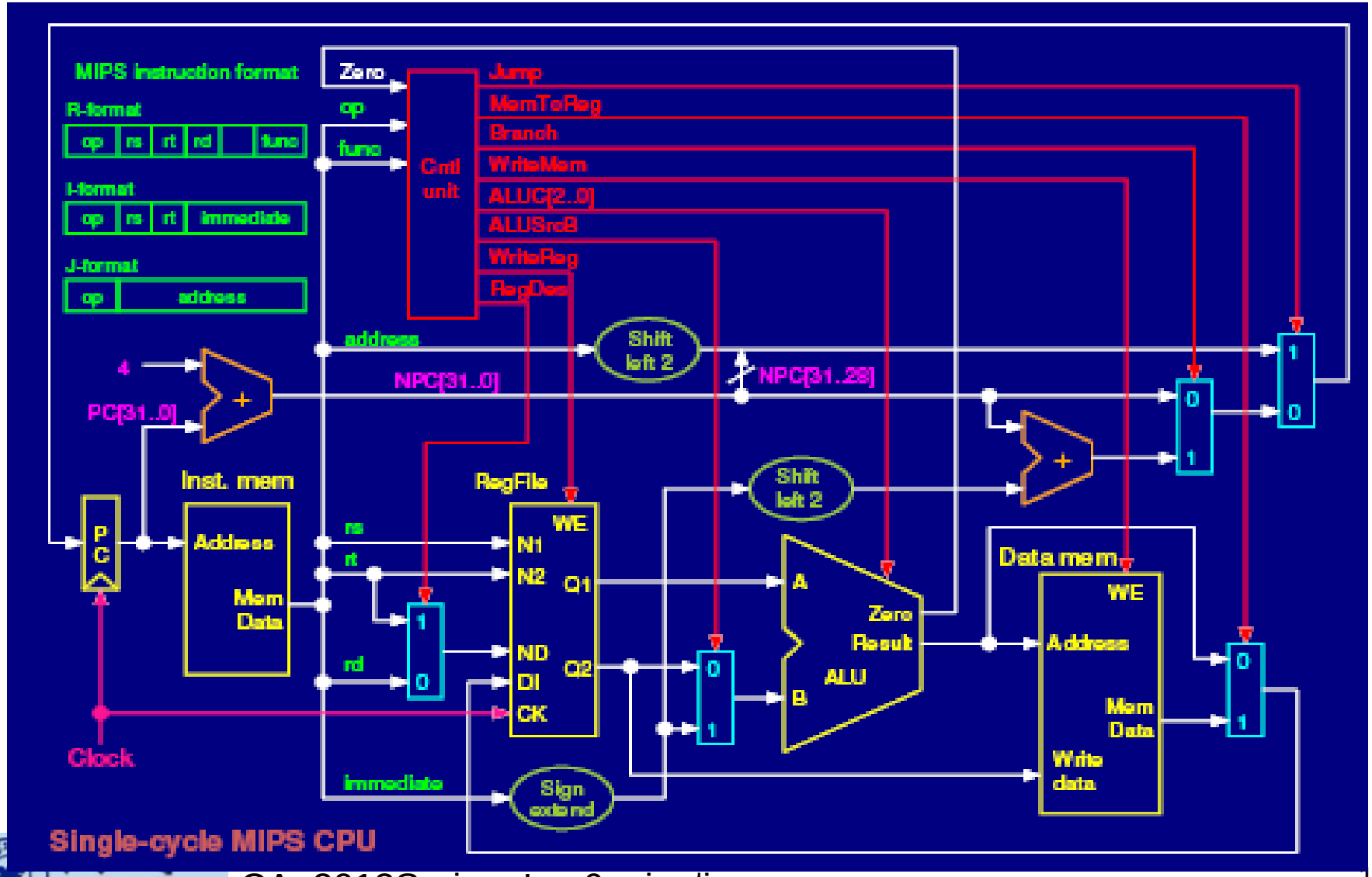
Data Path for SW



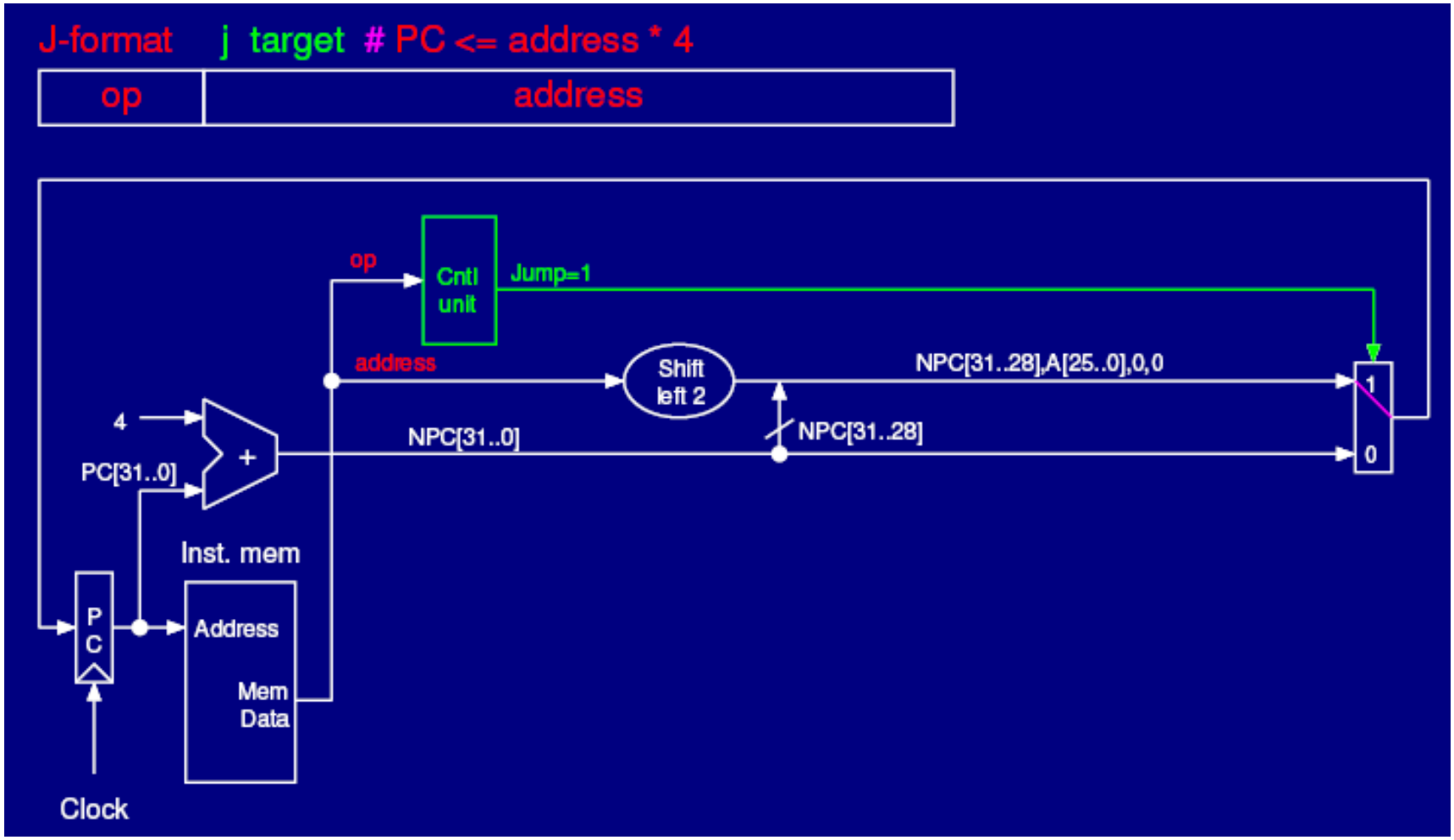
Data Path for BEQ

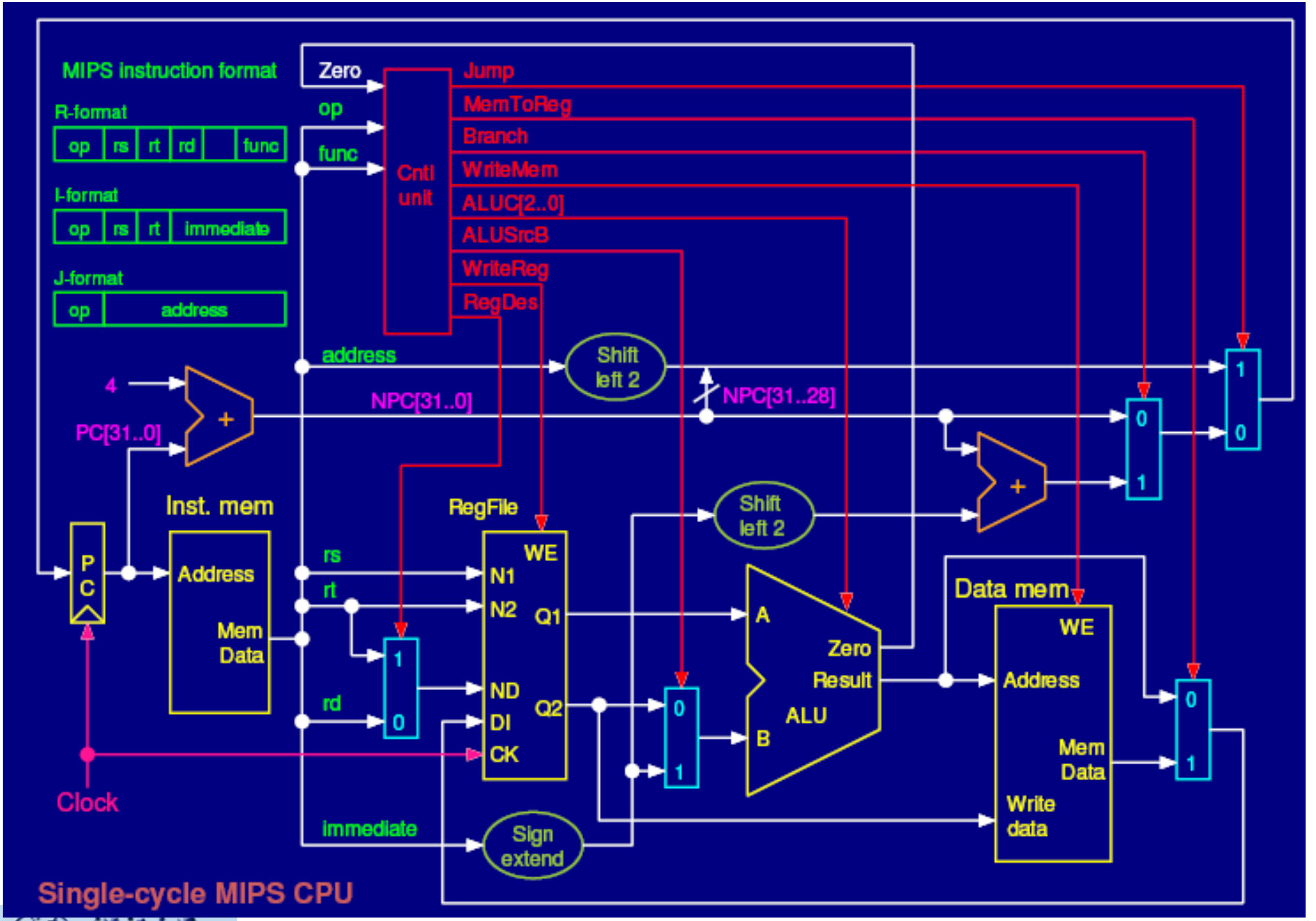


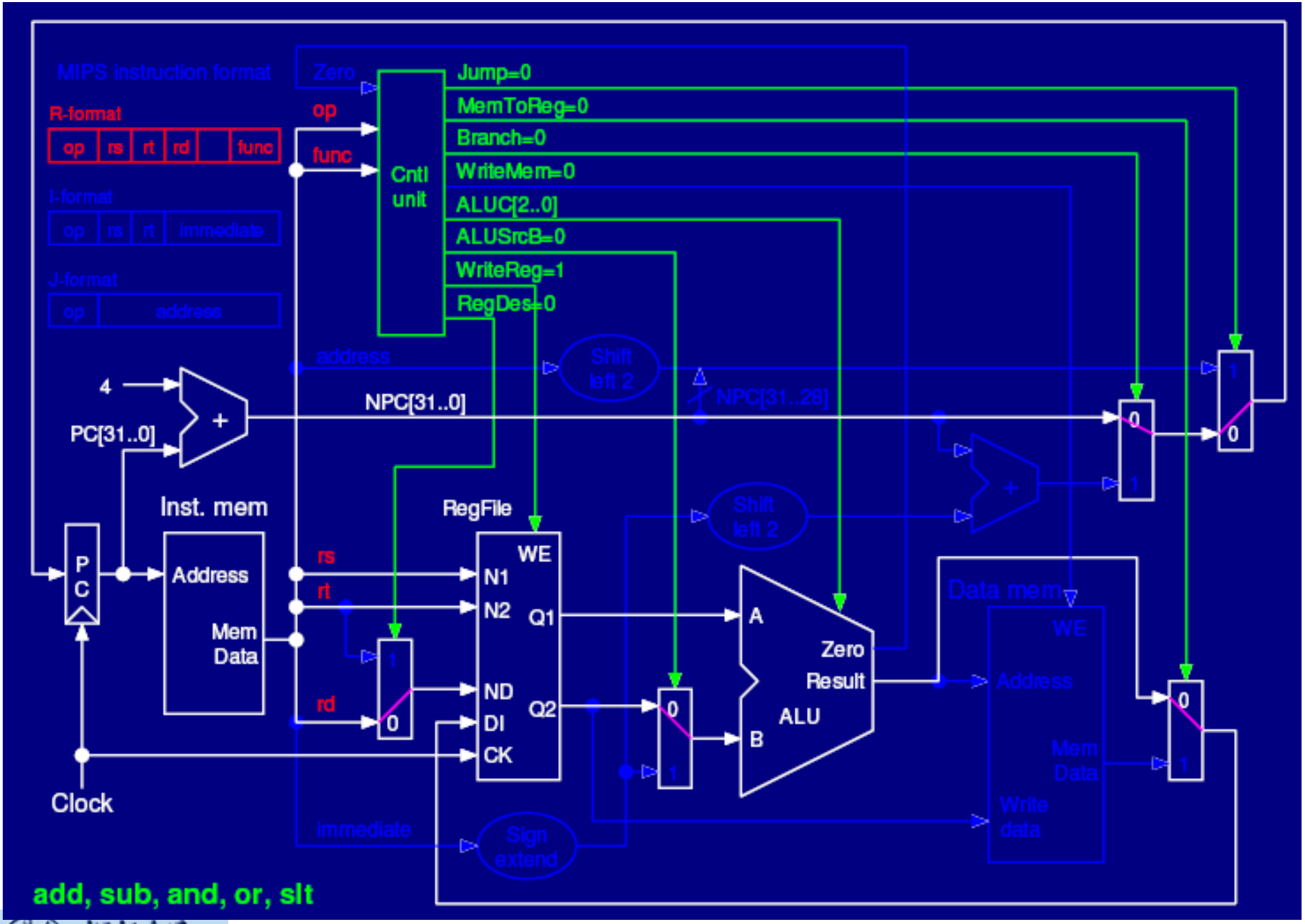
What's the data path for J-instruction (jump)?

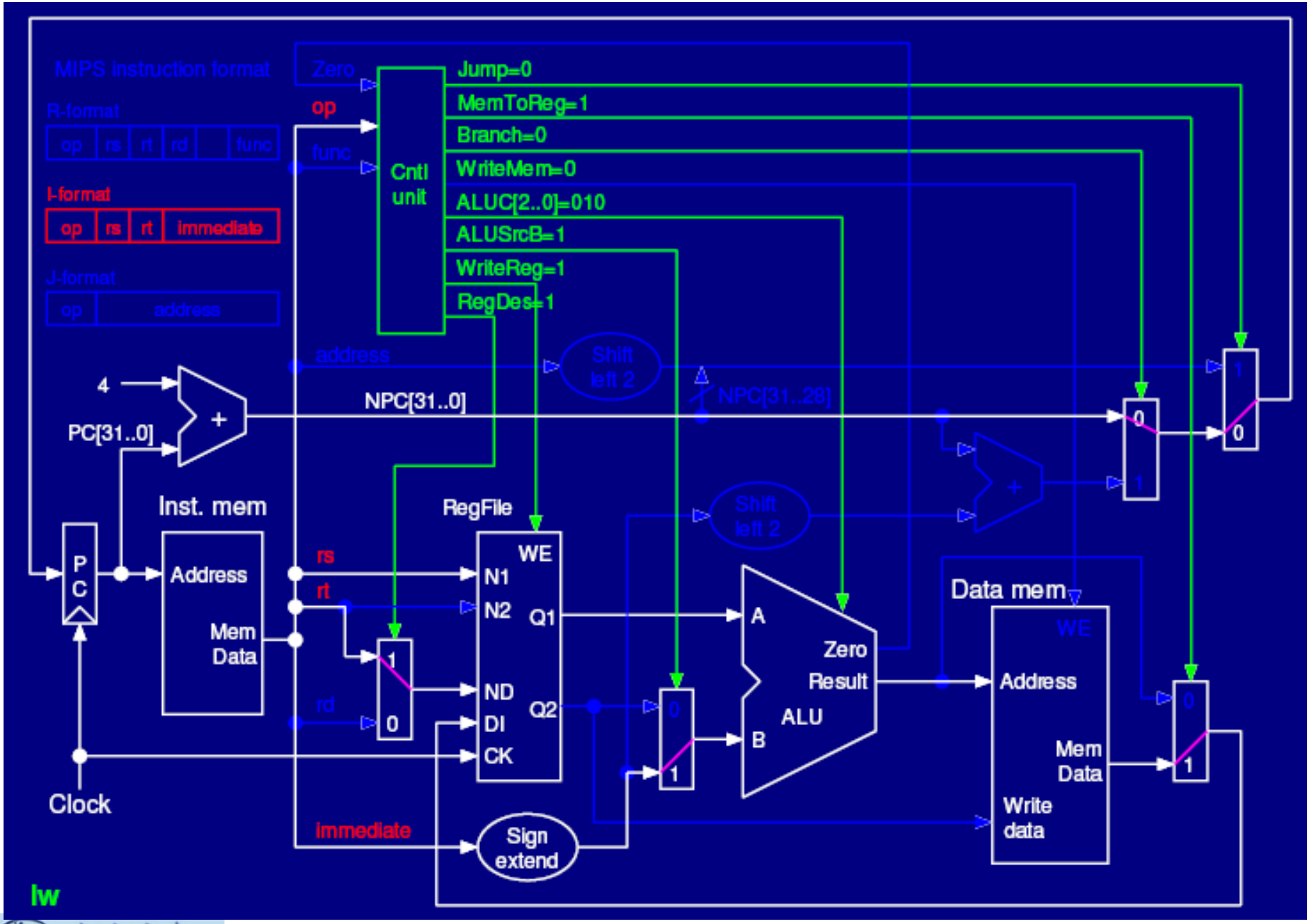


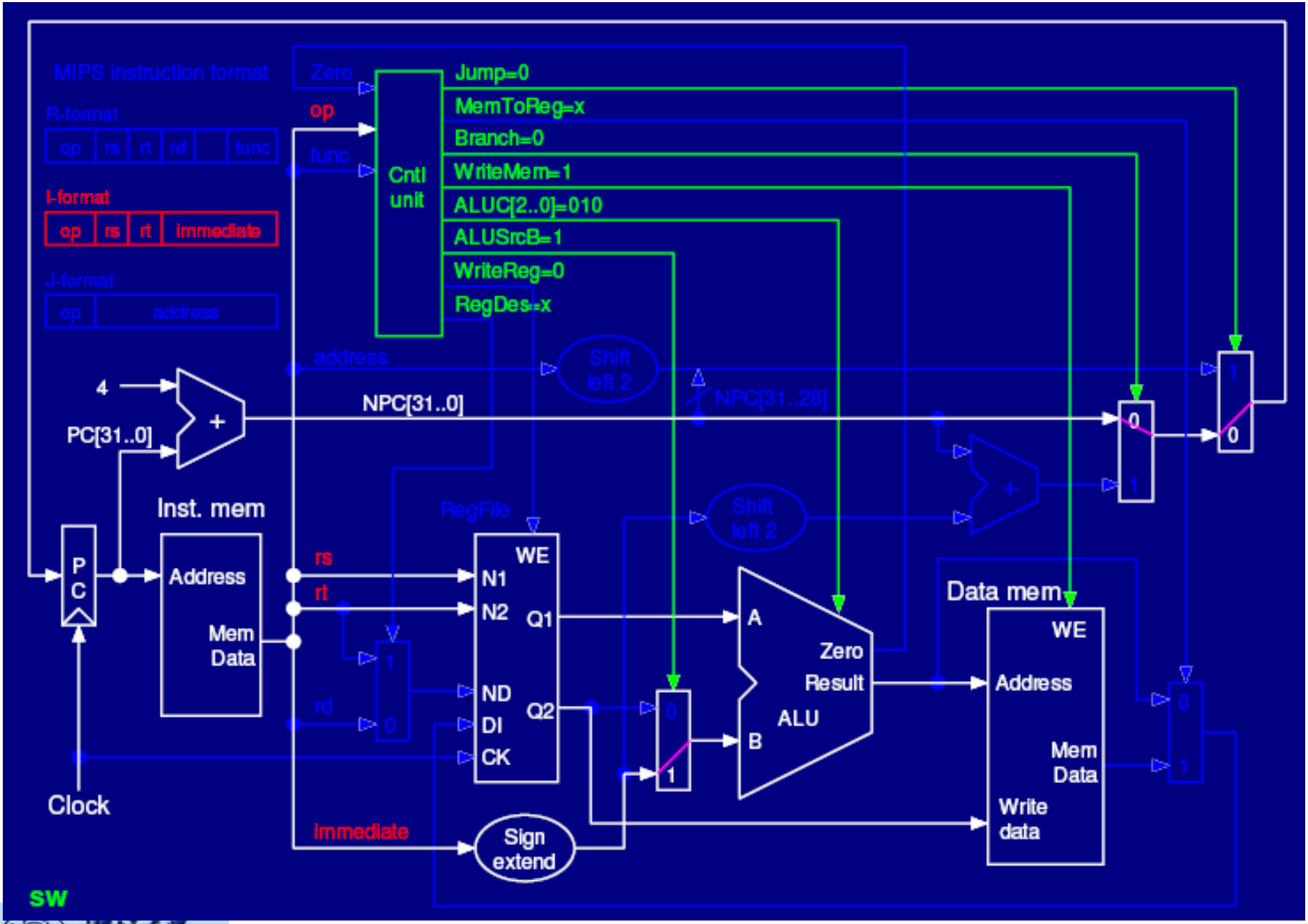
Data Path for Jump

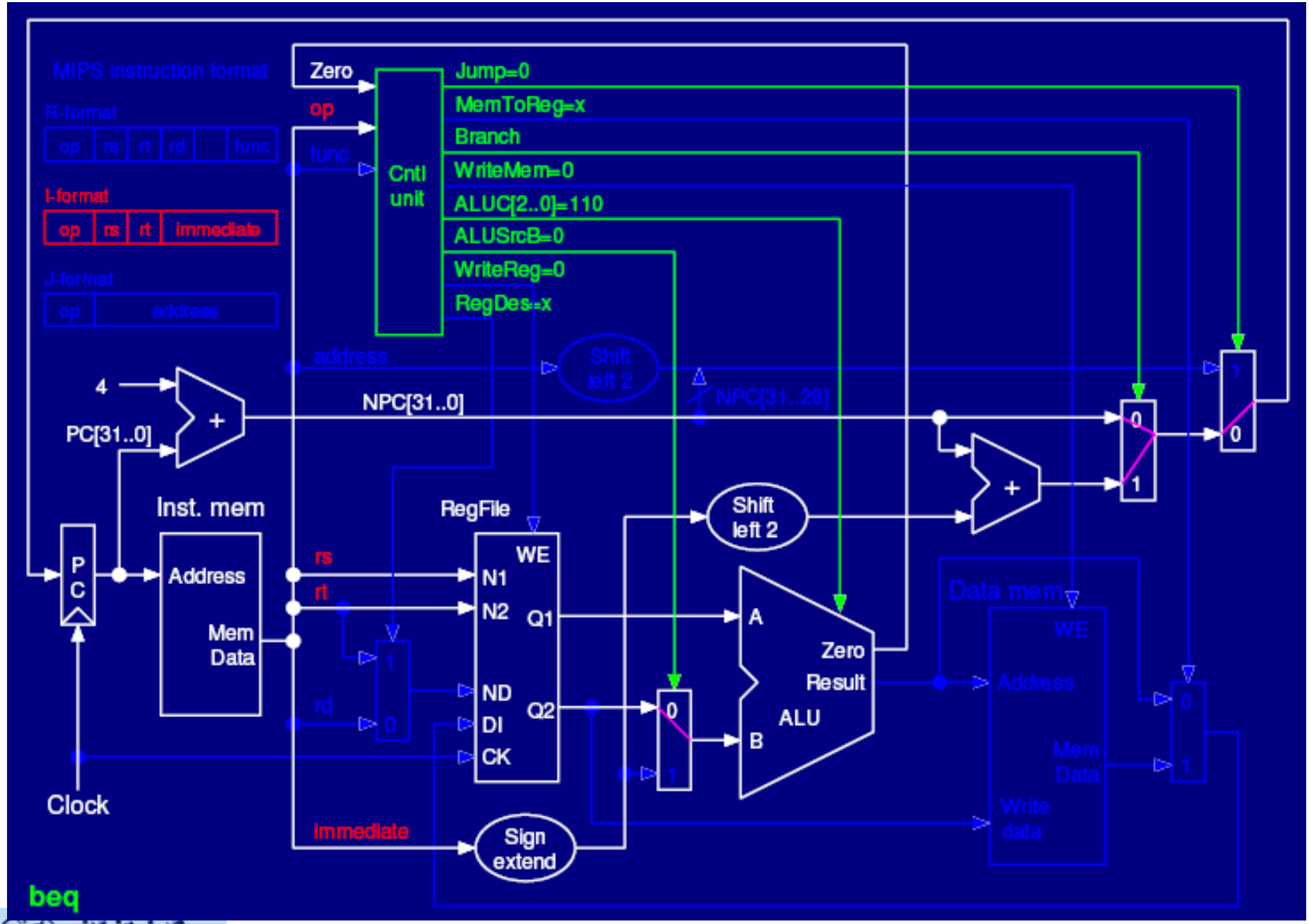


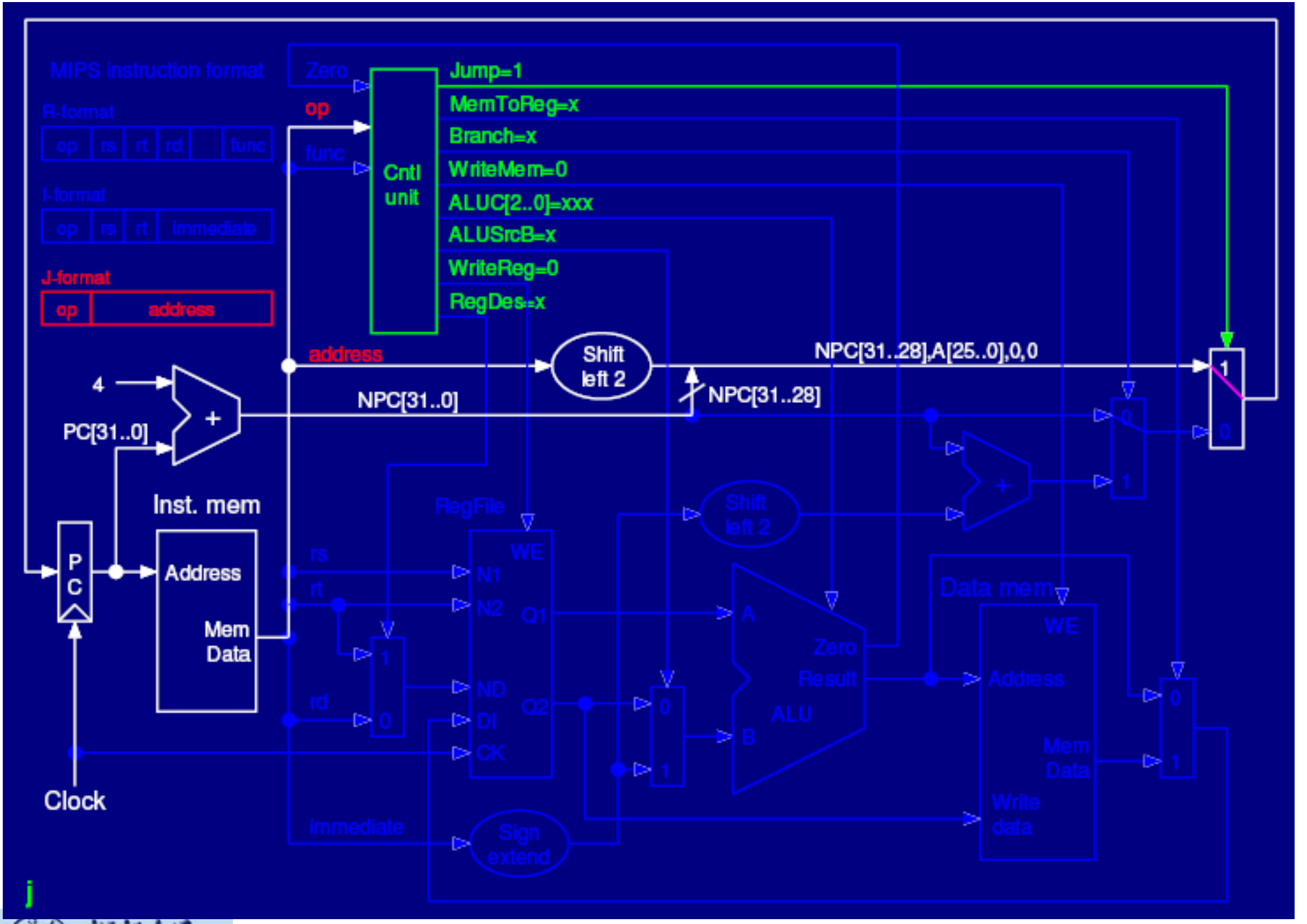

MIPS instruction format
R-format
op rs rt rd func
I-format
op rs rt immediate
J-format
op address
Zero
op
func
Cntl unit
Jump
MemToReg
Branch
WriteMem
ALUC[2..0]
ALUSrcB
WriteReg
RegDes
address
Shift left 2
NPC[31..0]
NPC[31..28]
4
PC[31..0]
+
Inst. mem
Address
Mem Data
PC
RegFile
WE
N1
N2
Q1
ND
DI
Q2
CK
rs
rt
rd
1
0
Shift left 2
A
B
ALU
Zero
Result
Data mem
WE
Address
Mem Data
Write data
Clock
immediate
Sign extend
Single-cycle MIPS CPU


MIPS instruction format
R-format
op rs rt rd func
I-format
op rs rt immediate
J-format
op address
Zero
op
func
Cntl unit
Jump=0
MemToReg=0
Branch=0
WriteMem=0
ALUC[2..0]
ALUSrcB=0
WriteReg=1
RegDes=0
address
Shift left 2
NPC[31..28]
4
PC[31..0]
NPC[31..0]
Inst. mem
RegFile
Shift left 2
Data mem
PC
Address
Mem Data
rs
rt
rd
WE
N1
N2
Q1
ND
DI
Q2
CK
A
B
ALU
Zero
Result
WE
Address
Mem Data
Write data
Clock
immediate
Sign extend
add, sub, and, or, slt

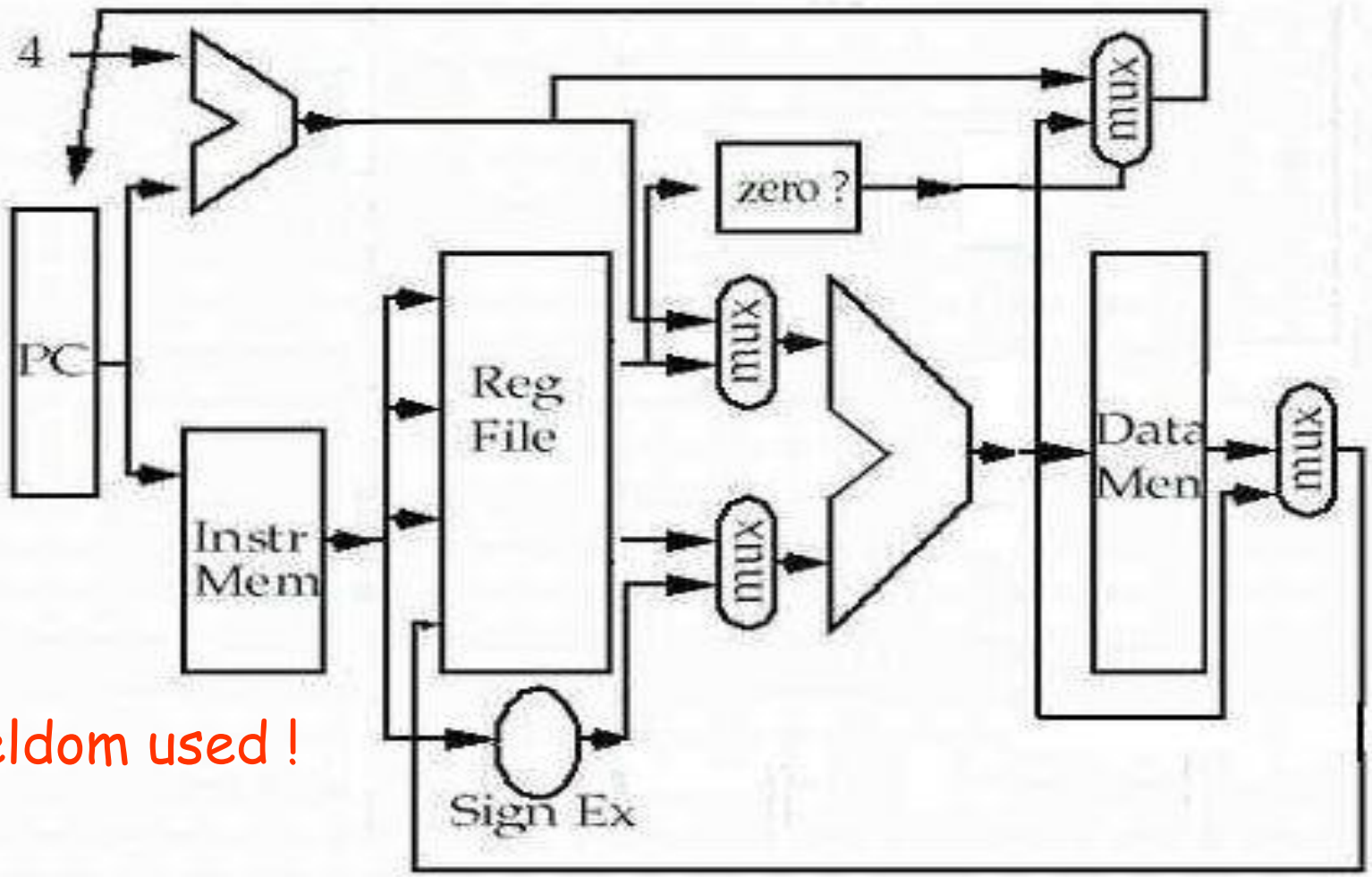

MIPS instruction format
R-format
op rs rt rd func
I-format
op rs rt immediate
J-format
op address
Zero
op
func
Cntl unit
Jump=0
MemToReg=1
Branch=0
WriteMem=0
ALUC[2..0]=010
ALUSrcB=1
WriteReg=1
RegDes=1
address
Shift left 2
NPC[31..28]
4
PC[31..0]
+
NPC[31..0]
Inst. mem
RegFile
Shift left 2
P C
Address
Mem Data
rs
rt
rd
WE
N1
N2
Q1
ND
DI
Q2
CK
A
Zero
Result
ALU
B
Data mem
WE
Address
Mem Data
Write data
Clock
immediate
Sign extend
lw


MIPS instruction format
R-format
op rs rt rd func
I-format
op rs rt immediate
J-format
op address
Zero
op
func
Cntl unit
Jump=0
MemToReg=x
Branch=0
WriteMem=1
ALUC[2..0]=010
ALUSrcB=1
WriteReg=0
RegDes=x
address
Shift left 2
NPC[31..28]
4
PC[31..0]
NPC[31..0]
Inst. mem
RegFile
Shift left 2
P C
Address
Mem Data
rs
rt
rd
WE
N1
N2
Q1
ND
DI
Q2
CK
A
Zero
Result
ALU
B
Data mem
WE
Address
Mem Data
Write data
Clock
immediate
Sign extend
sw


MIPS instruction format
R-format
op rs rt rd func
I-format
op rs rt immediate
J-format
op address
Zero
op
func
Cntl unit
Jump=0
MemToReg=x
Branch
WriteMem=0
ALUC[2..0]=110
ALUSrcB=0
WriteReg=0
RegDes=x
address
Shift left 2
NPC[31..28]
4
PC[31..0]
NPC[31..0]
Inst. mem
RegFile
Shift left 2
PC
Address
Mem Data
rs
rt
rd
WE
N1
N2
Q1
ND
DI
Q2
CK
A
Zero
Result
ALU
B
Data mem
WE
Address
Mem Data
Write data
Clock
immediate
Sign extend
beq

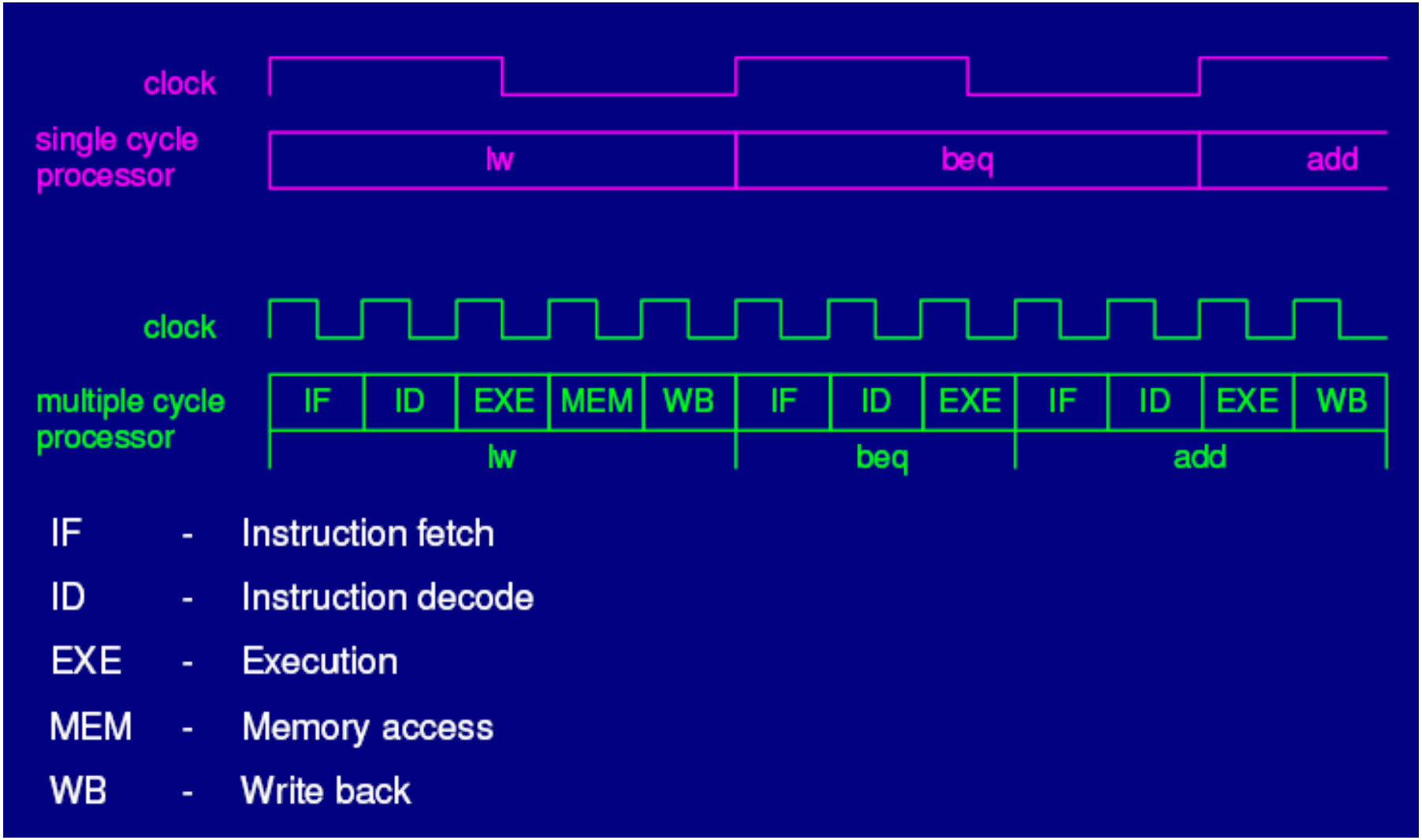

MIPS instruction format
R-format
op
rs
rt
rd
func
I-format
op
rs
rt
immediate
J-format
op
address
Zero
op
func
Cntl unit
Jump=1
MemToReg=x
Branch=x
WriteMem=0
ALUC[2..0]=xxx
ALUSrcB=x
WriteReg=0
RegDes=x
address
Shift left 2
NPC[31..28],A[25..0],0,0
4
PC[31..0]
+
NPC[31..0]
NPC[31..28]
Inst. mem
Address
Mem Data
PC
RegFile
WE
N1
N2
Q1
ND
DI
Q2
CK
rs
rt
rd
Shift left 2
A
B
ALU
Zero
Result
Data mem
WE
Address
Mem Data
Write data
immediate
Sign extend
Clock
j

Single-cycle implementation

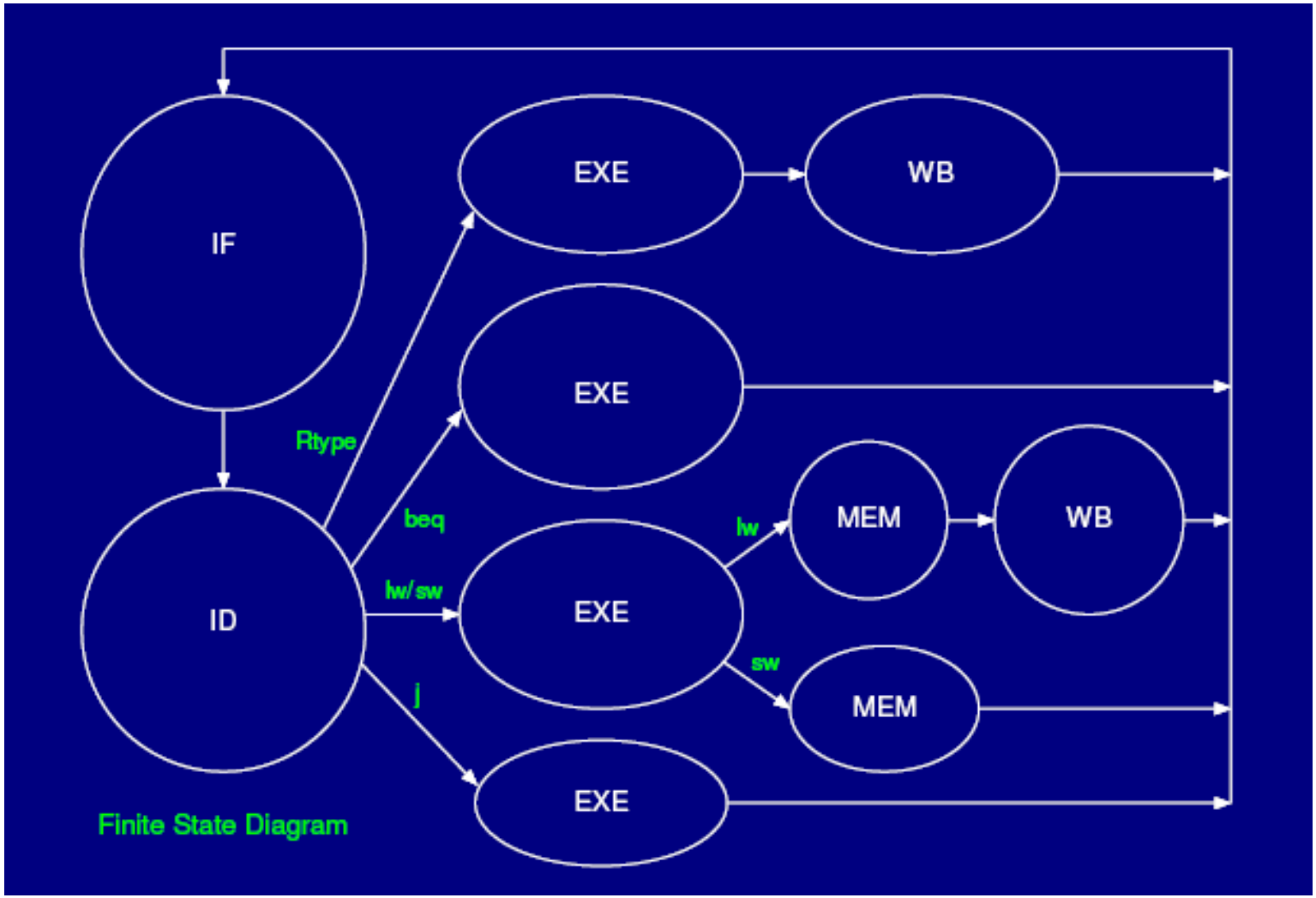


seldom used !

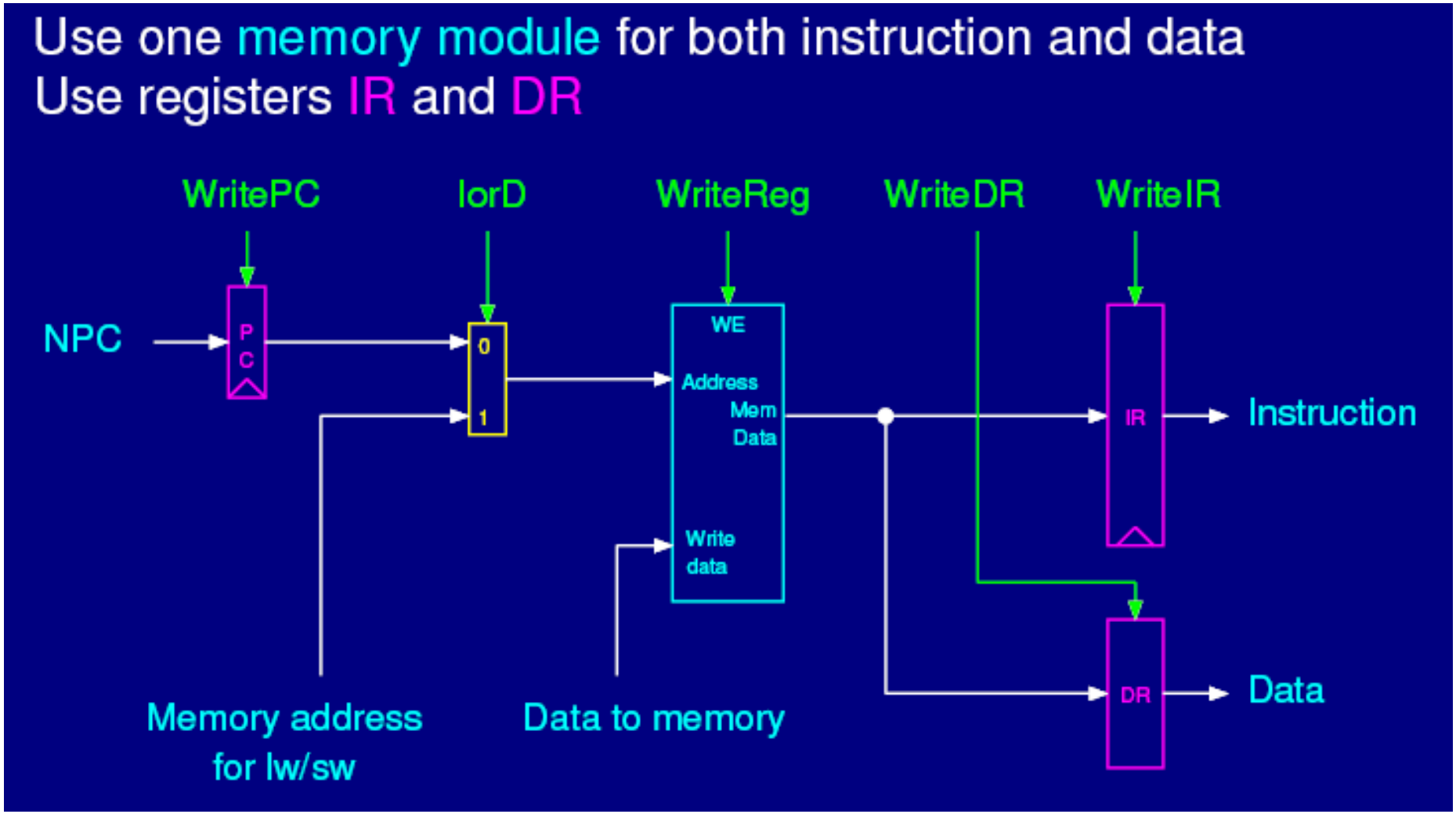
Single cycle → Multiple cycle



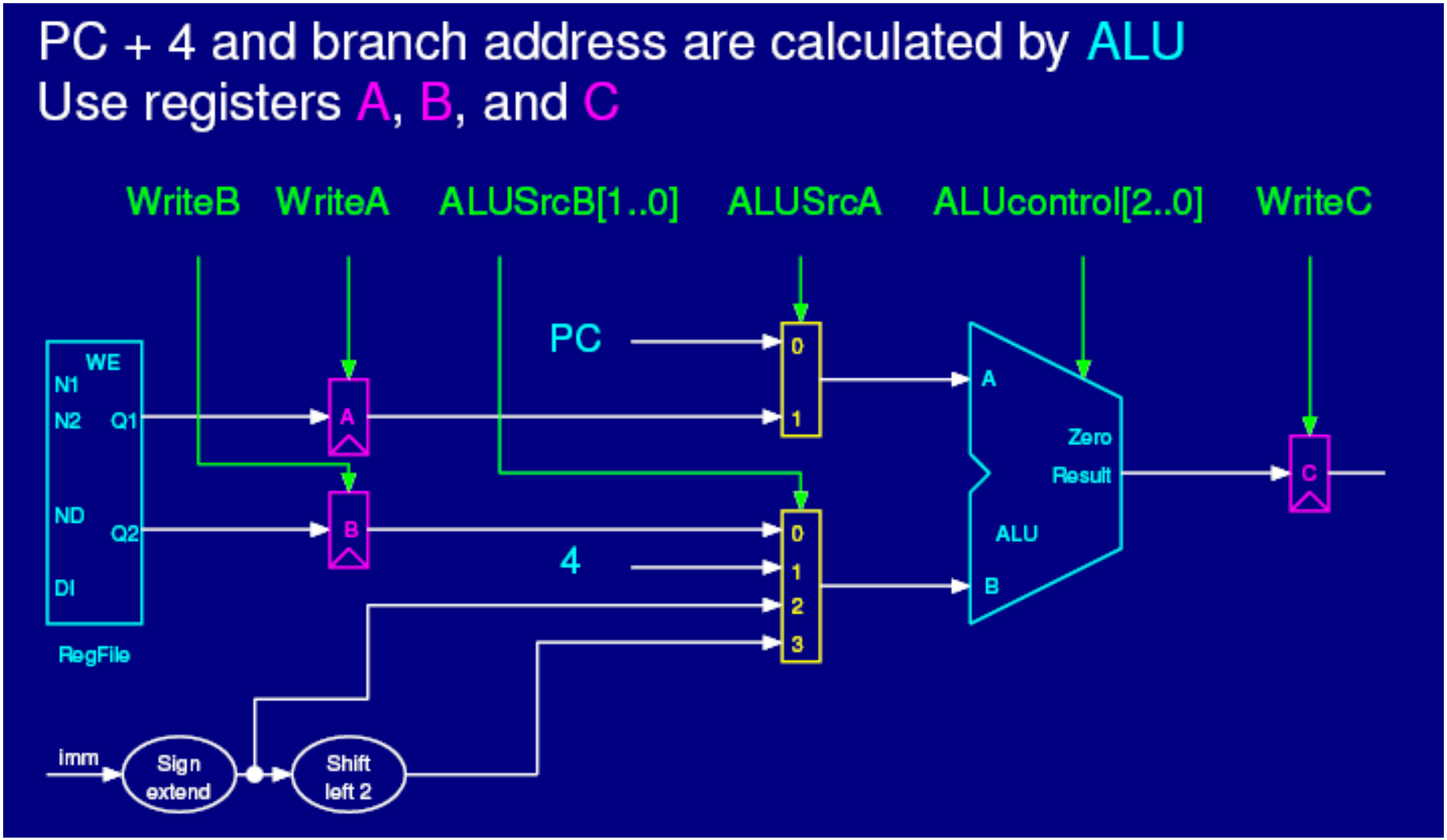
Finite State Diagram

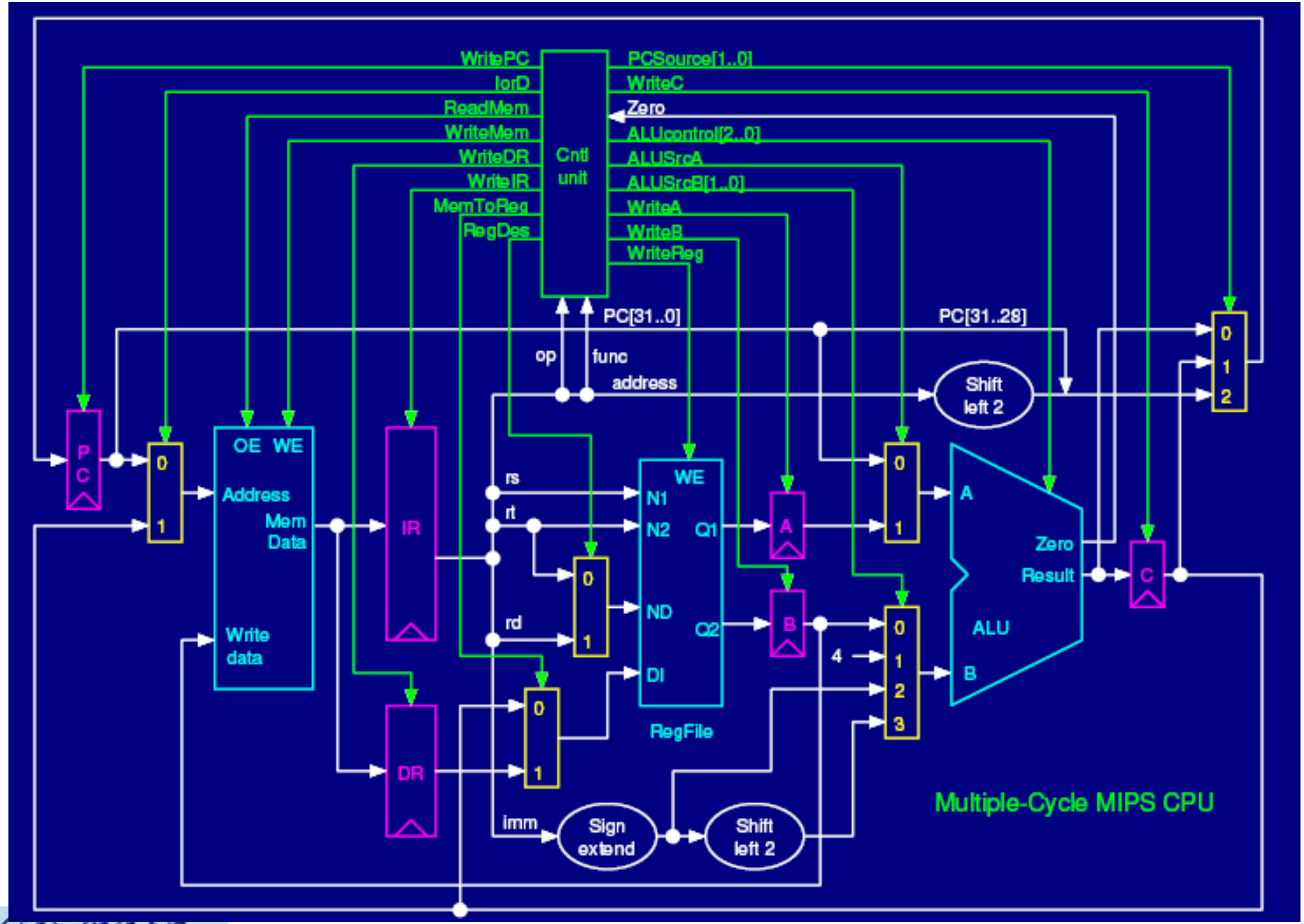


Multiple Cycle MIPS CPU



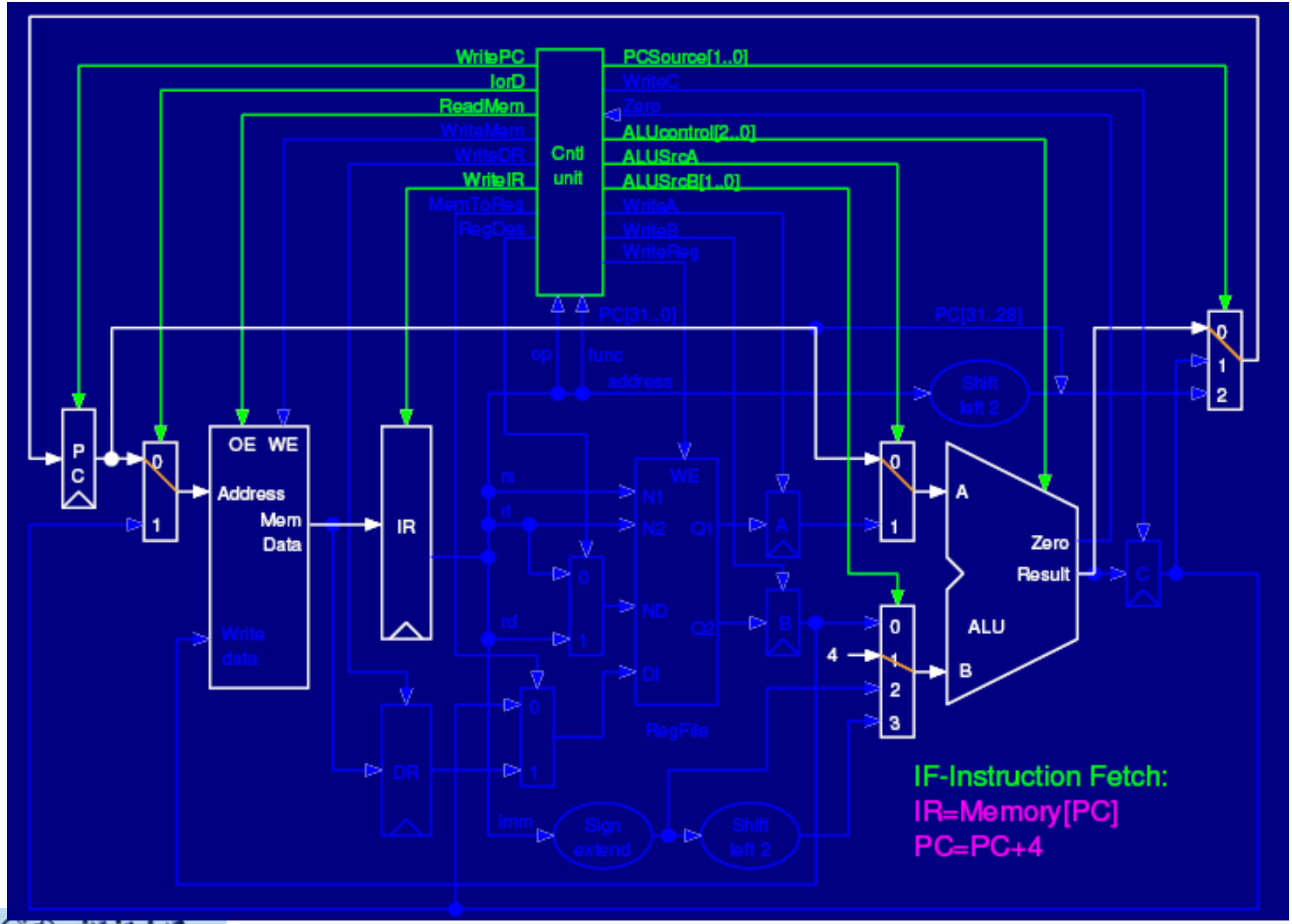
Multiple Cycle MIPS CPU

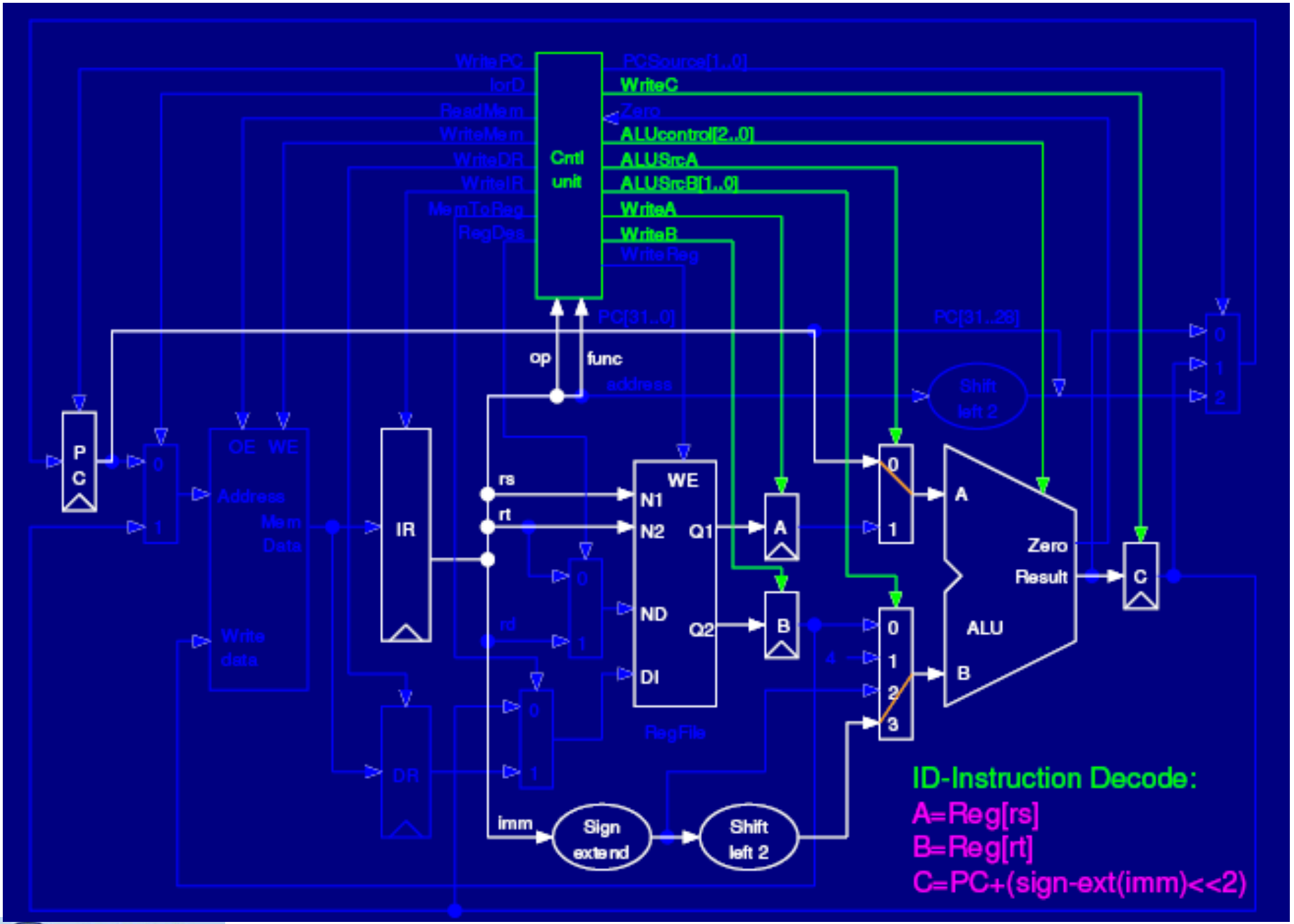


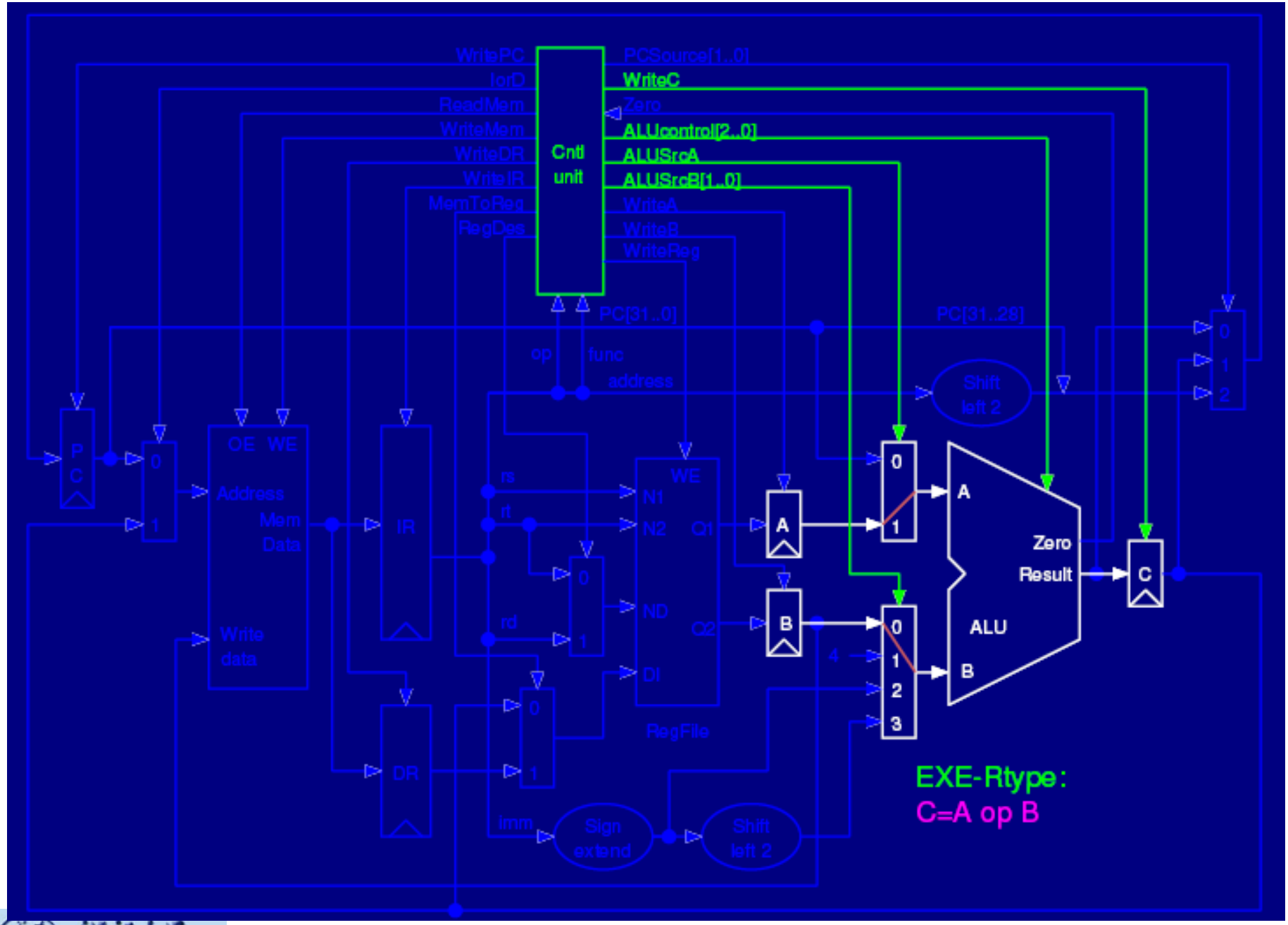

WritePC
IorD
ReadMem
WriteMem
WriteDR
WriteIR
MemToReg
RegDes
Cntl unit
PCSource[1..0]
WriteC
Zero
ALUcontrol[2..0]
ALUSrcA
ALUSrcB[1..0]
WriteA
WriteB
WriteReg
PC[31..0]
PC[31..28]
op
func
address
Shift left 2
PC
OE WE
Address
Mem Data
Write data
IR
DR
rs
rt
rd
WE
N1
N2
Q1
ND
Q2
DI
RegFile
A
B
4
ALU
Zero
Result
C
imm
Sign extend
Shift left 2
Multiple-Cycle MIPS CPU

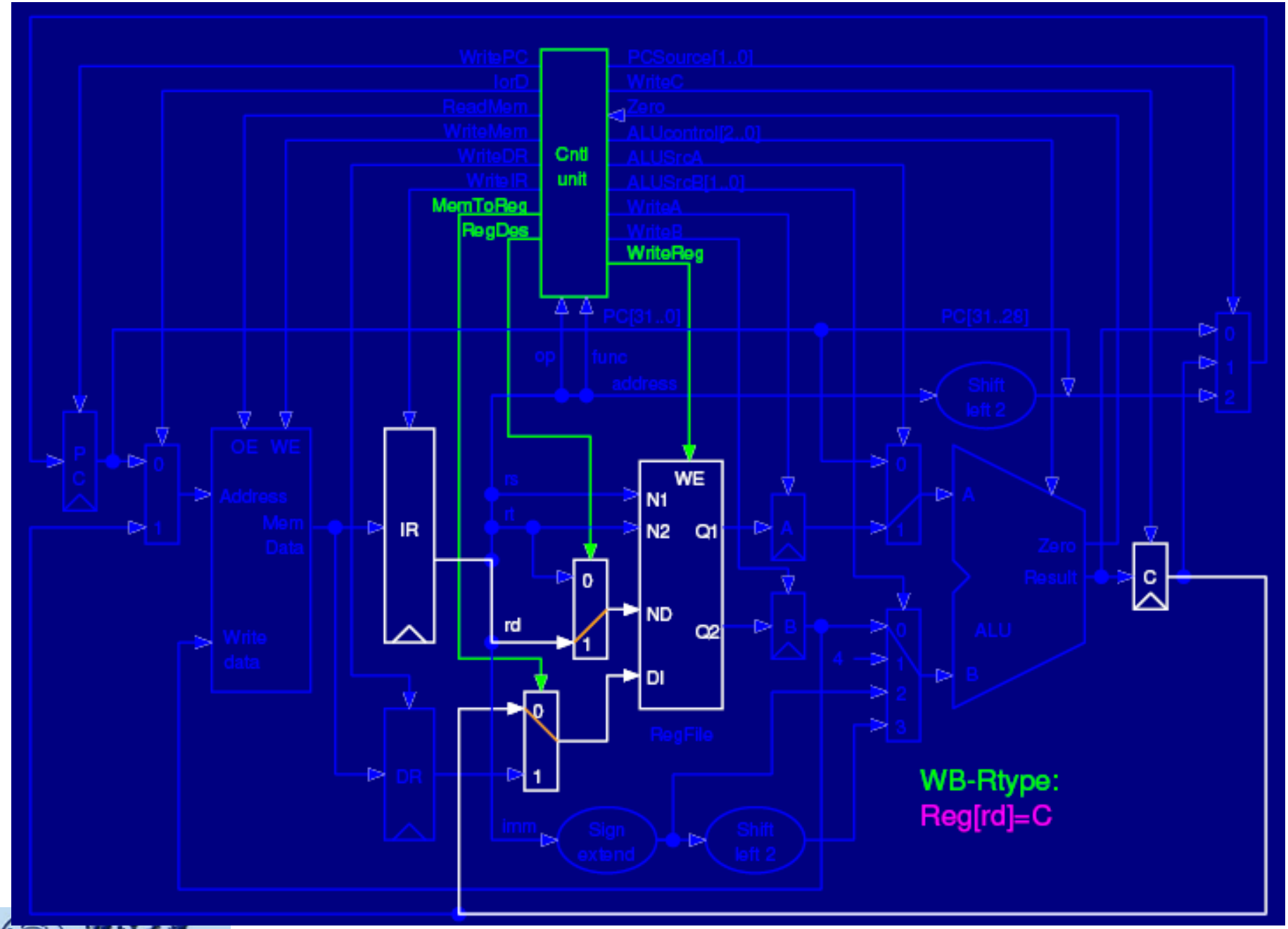
5-cycle MIPS CPU

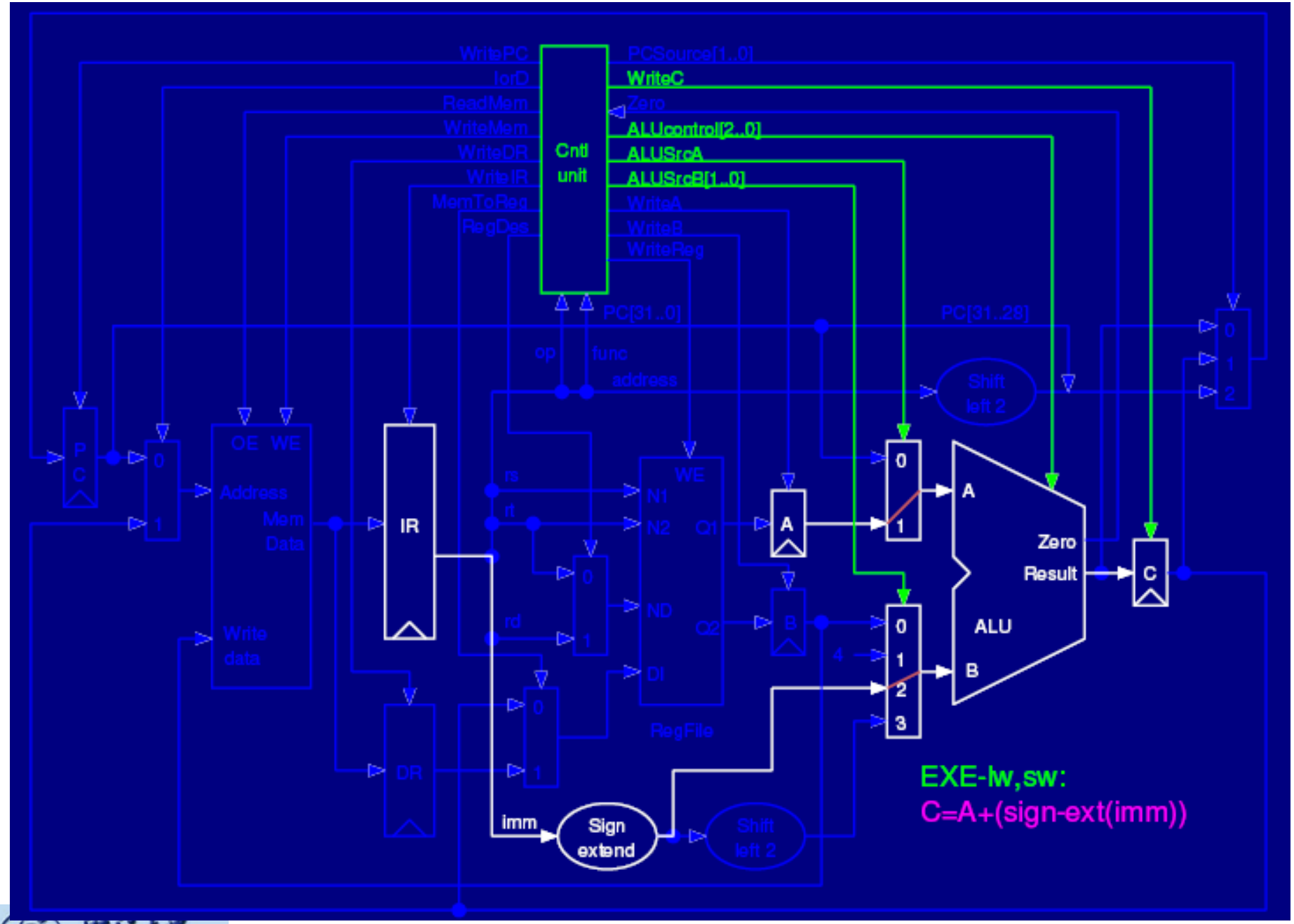
Steps	Rtype	lw/sw	beq	j
Instruction fetch (IF)	IR=Memory[PC] PC=PC+4			
Instruction decode and register fetch (ID)	A=Reg[rs] B=Reg[rt] C=PC+(sign-extend(imm)<<2)			
Execution (EXE)	C=A op B	C=A+ sign-extend(imm)	if (A−B)==0 then PC=C	PC=PC[31-28]|| (address<<2)
Memory access (MEM) or Rtype completion (WB)	Reg[rd]=C (WB)	lw: DR=Memory[C] sw: Memory[C]=B		
Memory access completion (WB)		lw: Reg[rt]=DR (WB)		

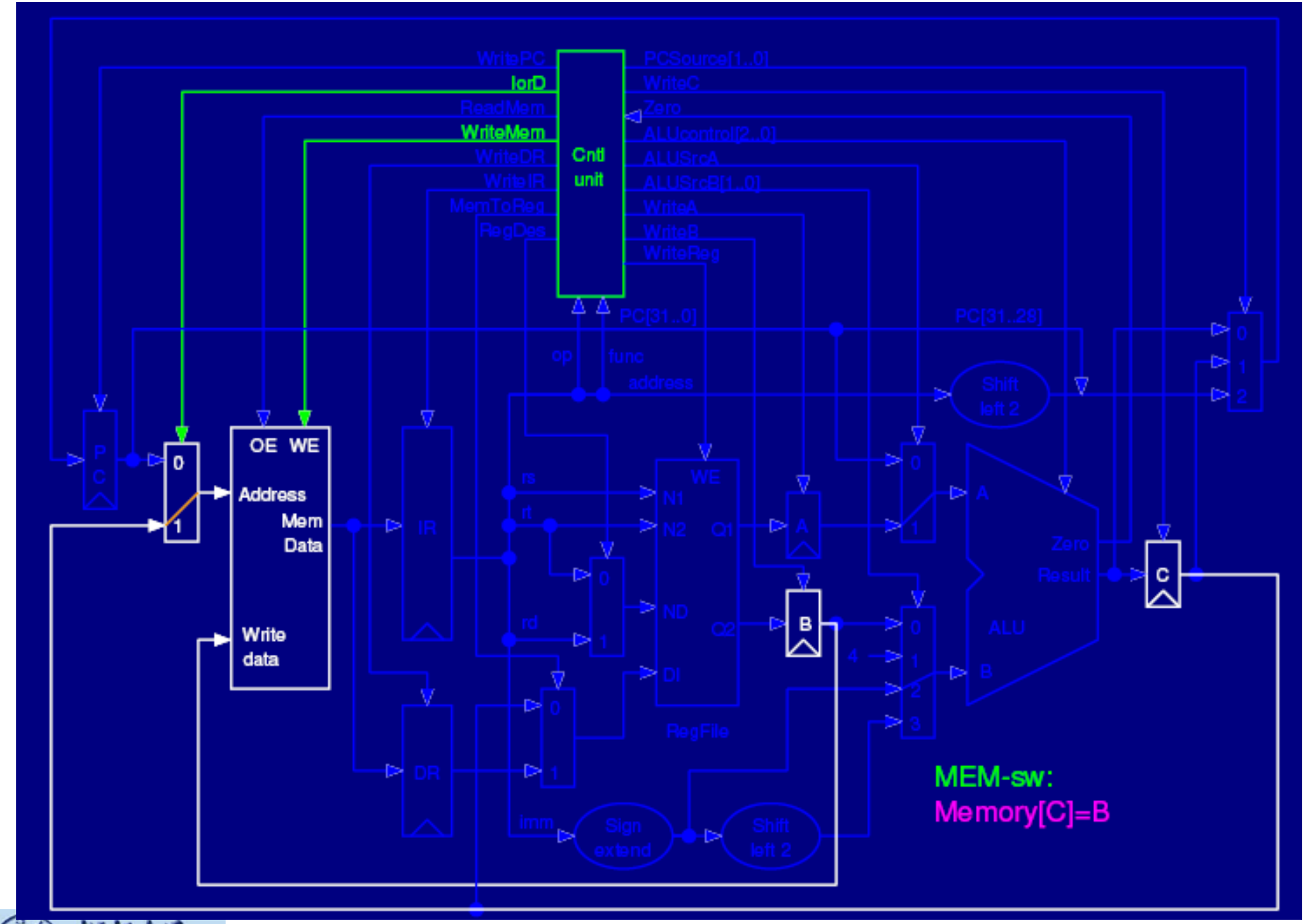

WritePC
IorD
ReadMem
WriteMem
WriteDR
WriteIR
MemToReg
RegDes
Cntl unit
PCSource[1..0]
WriteC
Zero
ALUcontrol[2..0]
ALUSrcA
ALUSrcB[1..0]
WriteA
WriteB
WriteReg
PC[31..0]
PC[31..28]
op
func
address
Shift left 2
P C
OE WE
Address
Mem Data
Write data
IR
rs
rt
rd
WE
N1
N2
ND
DI
Q1
Q2
RegFile
A
B
DR
imm
Sign extend
Shift left 2
4
ALU
Zero
Result
C
IF-Instruction Fetch:
IR=Memory[PC]
PC=PC+4

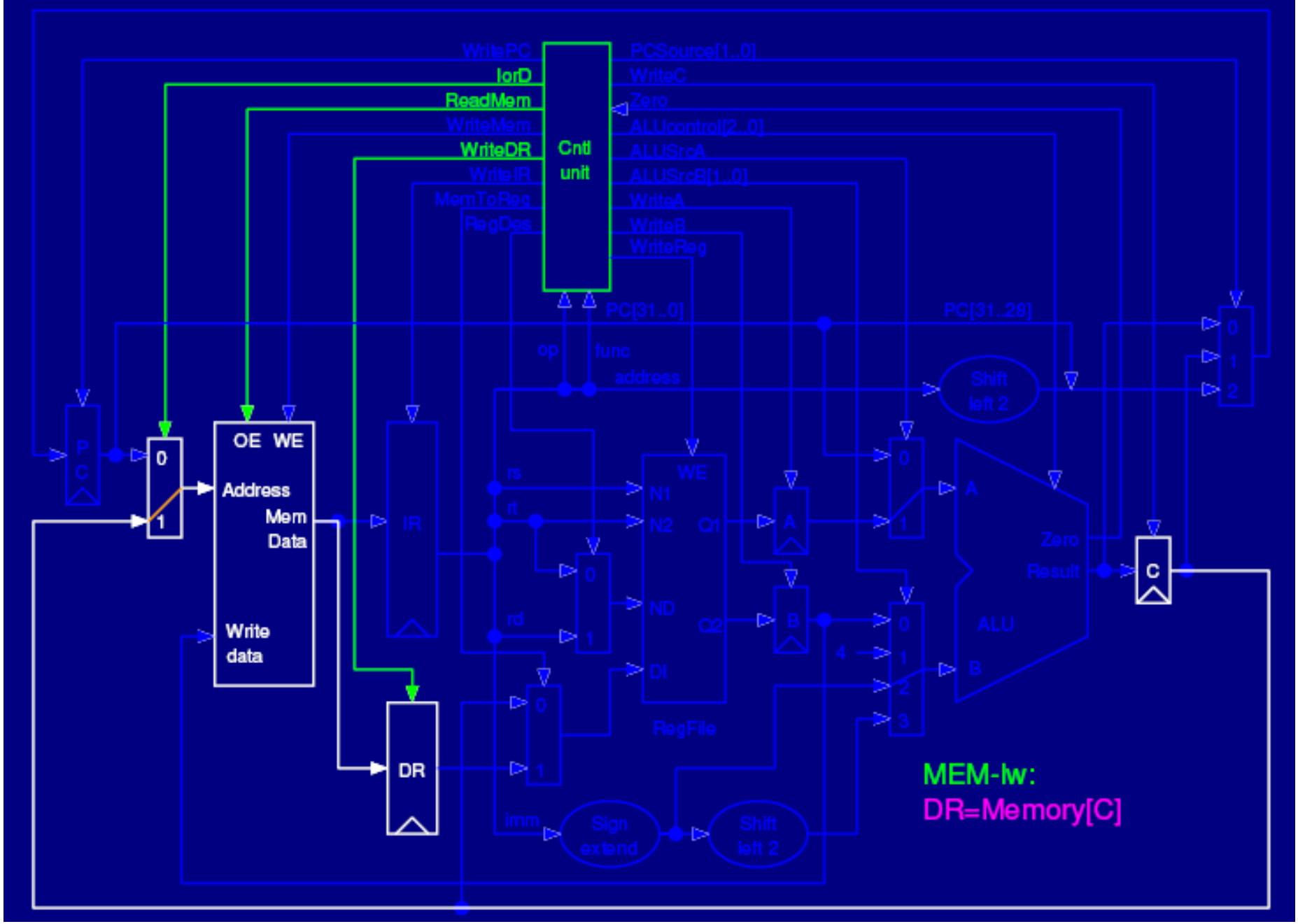

WritePC
IorD
ReadMem
WriteMem
WriteDR
WriteIR
MemToReg
RegDes
Cntl unit
PCSource[1..0]
WriteC
Zero
ALUcontrol[2..0]
ALUSrcA
ALUSrcB[1..0]
WriteA
WriteB
WriteReg
PC[31..0]
PC[31..28]
op
func
address
Shift left 2
P C
OE WE
Address
Mem Data
Write data
IR
DR
rs
rt
rd
WE
N1
N2
Q1
ND
Q2
DI
RegFile
A
B
4
ALU
Zero
Result
C
imm
Sign extend
Shift left 2
ID-Instruction Decode:
A=Reg[rs]
B=Reg[rt]
C=PC+(sign-ext(imm)<<2)

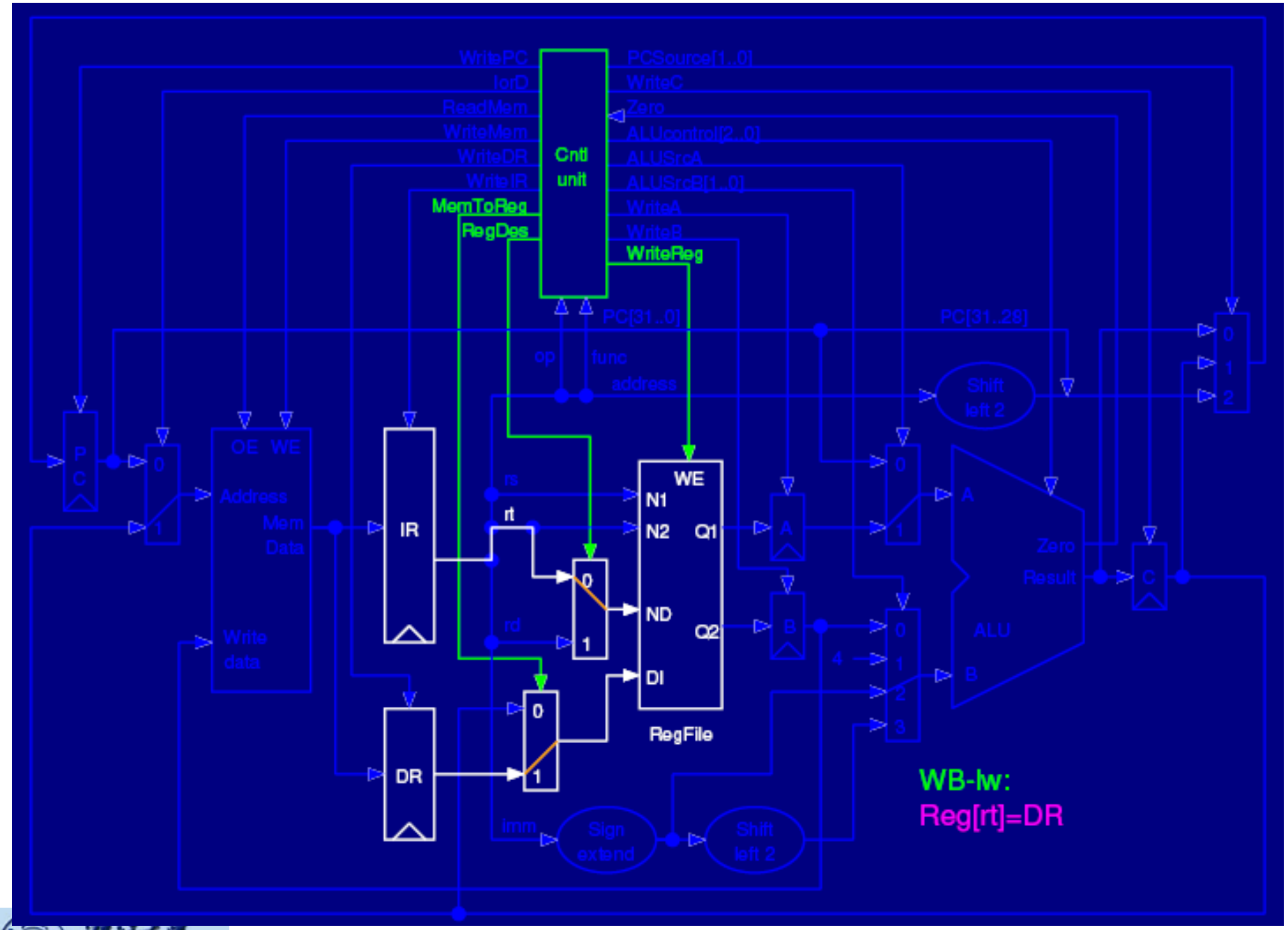

WritePC
IorD
ReadMem
WriteMem
WriteDR
WriteIR
MemToReg
RegDes
Cntl unit
PCSource[1..0]
WriteC
Zero
ALUcontrol[2..0]
ALUSrcA
ALUSrcB[1..0]
WriteA
WriteB
WriteReg
PC[31..0]
PC[31..28]
op
func
address
Shift left 2
P C
OE
WE
Address
Mem Data
Write data
IR
DR
rs
rt
rd
N1
N2
ND
DI
WE
Q1
Q2
RegFile
A
B
4
ALU
Zero
Result
C
imm
Sign extend
Shift left 2
EXE-Rtype:
C=A op B

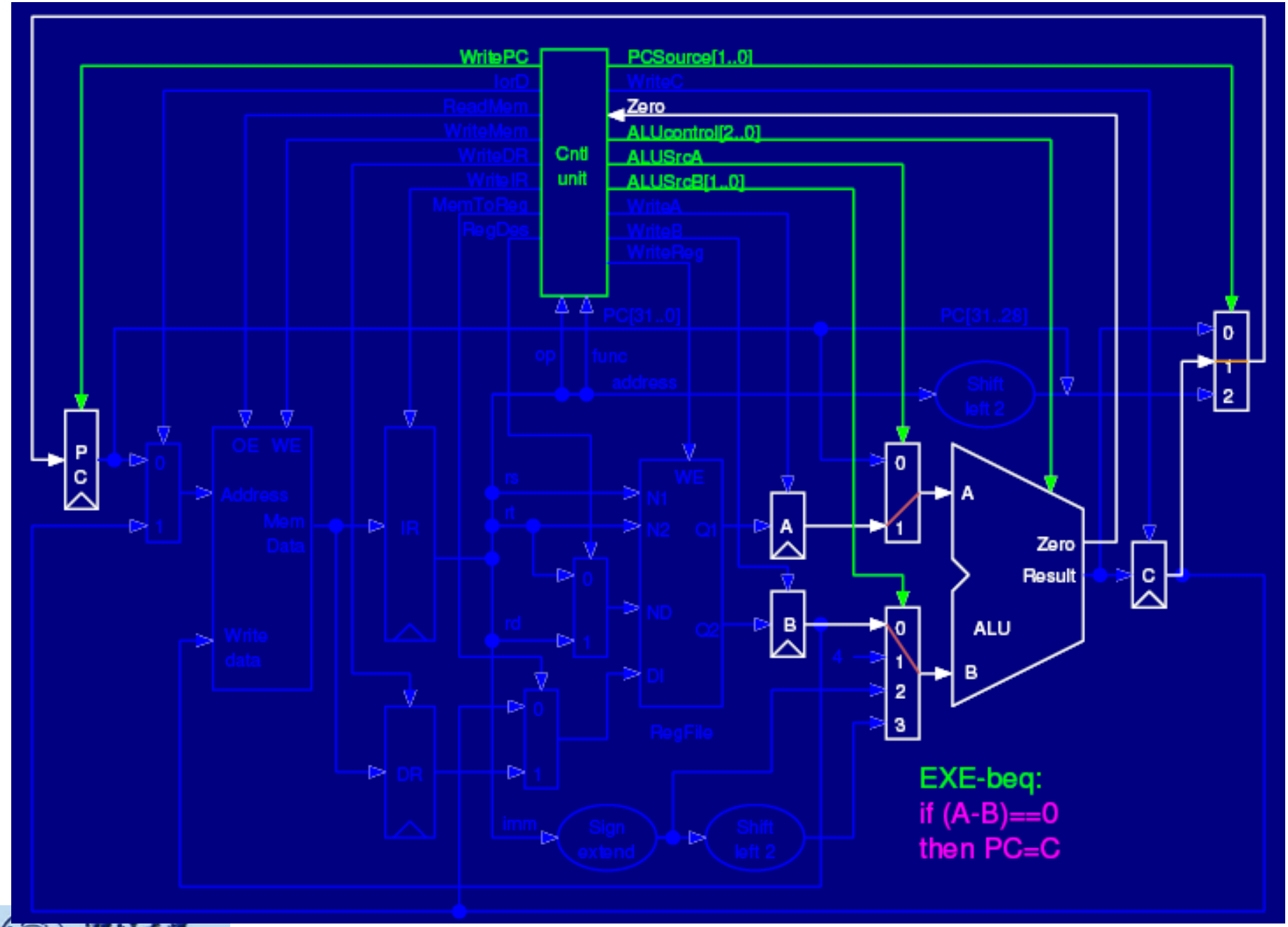

WritePC
IorD
ReadMem
WriteMem
WriteDR
WriteIR
MemToReg
RegDes
Cntl unit
PCSource[1..0]
WriteC
Zero
ALUcontrol[2..0]
ALUSrcA
ALUSrcB[1..0]
WriteA
WriteB
WriteReg
PC[31..0]
PC[31..28]
op
func
address
Shift left 2
P C
OE WE
Address
Mem Data
Write data
IR
DR
rs
rt
rd
WE
N1
N2
Q1
ND
Q2
DI
RegFile
A
B
4
ALU
Zero
Result
C
imm
Sign extend
Shift left 2
WB-Rtype:
Reg[rd]=C

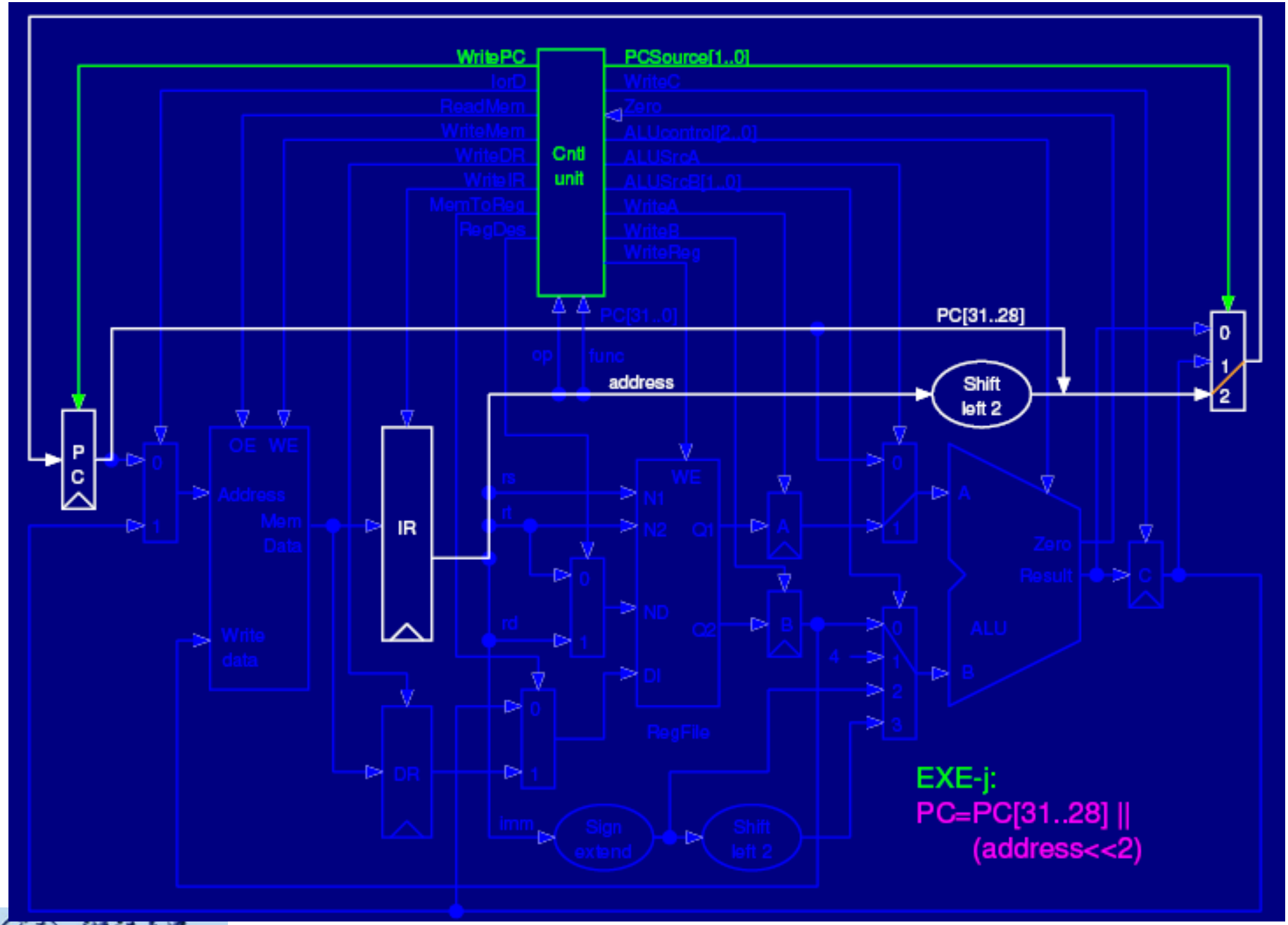

WritePC
IorD
ReadMem
WriteMem
WriteDR
WriteIR
MemToReg
RegDes
Cntl unit
PCSource[1..0]
WriteC
Zero
ALUcontrol[2..0]
ALUSrcA
ALUSrcB[1..0]
WriteA
WriteB
WriteReg
PC[31..0]
PC[31..28]
op
func
address
Shift left 2
P C
OE WE
Address
Mem Data
Write data
IR
rs
rt
rd
N1
N2
ND
DI
WE
Q1
Q2
RegFile
A
B
DR
0
1
2
3
4
ALU
Zero
Result
C
imm
Sign extend
Shift left 2
EXE-lw,sw:
C=A+(sign-ext(imm))

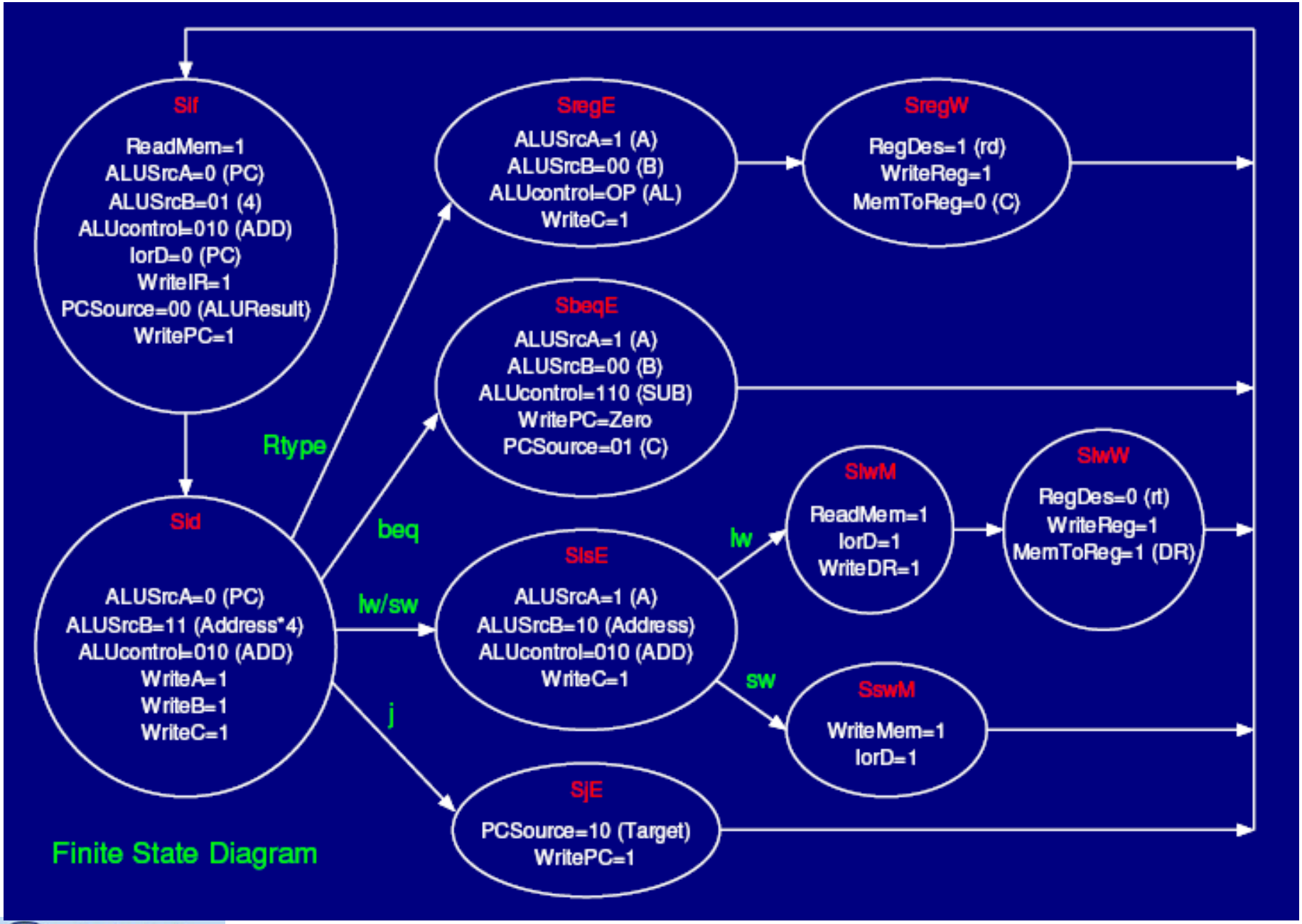

WritePC
IorD
ReadMem
WriteMem
WriteDR
WriteIR
MemToReg
RegDes
Cntl unit
PCSource[1..0]
WriteC
Zero
ALUcontrol[2..0]
ALUSrcA
ALUSrcB[1..0]
WriteA
WriteB
WriteReg
PC[31..0]
PC[31..28]
op
func
address
Shift left 2
PC
OE WE
Address
Mem Data
Write data
IR
DR
rs
rt
rd
imm
WE
N1
N2
ND
DI
O1
O2
RegFile
A
B
4
ALU
Zero
Result
C
Sign extend
Shift left 2
MEM-sw:
Memory[C]=B

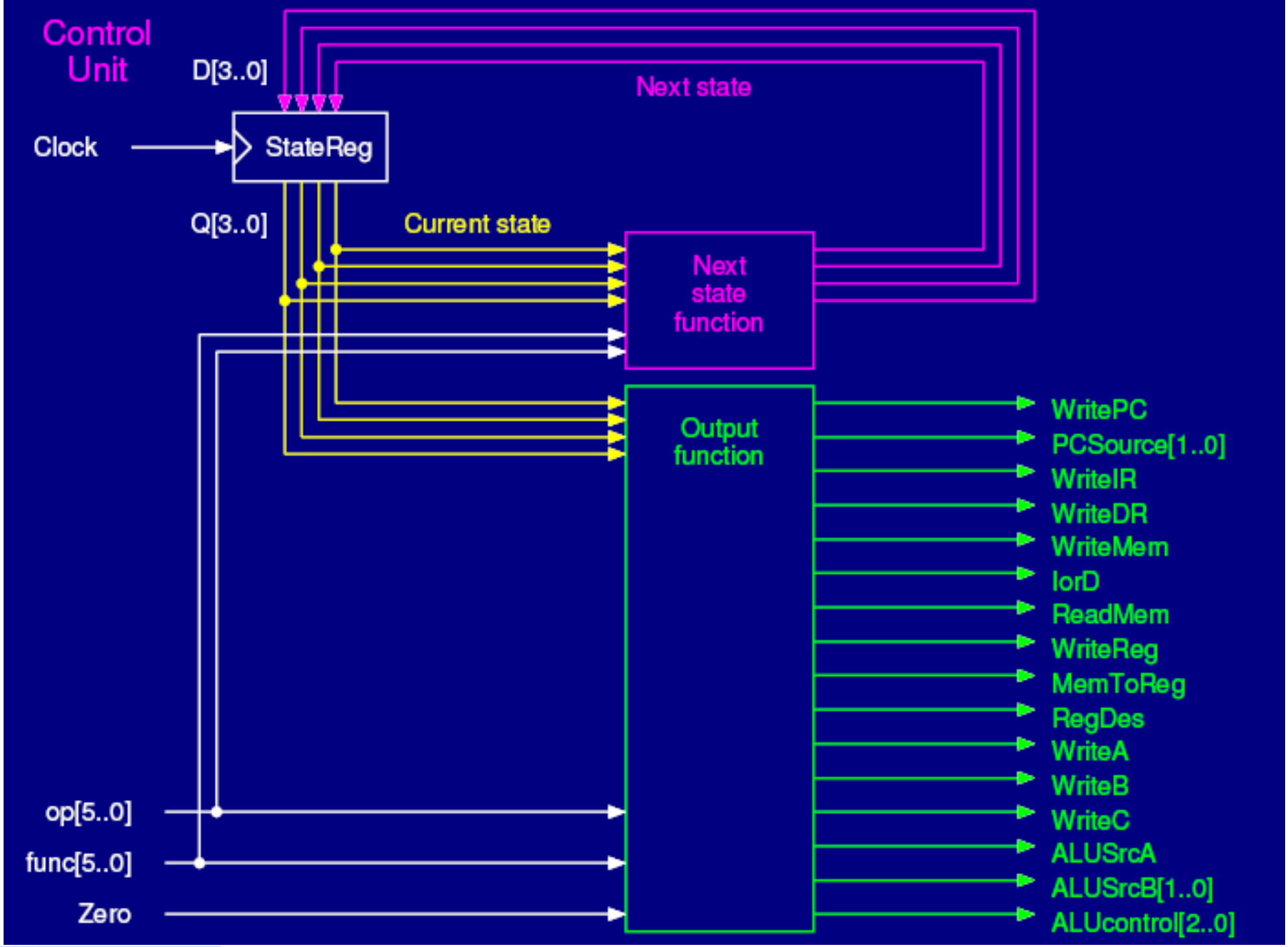

WritePC
IorD
ReadMem
WriteMem
WriteDR
WriteIR
MemToReg
RegDes
Cntl unit
PCSource[1..0]
WriteC
Zero
ALUcontrol[2..0]
ALUSrcA
ALUSrcB[1..0]
WriteA
WriteB
WriteReg
PC[31..0]
PC[31..28]
op
func
address
Shift left 2
P C
OE WE
Address
Mem Data
Write data
IR
DR
rs
rt
rd
WE
N1
N2
Q1
ND
Q2
DI
RegFile
A
B
ALU
Zero
Result
C
imm
Sign extend
Shift left 2
MEM-lw:
DR=Memory[C]

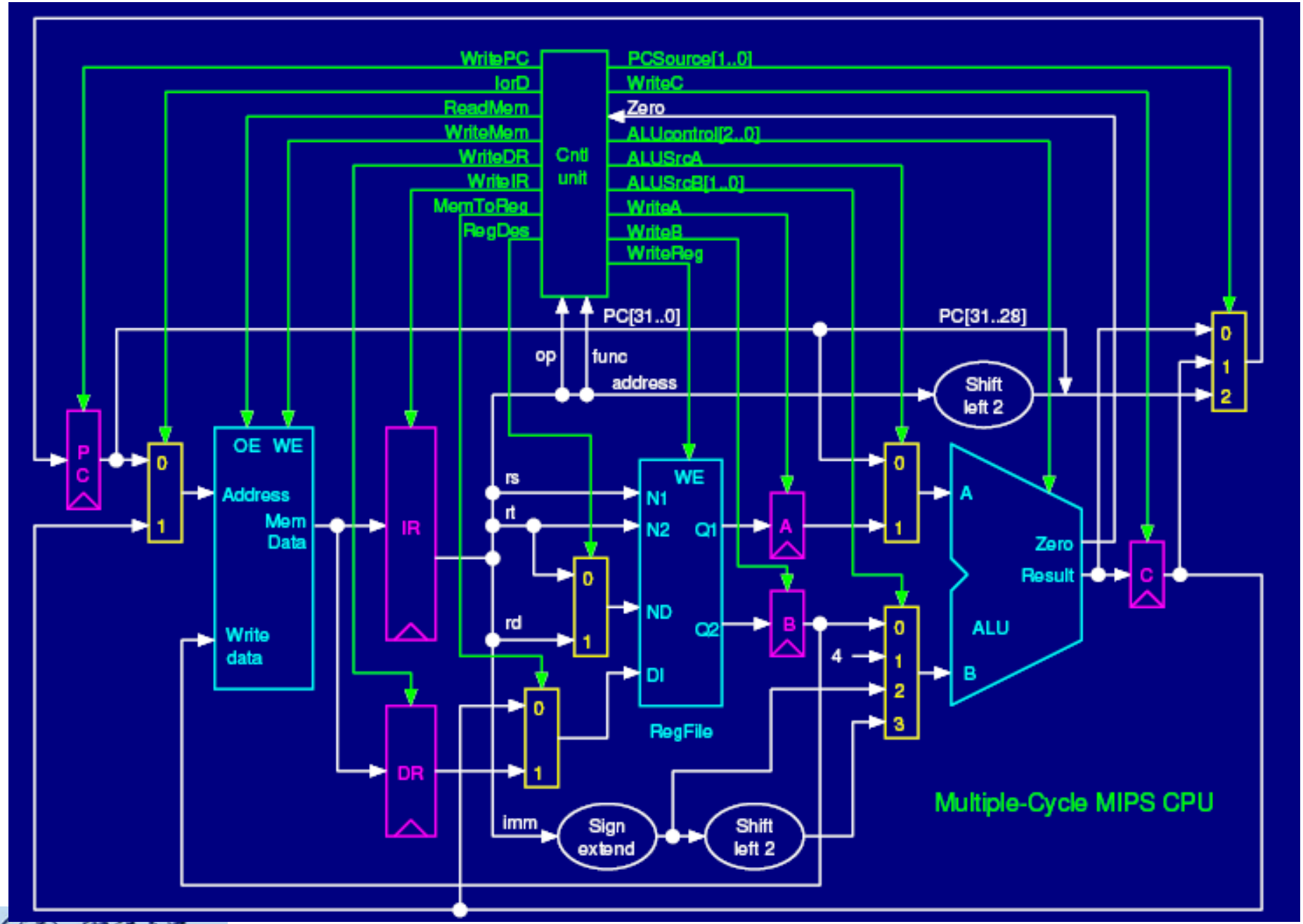

WritePC
IorD
ReadMem
WriteMem
WriteDR
WriteIR
MemToReg
RegDes
Cntl unit
PCSource[1..0]
WriteC
Zero
ALUcontrol[2..0]
ALUSrcA
ALUSrcB[1..0]
WriteA
WriteB
WriteReg
PC[31..0]
PC[31..28]
op
func
address
Shift left 2
P C
OE
WE
Address
Mem Data
Write data
IR
DR
rs
rt
rd
WE
N1
N2
Q1
ND
Q2
DI
RegFile
A
B
C
ALU
Zero
Result
4
imm
Sign extend
Shift left 2
WB-lw:
Reg[rt]=DR


WritePC
IorD
ReadMem
WriteMem
WriteDR
WriteIR
MemToReg
RegDes
Cntl unit
PCSource[1..0]
WriteC
Zero
ALUcontrol[2..0]
ALUSrcA
ALUSrcB[1..0]
WriteA
WriteB
WriteReg
PC[31..0]
PC[31..28]
op
func
address
Shift left 2
PC
OE WE
Address
Mem Data
Write data
IR
DR
rs
rt
rd
WE
N1
N2
ND
DI
O1
O2
RegFile
A
B
4
ALU
Zero
Result
C
imm
Sign extend
Shift left 2
EXE-beq:
if (A-B)==0
then PC=C

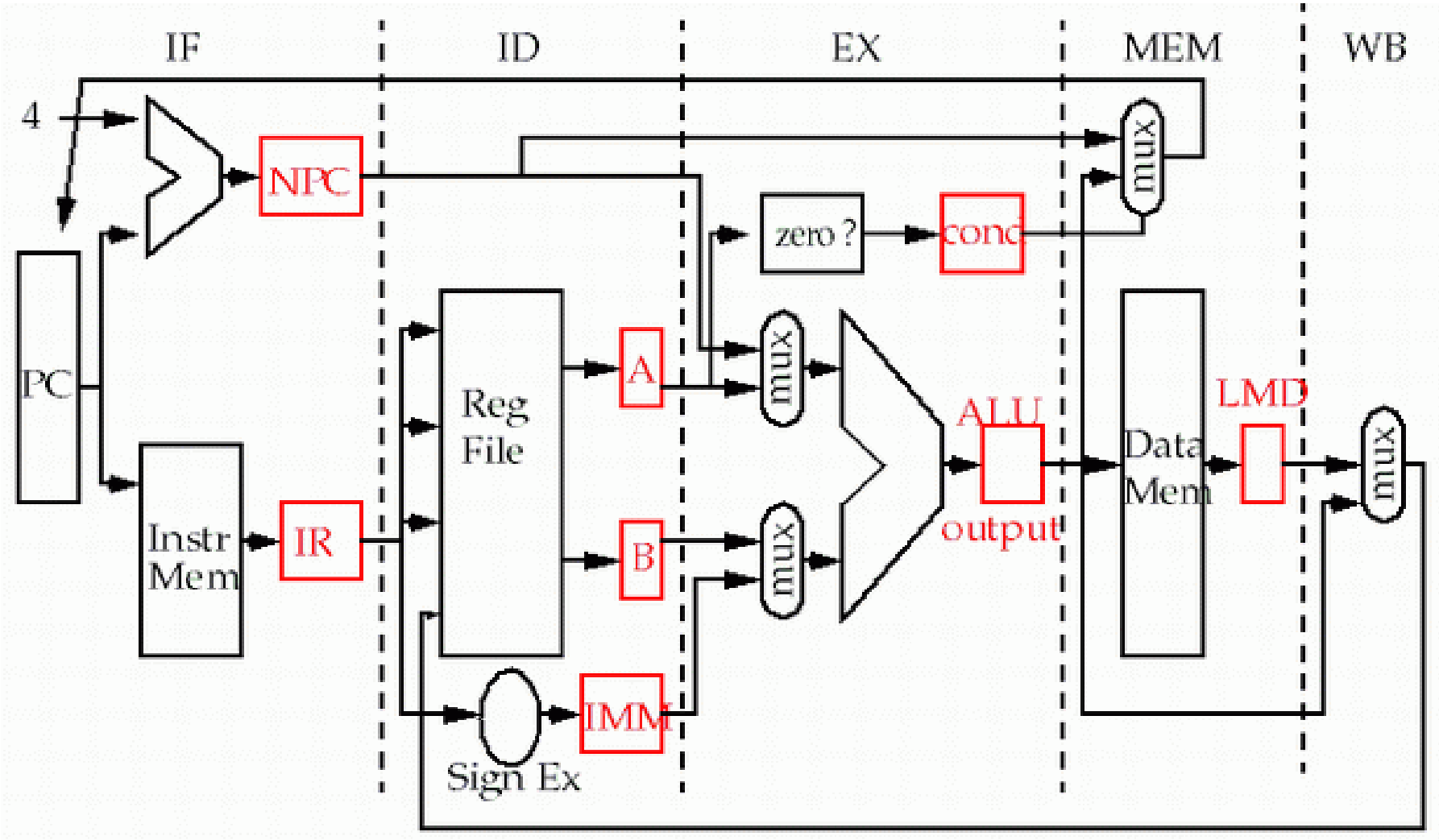

WritePC
IorD
ReadMem
WriteMem
WriteDR
WriteIR
MemToReg
RegDes
Cntl unit
PCSource[1..0]
WriteC
Zero
ALUcontrol[2..0]
ALUSrcA
ALUSrcB[1..0]
WriteA
WriteB
WriteReg
PC[31..0]
PC[31..28]
op
func
address
Shift left 2
PC
OE
WE
Address
Mem Data
Write data
IR
rs
rt
rd
N1
N2
ND
DI
WE
O1
O2
RegFile
A
B
ALU
Zero
Result
C
DR
imm
Sign extend
Shift left 2
EXE-j:
PC=PC[31..28] ||
(address<<2)


Sif
ReadMem=1
ALUSrcA=0 (PC)
ALUSrcB=01 (4)
ALUcontrol=010 (ADD)
IorD=0 (PC)
WriteIR=1
PCSource=00 (ALUResult)
WritePC=1
Sid
ALUSrcA=0 (PC)
ALUSrcB=11 (Address*4)
ALUcontrol=010 (ADD)
WriteA=1
WriteB=1
WriteC=1
Rtype
beq
lw/sw
j
SregE
ALUSrcA=1 (A)
ALUSrcB=00 (B)
ALUcontrol=OP (AL)
WriteC=1
SbeqE
ALUSrcA=1 (A)
ALUSrcB=00 (B)
ALUcontrol=110 (SUB)
WritePC=Zero
PCSource=01 (C)
SlsE
ALUSrcA=1 (A)
ALUSrcB=10 (Address)
ALUcontrol=010 (ADD)
WriteC=1
lw
sw
SjE
PCSource=10 (Target)
WritePC=1
SregW
RegDes=1 (rd)
WriteReg=1
MemToReg=0 (C)
SlwM
ReadMem=1
IorD=1
WriteDR=1
SlwW
RegDes=0 (rt)
WriteReg=1
MemToReg=1 (DR)
SswM
WriteMem=1
IorD=1
Finite State Diagram


Control Unit
D[3..0]
Next state
Clock
StateReg
Q[3..0]
Current state
Next state function
Output function
WritePC
PCSource[1..0]
WriteIR
WriteDR
WriteMem
IorD
ReadMem
WriteReg
MemToReg
RegDes
WriteA
WriteB
WriteC
ALUSrcA
ALUSrcB[1..0]
ALUcontrol[2..0]
op[5..0]
func[5..0]
Zero


WritePC
IorD
ReadMem
WriteMem
WriteDR
WriteIR
MemToReg
RegDes
Cntl unit
PCSource[1..0]
WriteC
Zero
ALUcontrol[2..0]
ALUSrcA
ALUSrcB[1..0]
WriteA
WriteB
WriteReg
PC[31..0]
PC[31..28]
op
func
address
Shift left 2
PC
OE WE
Address
Mem Data
Write data
IR
DR
rs
rt
rd
WE
N1
N2
Q1
ND
Q2
DI
RegFile
A
B
4
ALU
Zero
Result
C
imm
Sign extend
Shift left 2
Multiple-Cycle MIPS CPU

Multi-cycle implementation



About Multi-cycle implementation

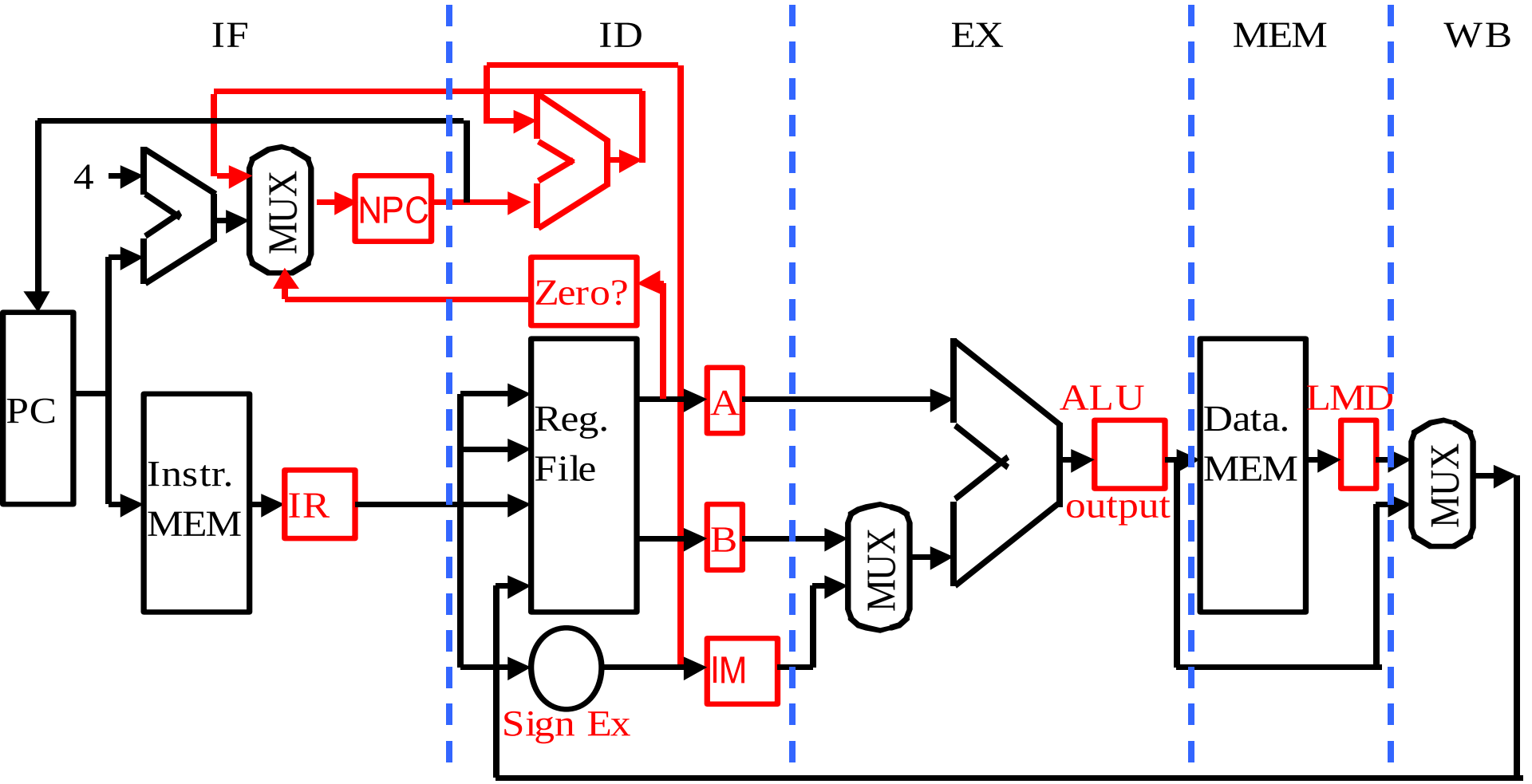
- The temporary storage locations were added to the datapath of the unpipelined machine to make it easy to pipeline.
- Note that branch and store instructions take 4 clock cycles.
  - Assuming branch frequency of 12% and a store frequency of 10%, CPI is 4.78.
- This implementation is not optimal.

How to improve the performance ?

- For a possible branch, do the equality test and compute the possible branch target by adding the sign-extended offset to the incremented PC earlier in ID.
- Completing ALU instructions during the MEM cycle
- So, branch instructions take only 2 cycles, store and ALU instructions take 4 cycles, and load instruction takes the longest time 5 cycles.
- CPI drops to 4.07 assuming 47% ALU operation frequency.

$$2\times12\% + 4\times(10\%+47\%) + 31\times5 = 4.07$$

Optimized Multi-cycle implementation



Temporary storage locations

Improvement on hardware redundancy

- ALU can be shared.
- Data and instruction memory can be combined since access occurs on different clock cycles.

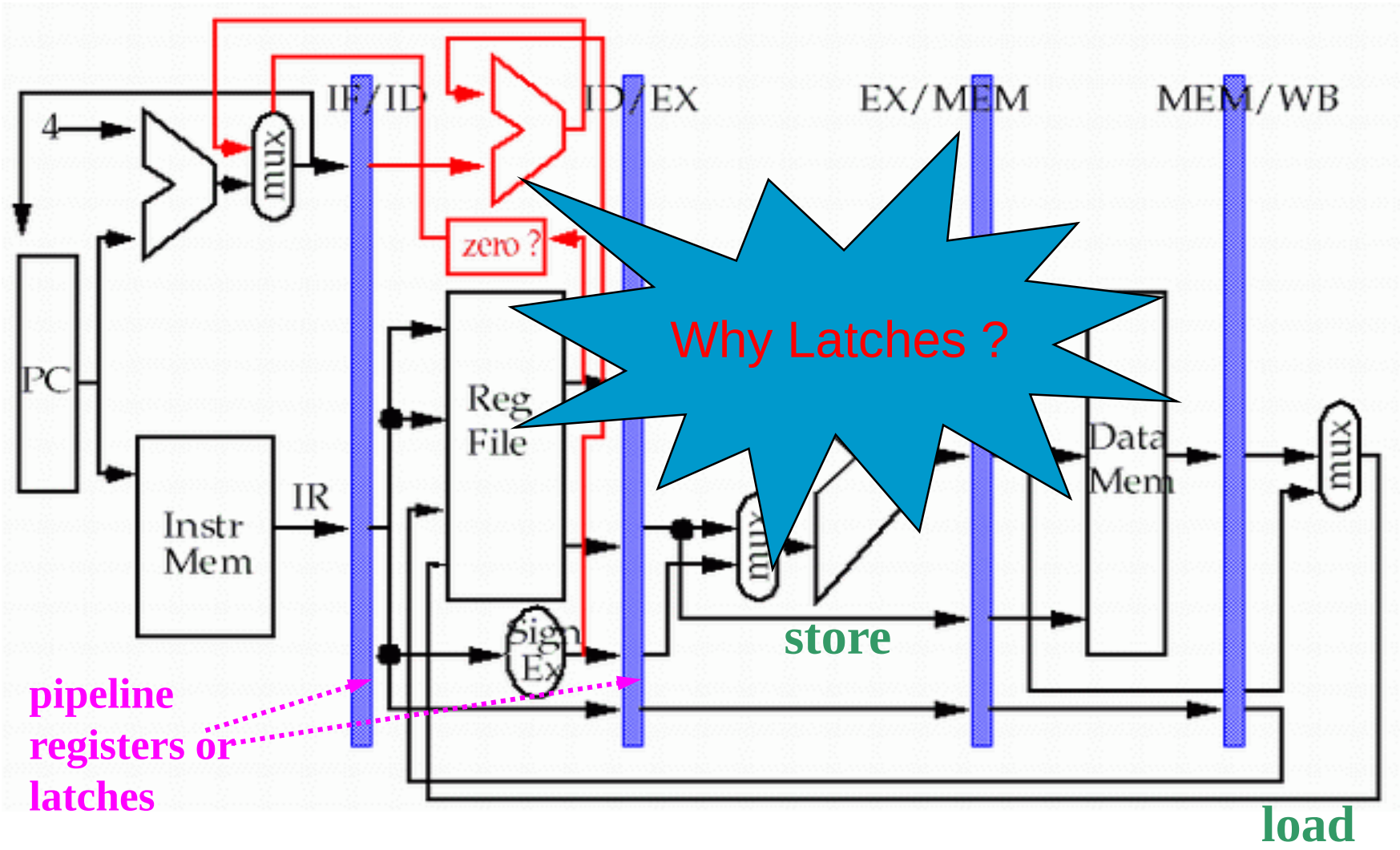
Pipelining MIPS instruction set

- Since there are five separate stages, we can have a pipeline in which one instruction is in each stage.
- CPI is decreased to 1, since one instruction will be issued (or finished) each cycle.
- During any cycle, one instruction is present in each stage.

	Clock Number								
	1	2	3	4	5	6	7	8	9
Instruction i	IF	ID	EX	MEM	WB				
Instruction i+1		IF	ID	EX	MEM	WB			
Instruction i+2			IF	ID	EX	MEM	WB		
Instruction i+3				IF	ID	EX	MEM	WB	
Instruction i+4					IF	ID	EX	MEM	WB

- Ideally, performance is increased five fold !

5-stage Version of MIPS Datapath

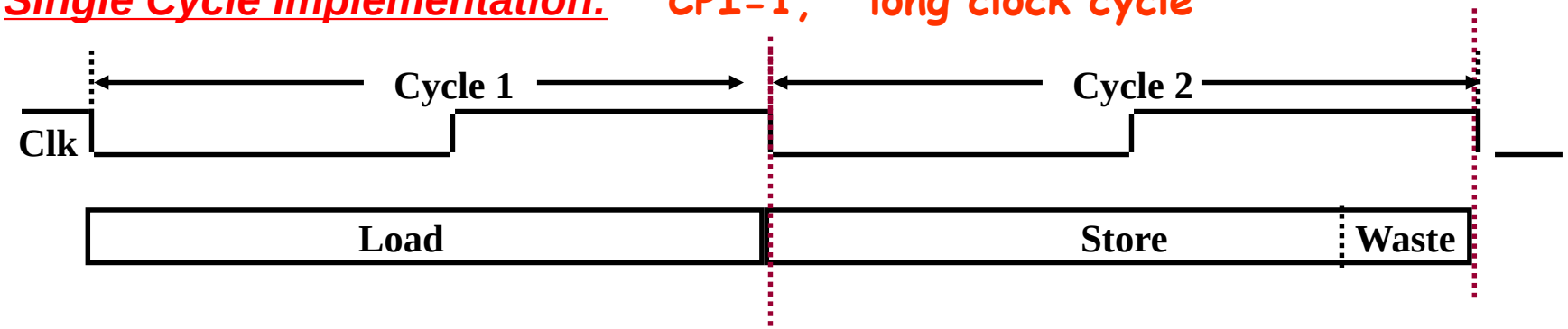


What should be latched ?

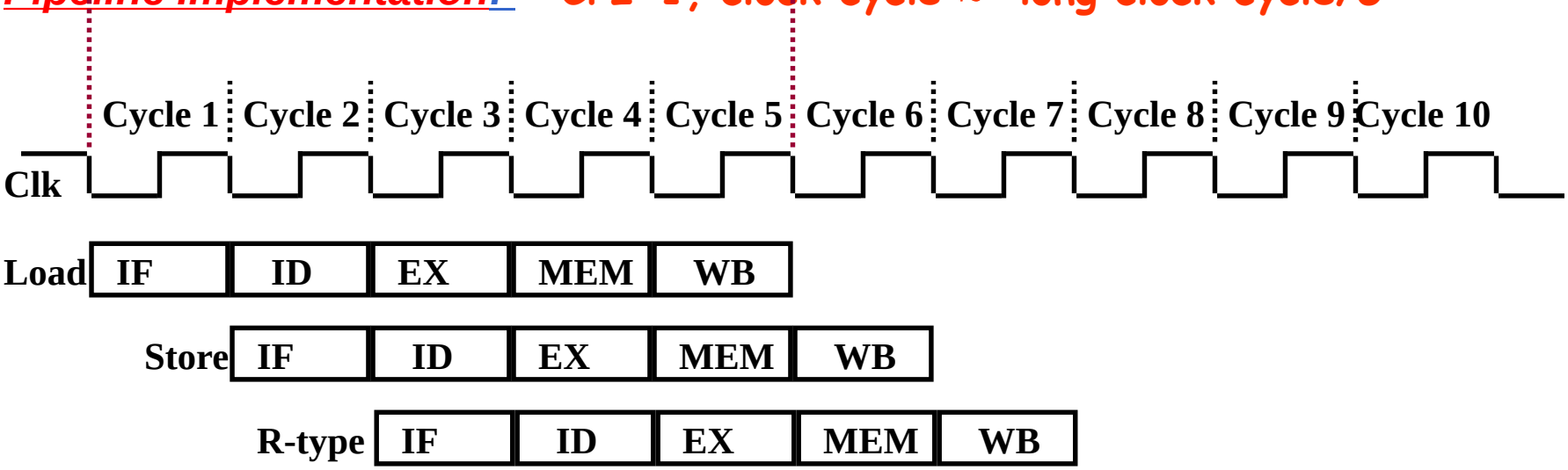
- Is register PC a latch ?
- What are stored in latch IF/ID ?
  - IR, anythingelse ?
- What are stored in latch ID/EX ?
  - A, B, imm, anythingelse ?
- What are stored in latch EX/MEM ?
  - ALUoutput, anythingelse ?
- What are stored in latch MEM/WB ?
  - LMDR, ALUoutput, Wreg, anythingelse ?

Single-cycle implementation vs. pipelining

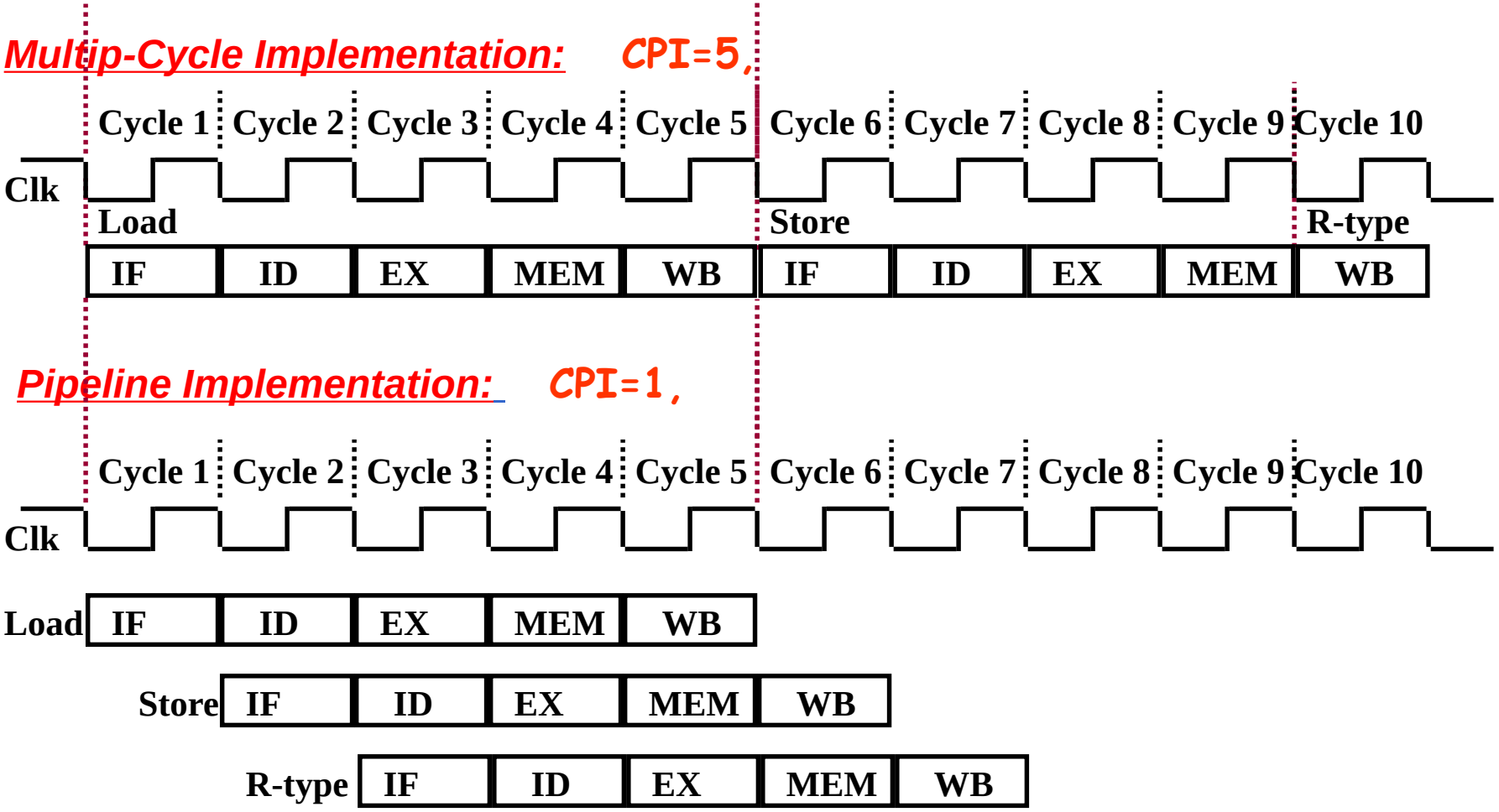
Single Cycle Implementation: CPI=1, long clock cycle



Pipeline Implementation: CPI=1, clock cycle ≈ long clock cycle/5



Multi-cycle implementation vs. pipelining



How pipelining decrease execution time?

If your starting point is

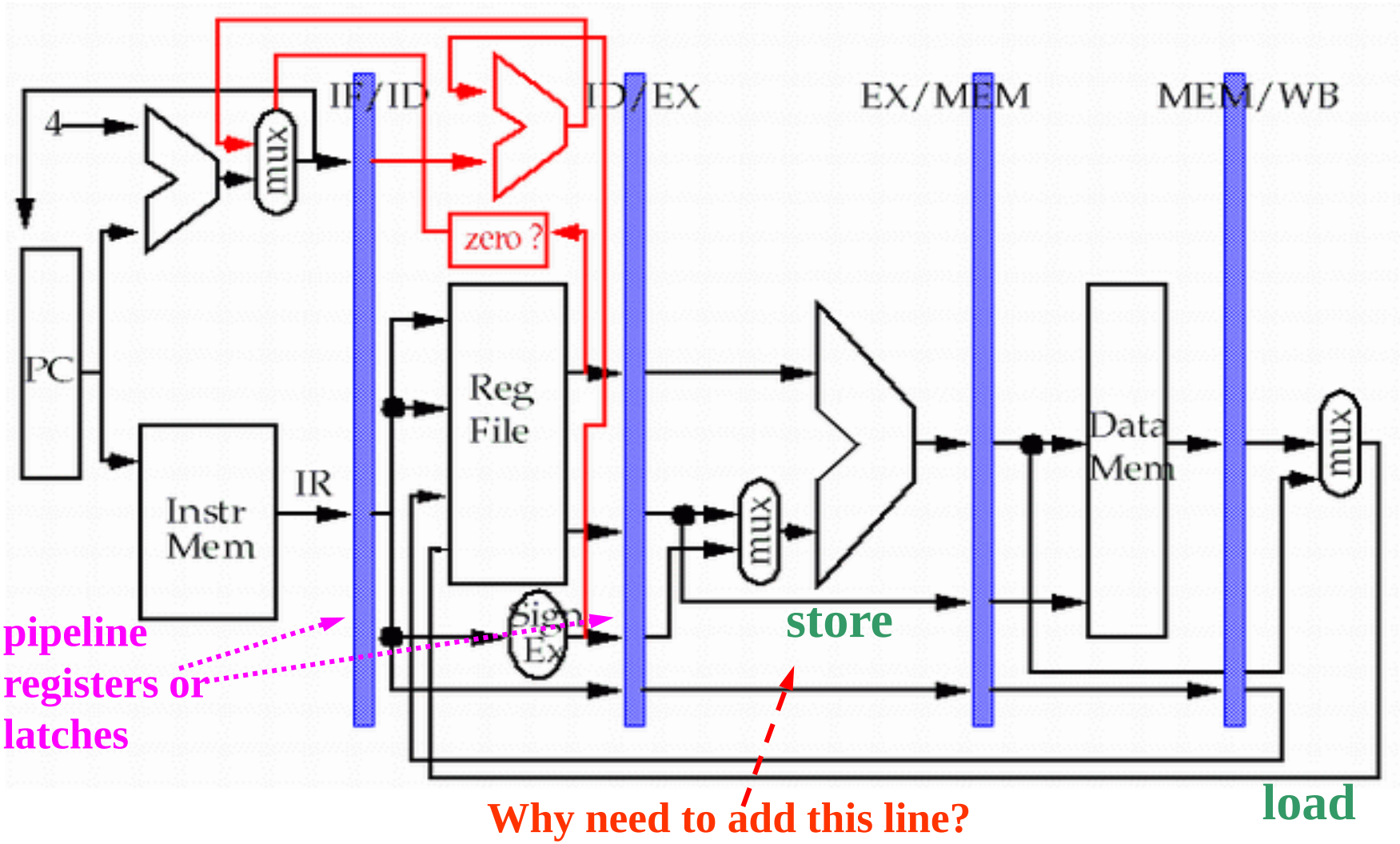
- a single clock cycle per instruction machine then
  - pipelining decreases cycle time.

- a multiple clock cycle per instruction machine then
  - pipelining decreases CPI.

performance issues in pipelining

- Latency: The execution time of each instruction in pipelining does not decrease, instead, always longer than that of unpipelined machine.
- Imbalance among stages reduces performance
- Overhead rise from register delay and clock skew also contribute to the lower limit of machine cycle.
- Pipeline hazards are the major hurdle of pipeline, which prevent the machine from reaching the ideal performance.
- Time to "fill" pipeline and time to "drain" it reduces speedup

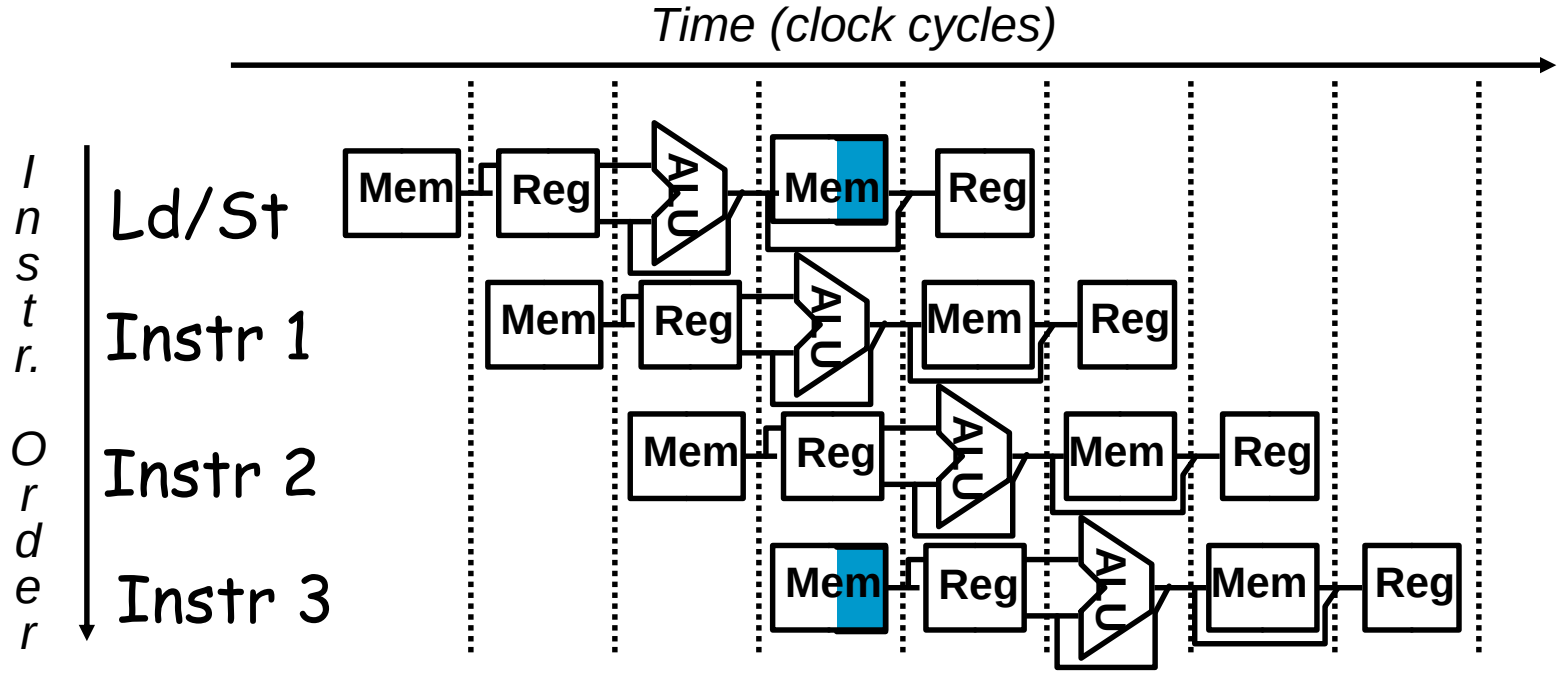
How simple as this ! Really ?



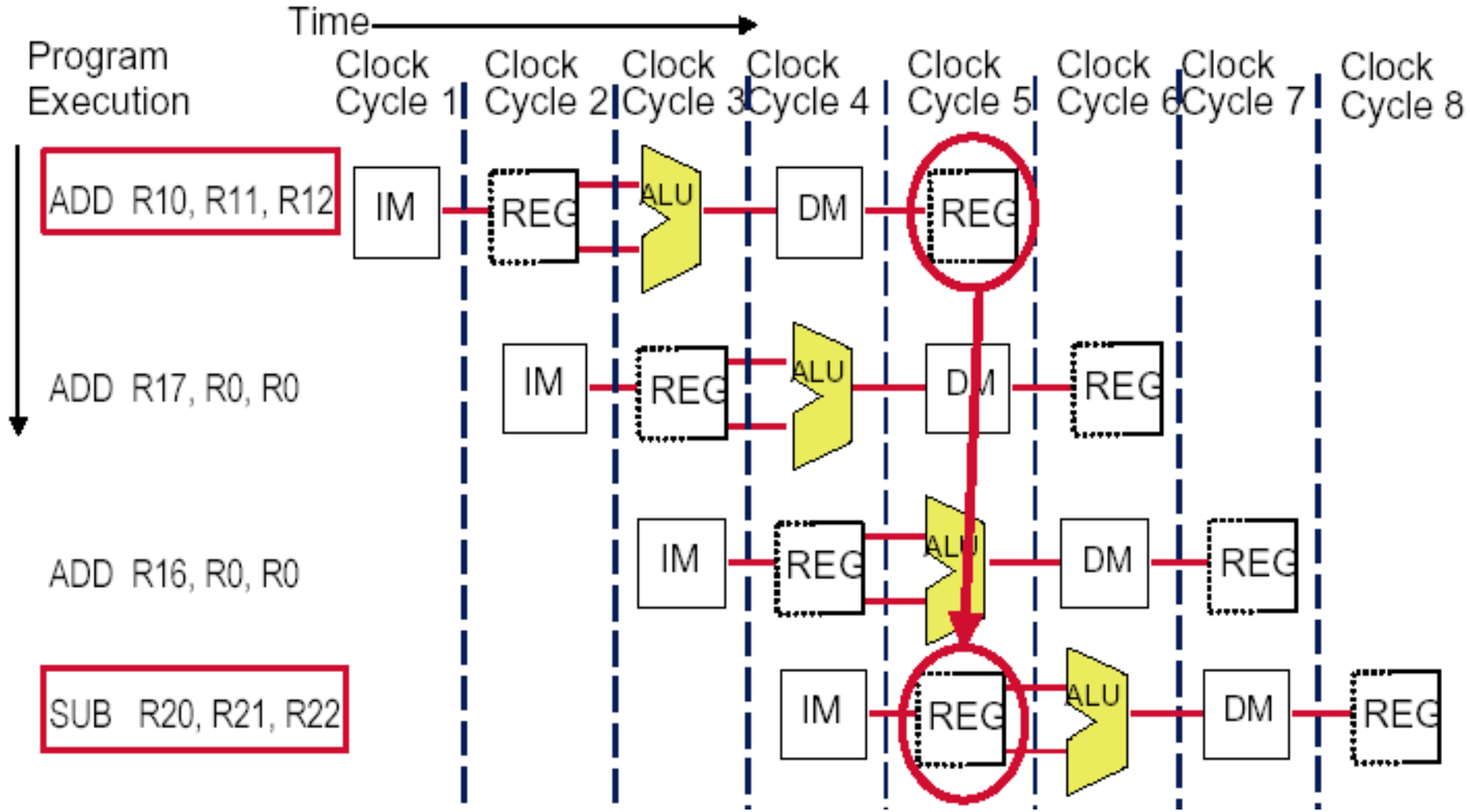
Problems that pipelining introduces

Focus: no different operations with the same data path resource on the same clock cycle. (structure hazard)

- There is conflict about the memory !



The conflict about the registers !



Conflict occurs when PC update

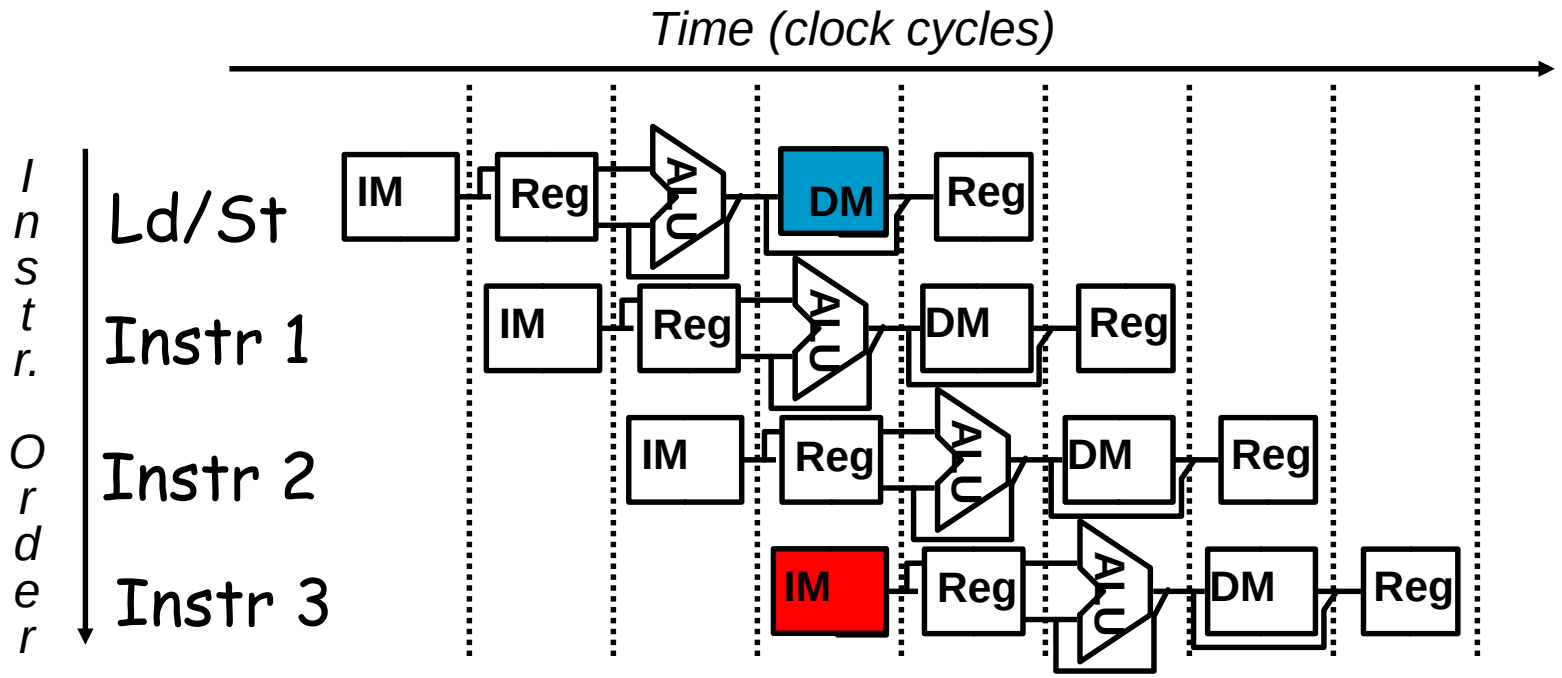
- Must increment and store the PC every clock.
- What happens when meet a branch ?
  - Branches change the value of the PC -- but the condition is not evaluated until ID !
  - If the branch is taken, the instructions fetched behind the branch are invalid !
- This is clearly a serious problem (Control hazard) that needs to be addressed. We will deal it later.

Must latches be engaged ? Yeah !

- Ensure the instructions in different stages do not interfere with one another .
- Through the latches, can the stages be combined one by one to form a pipeline.
- The latches are the pipeline registers , which are much more than those in multi-cycle version
  - IR: IF/ID.IR; ID/EX.IR; EX/DM.IR; DM/WB.IR
  - B: ID/EX.B; EX/DM.B
  - ALUoutput: EX/DM.ALUoutput, DM/WB.ALUoutput
- Any value needed on a later stage must be placed in a register and copied from one register to the next, until it is no longer needed.

Separate instruction and data memories

- use split instruction and data cache



- the memory system must deliver 5 times the bandwidth over the unpipelined version.

Pipeline hazard: the major hurdle

- *A hazard* is a condition that prevents an instruction in the pipe from executing its next scheduled pipe stage
- Taxonomy of hazard
  - **Structural hazards**
    - These are conflicts over hardware resources.
  - **Data hazards**
    - Instruction depends on result of prior computation which is not ready (computed or stored) yet
  - **Control hazards**
    - branch condition and the branch PC are not available in time to fetch an instruction on the next clock

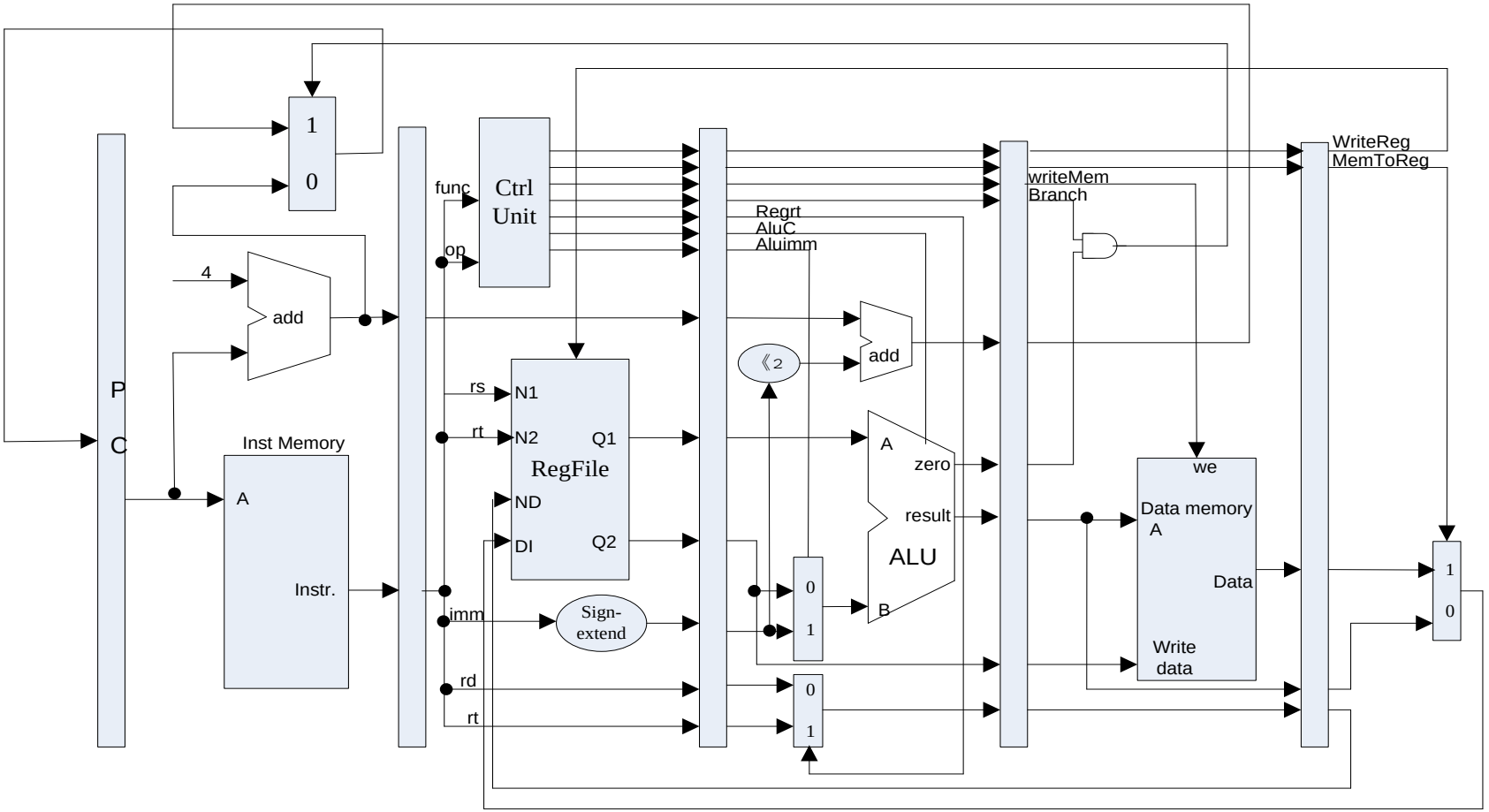
Hazards can always be resolved by Stall

- The simplest way to "fix" hazards is to **stall** the pipeline.
- Stall means suspending the pipeline for some instructions by one or more clock cycles.
- The stall delays all instructions issued after the instruction that was stalled, while other instructions in the pipeline go on proceeding.
- A pipeline stall is also called a pipeline bubble or simply bubble.
- No new instructions are fetched during a stall .

How to stall ?

- Stall = control hazard || structural hazard || data hazard
  - Control hazard:
    - Instruction in EX or MEM is a Branch or JMP
  - Data hazard:
    - RD of Instruction in EX == Rs or Rt of instruction in ID
    - RD of Instruction in MEM == Rs or Rt of instruction in ID

Data path and CU for pipelined CPU



IF	ID	EXE	MEM	WB

Control signals

	Output Signal	Meaning When 1	Meaning When 0
1	Cu_branch	Branch Instr.	Non-Branch Instr.
2	Cu_shift	sa	Register data1
3	Cu_wmem	Write Mem.	Not Write Mem.
4	Cu_Mem2Reg	From Mem. To Reg	From ALUOut To Reg
5	Cu_sext	Sign-extend the imm.	No sign extended the imm.
6	Cu_aluc	ALU Operation	
7	Cu_aluimm	Imm.	Register data2
8	Cu_wreg	Write Reg.	Not Write Reg.
9	Cu_regrt	rt	rd

Something about control signals

- How to know the instruction in EXE/MEM is a Branch?
  - EXE.branch ==1 or Mem.branch == 1
- How to know the instruction in EXE/MEM is a ALU or LW instruction (has a destination reg.) ?
  - EXE.Wreg ==1 or Mem.Wreg == 1
- How to know the instruction in EXE/MEM is a LW ?
  - EXE.Wreg==1 and Mem2Reg == 1
- Which instruction has no rs ?
  - Jmp, JAL
- Which instruction has rt as source register?
  - ALU r-r, Beq, Bne, SW

How to stall ?

- If stall == 0 //stop the latter ones
  - then PC ← new PC; IF/ID.IR ← IF.IR
  - IF/ID.NPC ← PC + 4
- If stall == 1 // push bubble forward
  - then ID/EX.nop ←1 else ID/EX.nop ←0
- When initial

  ID/EX.nop ←0, EX/MEM.nop ← 0, Mem.WB.nop ←0

Performance of pipeline with stalls

- Pipeline stalls decrease performance from the ideal
- Recall the speedup formula:

$$\text{Speedup from pipelining} = \frac{\text{Average instruction time unpipelined}}{\text{Average instuction time pipelined}}$$

$$= \frac{\text{CPI unpipelined} \times \text{Clock cycle unpipelined}}{\text{CPI pipelined} \times \text{Clock cycle pipelined}}$$

$$= \frac{\text{CPI unpipelined}}{\text{CPI pipelined}} \times \frac{\text{Clock cycle unpipelined}}{\text{Clock cycle pipelined}}$$

Assumptions for calculation

- The ideal CPI on a pipelined processor is almost always 1. (may less than or greater that)

So

$$\text{CPI pipelined} = \text{Ideal CPI} + \text{Pipeline stall clk cycles per instuction}$$

$$= 1 + \text{Pipeline stall clk cycles per instruction}$$

- Ignore the overhead of pipelining clock cycle.
- Pipe stages are ideal balanced.
- Two different implementation
  - single-cycle implementation
  - multi-cycle implementation

Case of single-cycle implementation

- CPI unpipelined = 1

- Clock cycle pipelined = $\frac{\text{Clock cycle unpipelined}}{\text{pipeline depth}}$

$$\text{Speedup} = \frac{1}{1 + \text{Pipeline Stall cycles per instruction}} \times \frac{\text{Clock cycle unpipelined}}{\text{Clock cycle pipelined}}$$

$$\text{Speedup} = \frac{\text{Pipeline depth}}{1 + \text{Pipeline stall cycles per instruction}}$$

Case of multi-cycle implementation

- Clock cycle unpipelined = Clock cycle pipelining
- CPI unpipelined = pipeline depth

$$\text{Speedup} = \frac{\text{CPI unpipelined}}{1 + \text{Pipeline stall cycles per instruction}}$$

$$\text{Speedup} = \frac{\text{Pipeline depth}}{1 + \text{Pipeline stall cycles per instruction}}$$

- That’s all for today !