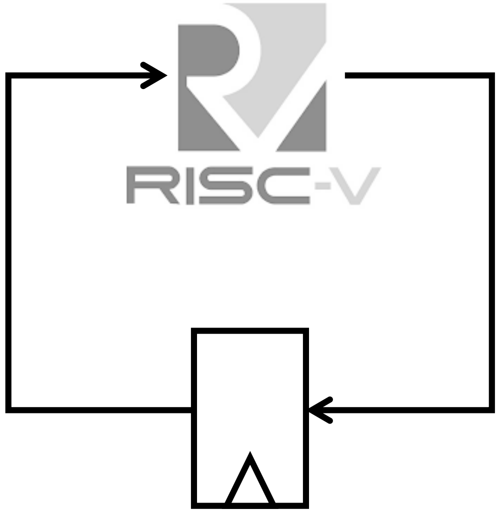
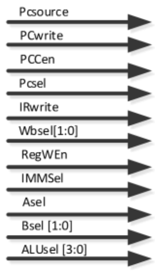
EE 044252: Digital Systems and Computer Structure

Spring 2019

Lecture 11: RISC-V Multi-Cycle





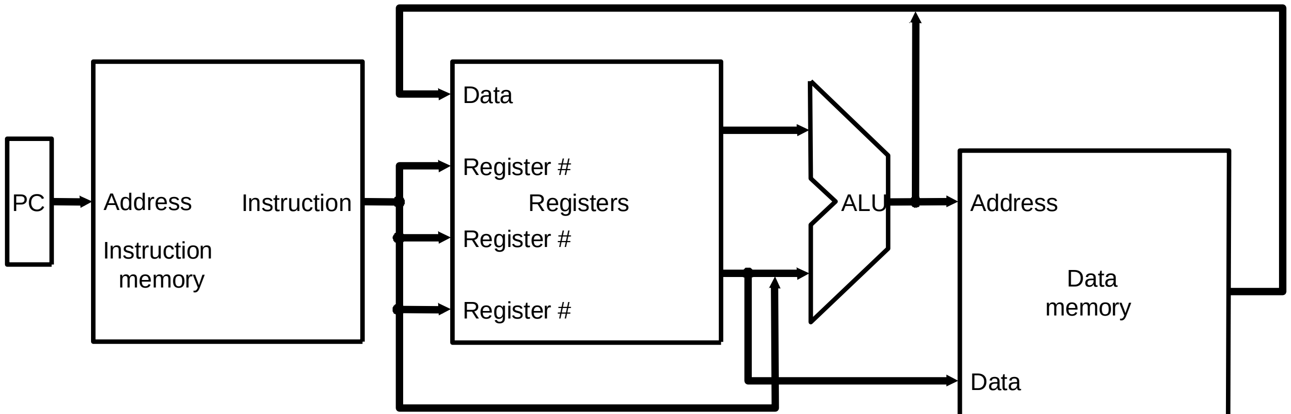
EE 044252: Digital Systems and Computer Structure

Topic	wk	Lectures	Tutorials	Workshop	Simulation
Arch	1	Intro. RISC-V architecture	Numbers. Codes		
Comb	2	Switching algebra & functions	Assembly programming		
	3	Combinational logic	Logic minimization	Combinational	
	4	Arithmetic. Memory	Gates		Combinational
Seq	5	Finite state machines	Logic		
	6	Sync FSM	Flip flops, FSM timing	Sequential	Sequential
	7	FSM equiv, scan, pipeline	FSM synthesis		
	8	Serial comm, RISC-V functions	Serial comm, pipeline		
µArch	9	RISC-V instruction formats	Function call		
	10	RISC-V single cycle	Single cycle RISC-V		Multi-cycle
	11	Multi-cycle RISC-V	Multi-cycle RISC-V		
	12	Interrupts, pipeline RISC-V	Microcode, interrupts		
	13	Dependencies in pipeline RISC-V	Depend. in pipeline RISC-V		

Agenda

- Multi-Cycle RISC-V Datapath
- Performance
- ROM Controller
- Micro-Coded Controller

Reminder: Single Cycle Datapath



More about single cycle datapath

- Simple to design and understand
- Performance limited by critical path—slowest instruction
  - LW
- Discouraged from adding even more complicated instructions

Elastic Cycle Time

- We saw that Load and Store instructions take more time than others
  - May be we can write a program that contains only "fast" commands?
- We know the dynamic instruction distribution (in many programs):
  - 24% - Loads, 12% - Stores, 44% - R-Type, 18% - Branches, 2% - Jumps
- We know the actual time <u>required</u> for each instruction (from Lecture 10):
  - Load—800ps, Stores—700ps, R-Type—600ps, Branches/Jumps—500ps
- Thus, the <u>average</u> time <u>required</u> for instruction execution is:
  - 800×24% + 700×12% + 600×44% + 500×18% + 500×2% = 640 ps
- We could create an elastic clock. It will allocate just the required time for each instruction
  - Its average cycle time is 640ps
  - Its average frequency is 1/640ps = 1.56GHz
- Improvement relative to single cycle: 800ps / 640ps = 1.56G / 1.25G = 1.25

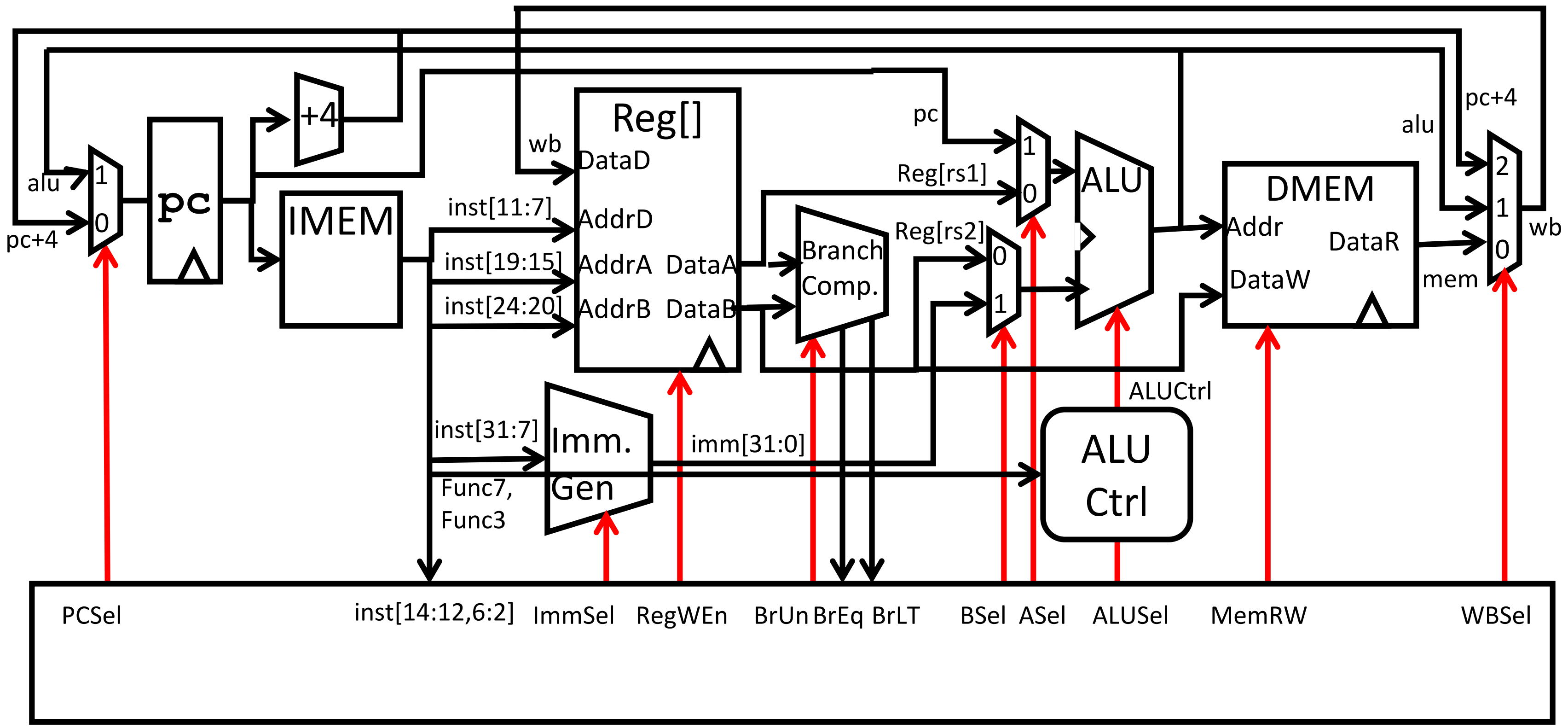
Multi-Cycle RISC-V

- Let’s break computation down into multiple stages
- Perform one stage per one clock cycle
- Different instructions require a different number of cycles
- Goal: Faster computer. How is this possible?
- Same functional units can be re-used in multiple cycles of same instruction
  - We could not do that in a single-cycle RISC-V
- Need new registers for intermediate storage (in-between cycles of same instruction)
- Single-cycle RISC-V uses a combinational controller
  Multi-cycle RISC-V uses a FSM controller

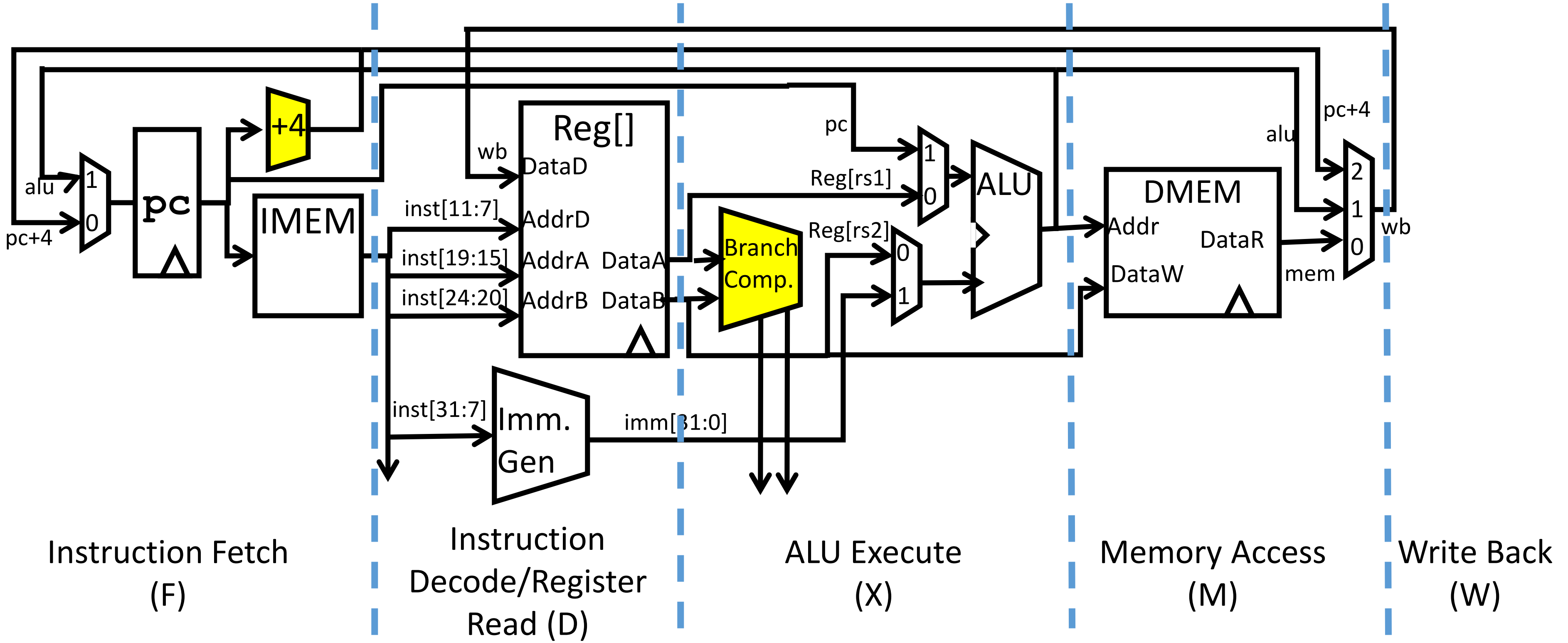
Two types of registers

- Registers holding persistent data
  - Contents used/written in one instruction can be used in a following instruction
  - Register File (32 registers), PC
  - Defined in architecture, visible in instruction simulators (e.g., venus)
- NEW in multi-cycle: Registers holding temporary data
  - Contents held only during execution of a single instruction
  - Unknown to programmer and to assembly language

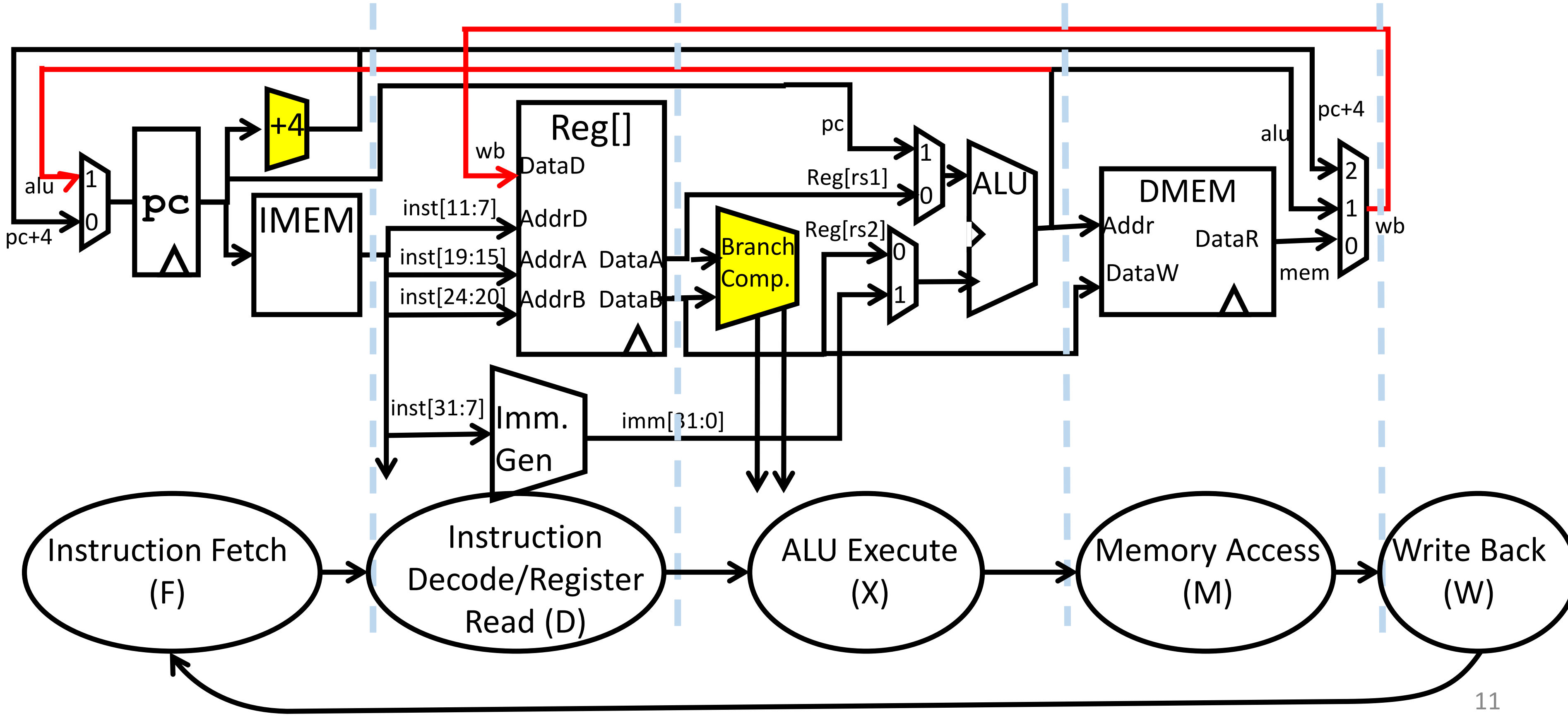
Single-Cycle RISC-V RV32I Datapath



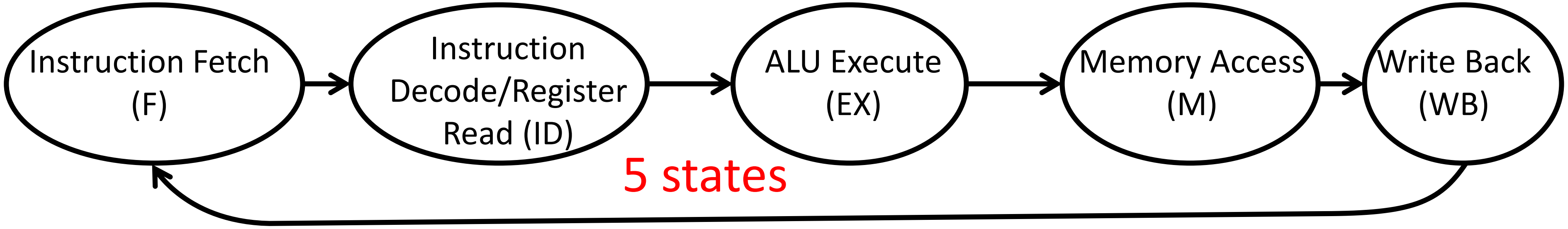
Breaking the datapath into 5 stages



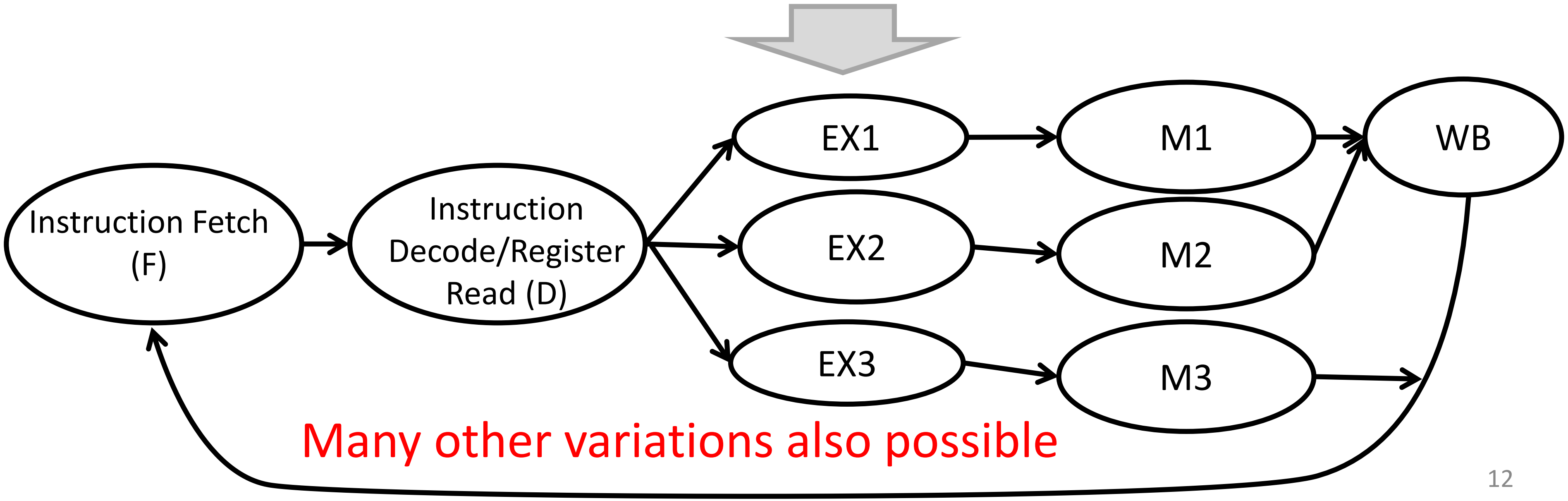
5 stages → 5 cycles (and loop back)



The controller is FSM

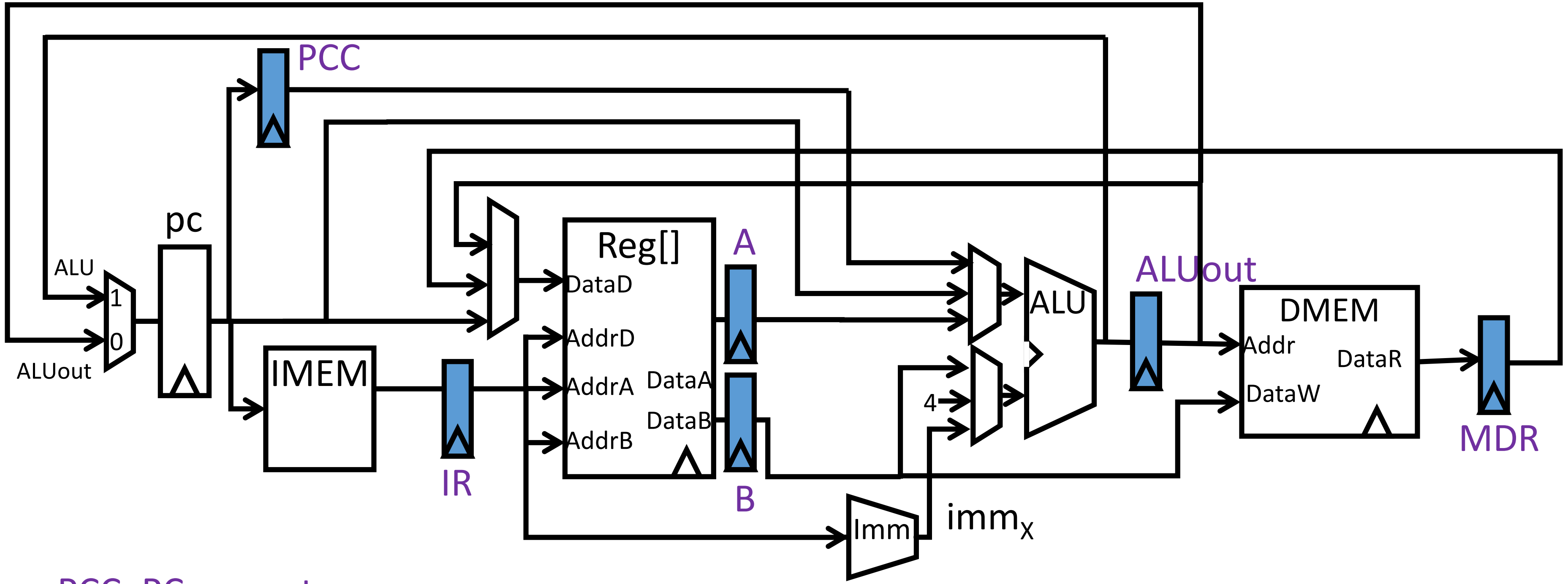


5 states



Many other variations also possible

Adding temporary registers (and making some changes)



PCC=PC current

IR=Instruction Register

A

B

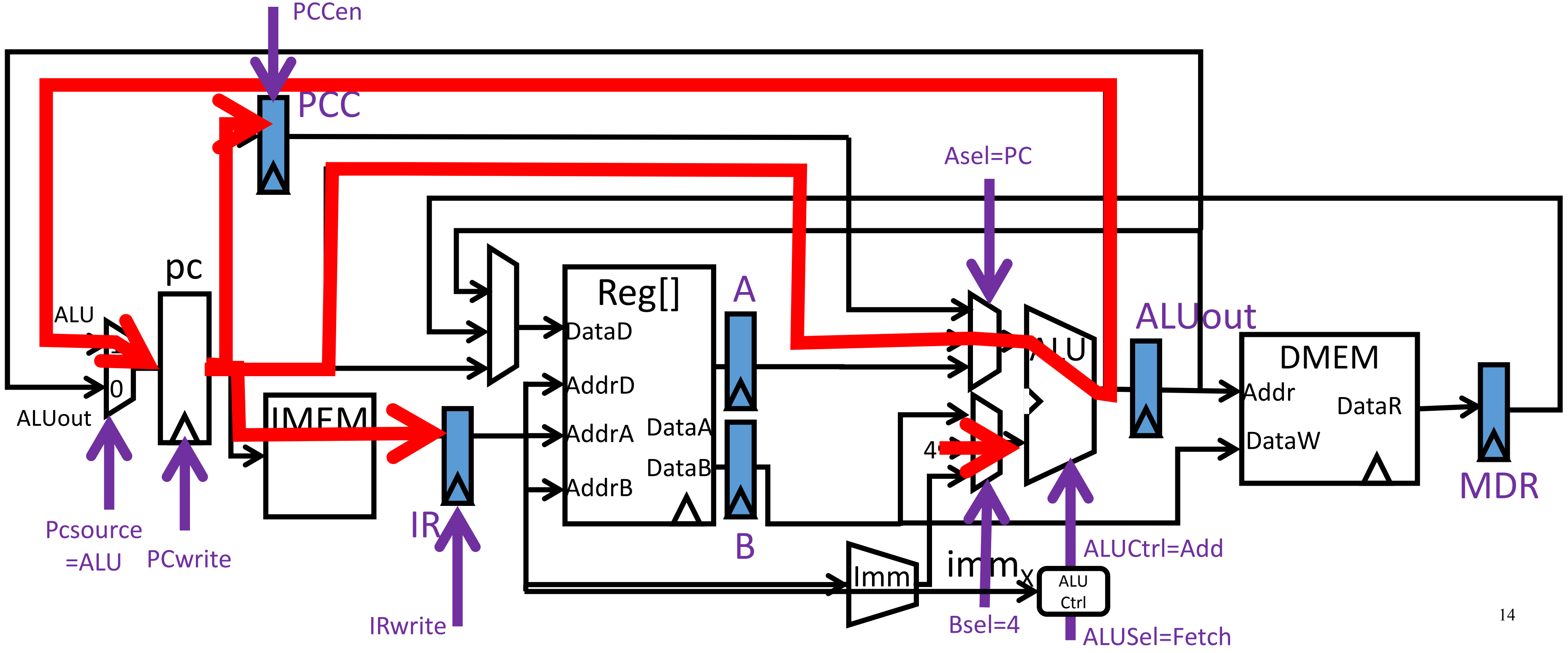
MDR=Memory Data Register

ALUout

FETCH cycle is the same for all instructions

FETCH: fetching the instruction pointed to by the PC

1. IR = Memory[PC]
2. PC = PC+4, PCC = PC

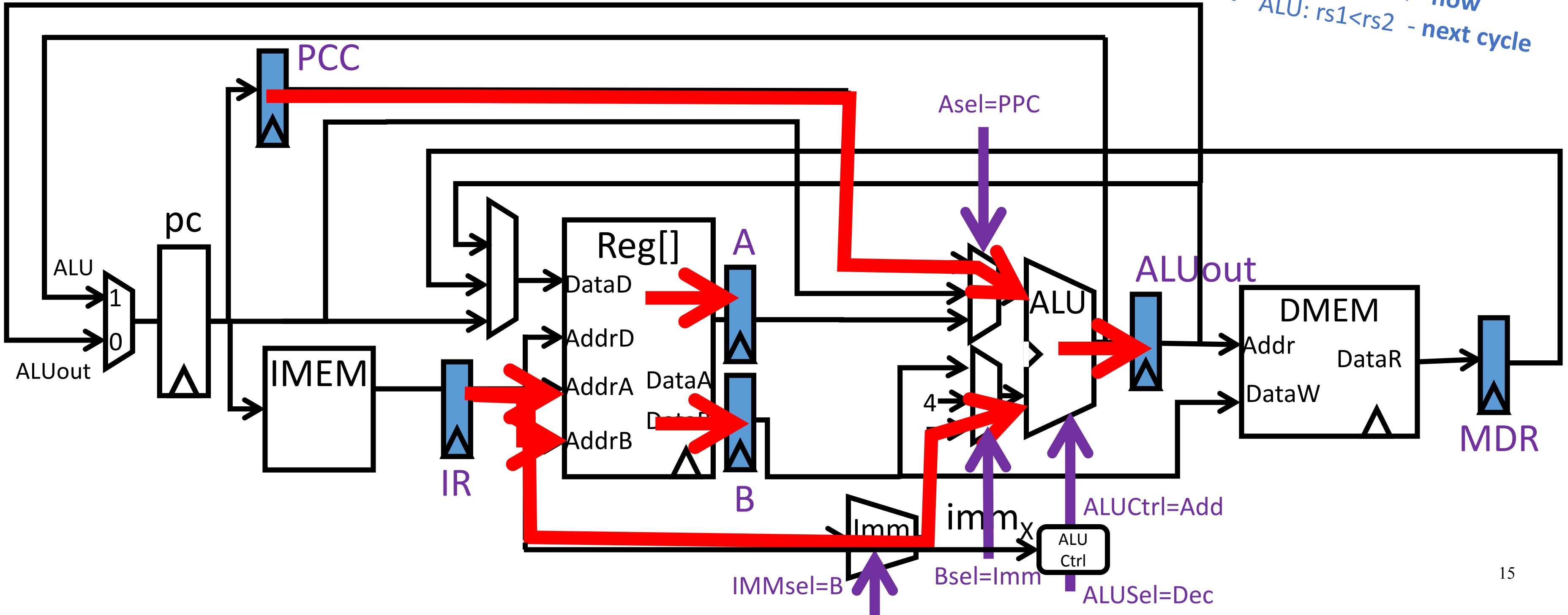


DECODE cycle is the same for all instructions

DECODE: decode instruction, read two registers, (speculatively) compute branch target address

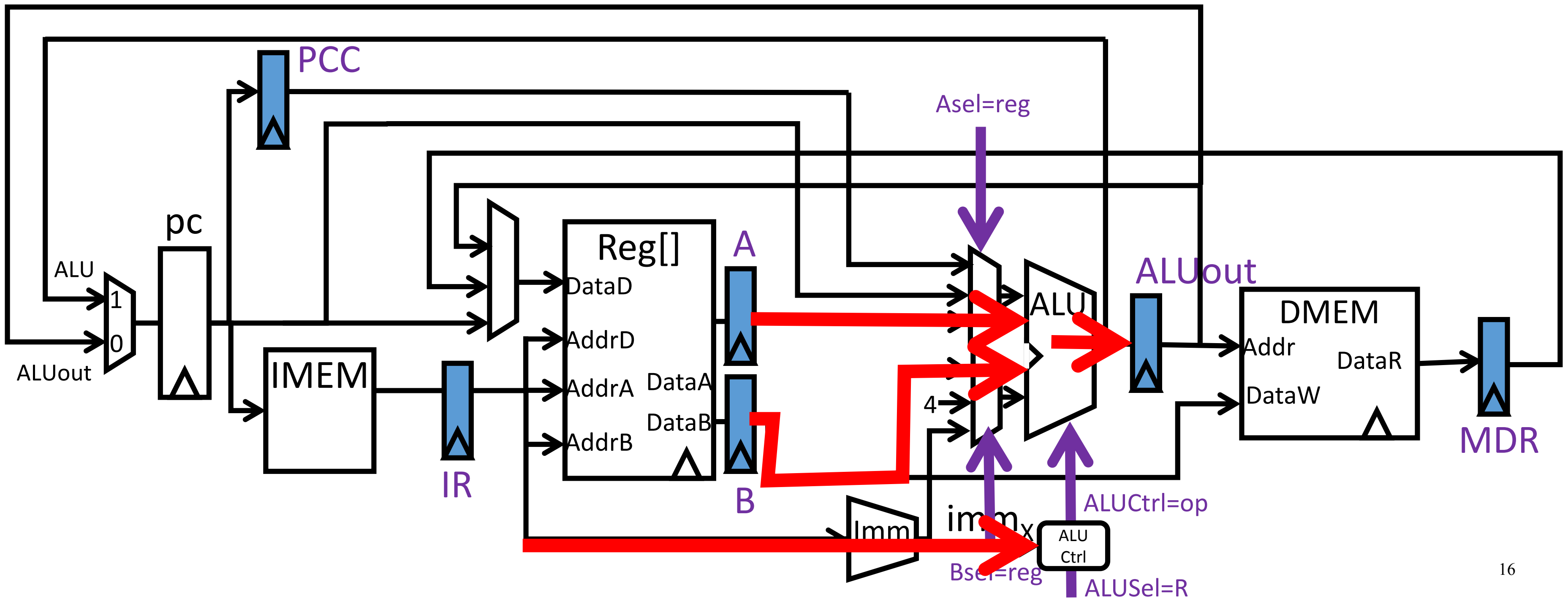
1. A = Reg[rs1], B = Reg[rs2]
2. ALUOut = PCC + branch offset

BLT rs1,rs2,imm
- ALU: PC+imm - **now**
- ALU: rs1<rs2 - **next cycle**



Execute Cycle for R-Type

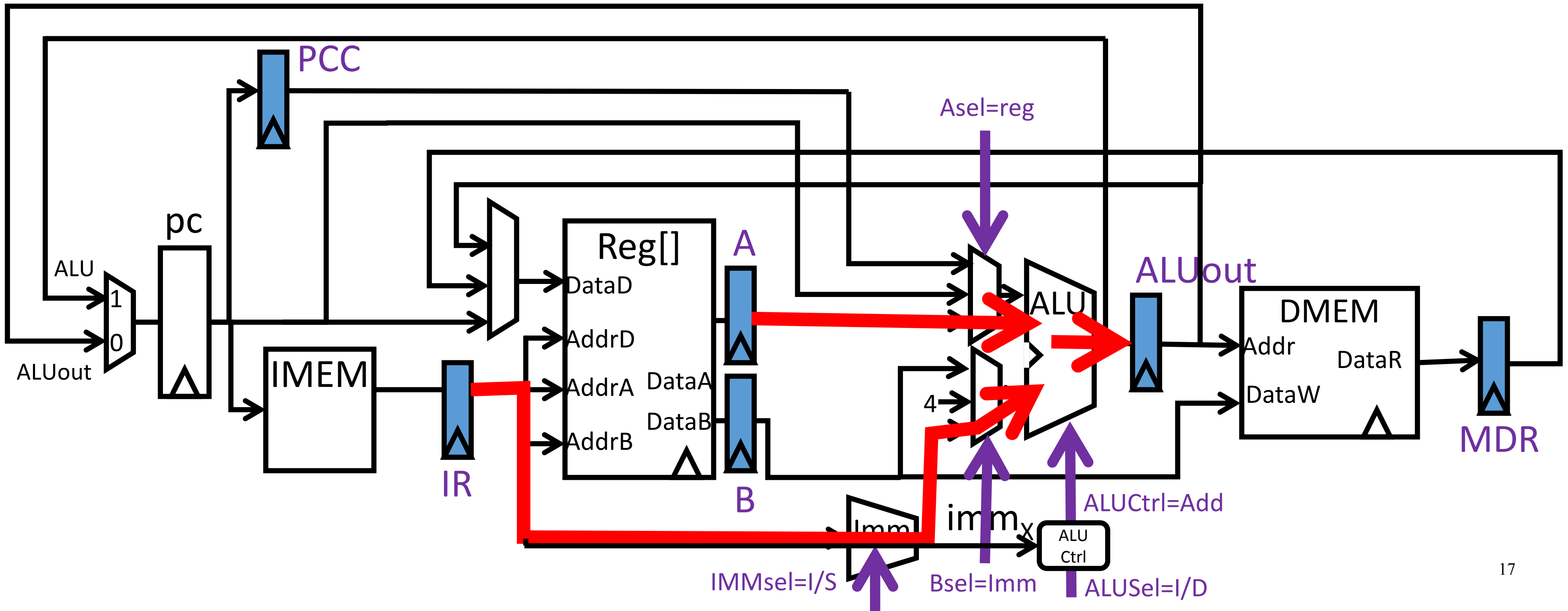
- ALUOut = A *op* B



Execute Cycle for Load / Store

- ALUOut = A + immediate (I or S format)

LW rd , rs1,imm
SW rs1,rs2,imm

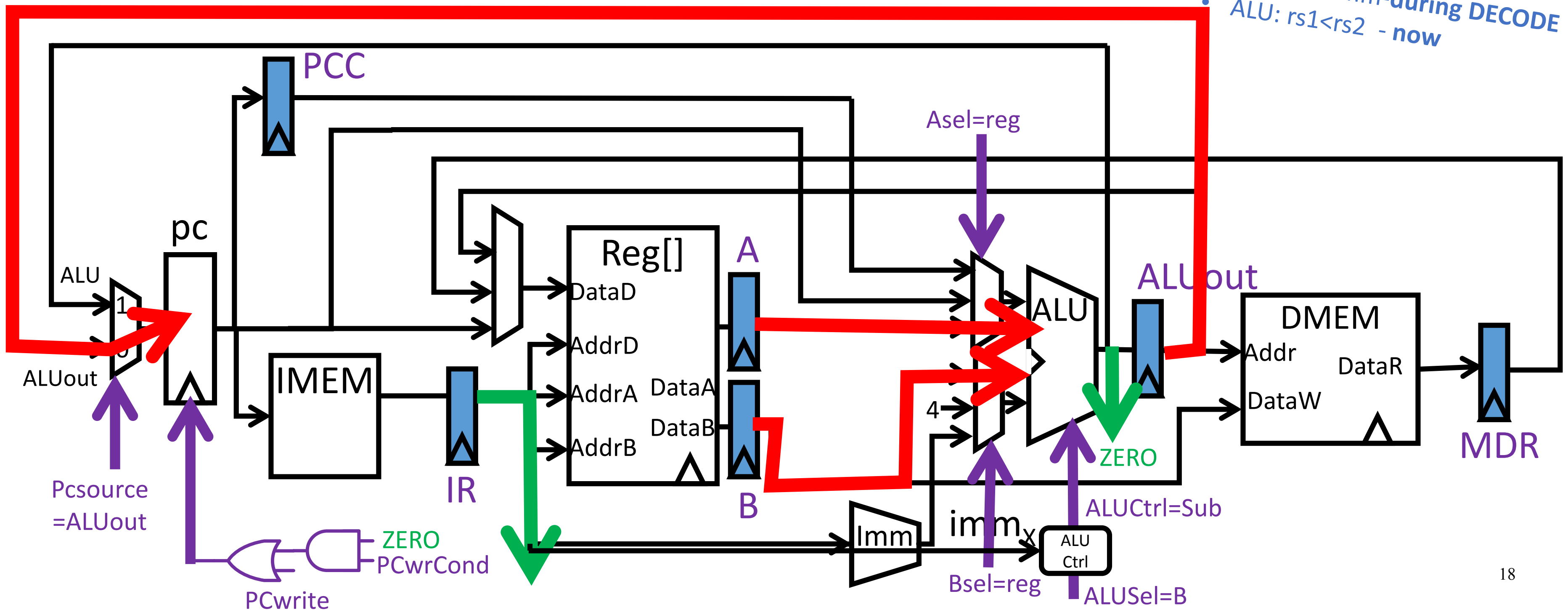


Execute Cycle for Branch

- If (condition) then PC = ALUOut
- Branch condition is checked by ALU and controller (Zero output from ALU)

BLT rs1,rs2,imm

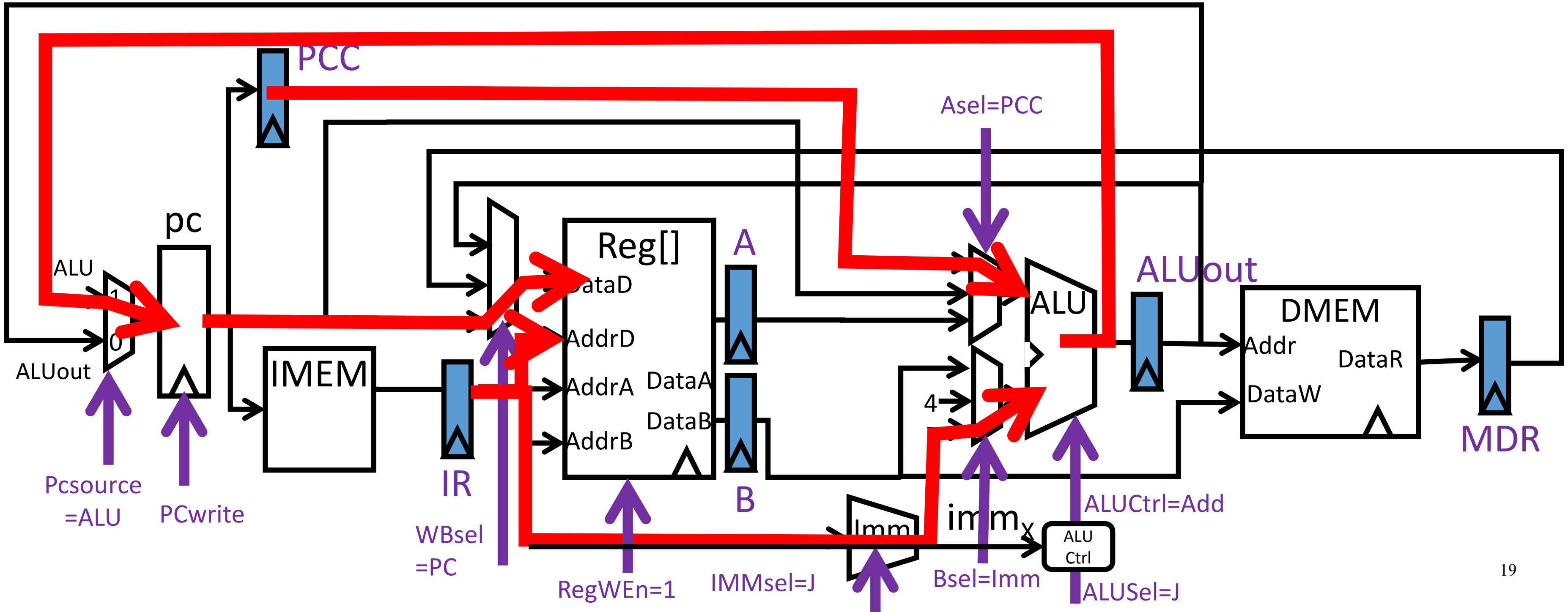
- ALU: PC+imm-**during DECODE**
- ALU: rs1<rs2 - **now**



Execute Cycle for JAL

- Reg[rd] = PC+4 (the return address)
- PC = PCC + offset (PC-relative jump)

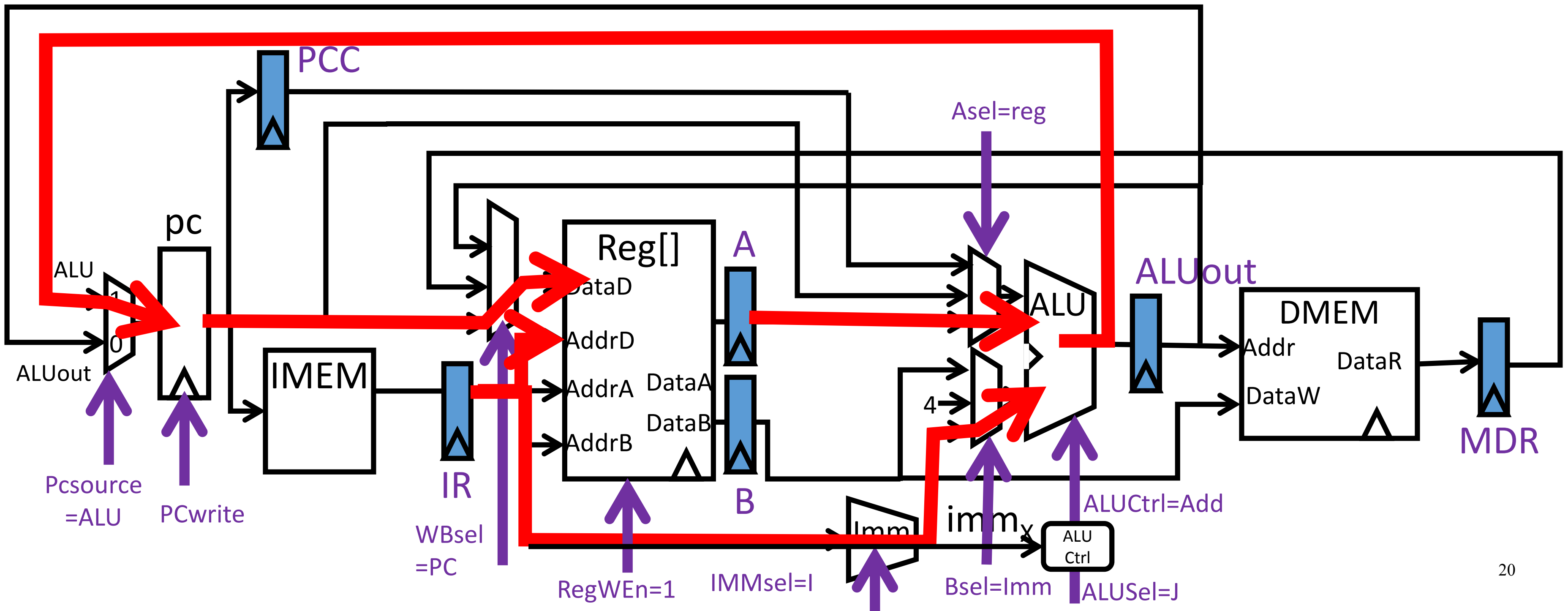
Recall: PC contains PC+4 after the Fetch cycle!



Execute Cycle for JALR

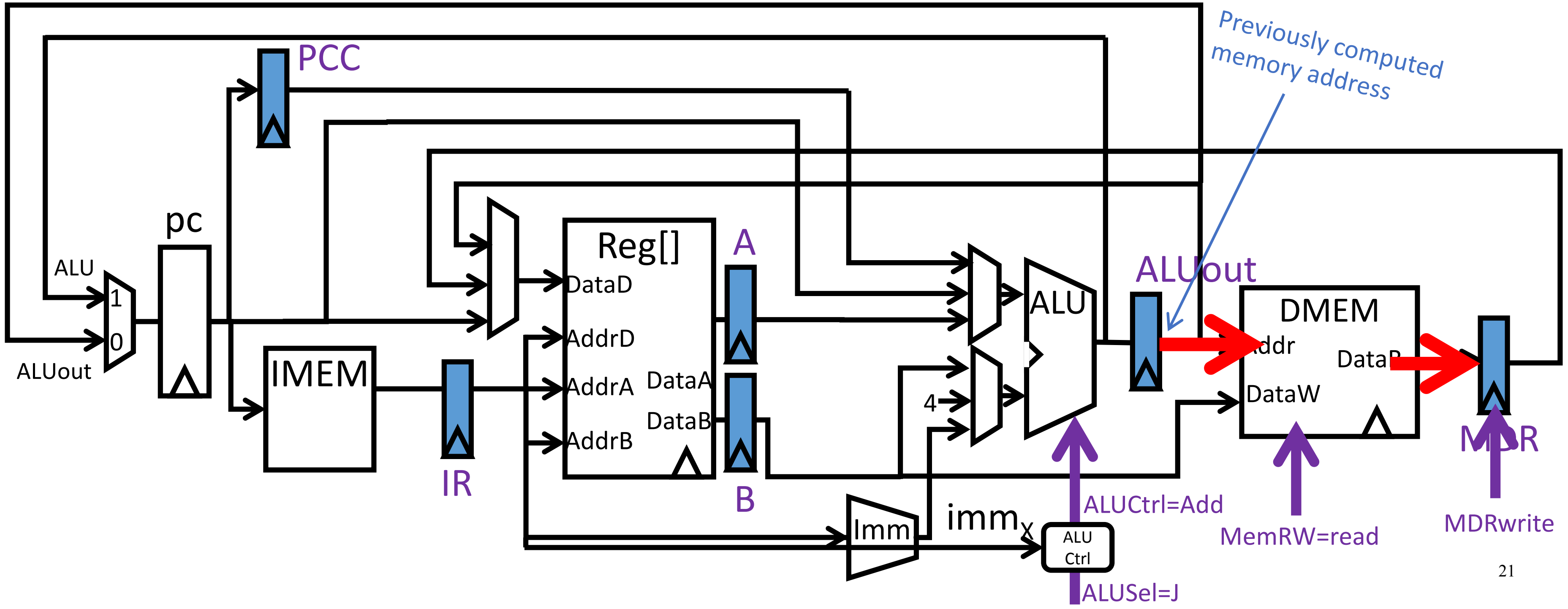
- Reg[rd] = PC+4 (return address)
- PC = Reg[rs1] + immediate

JALR rd,rs1,imm



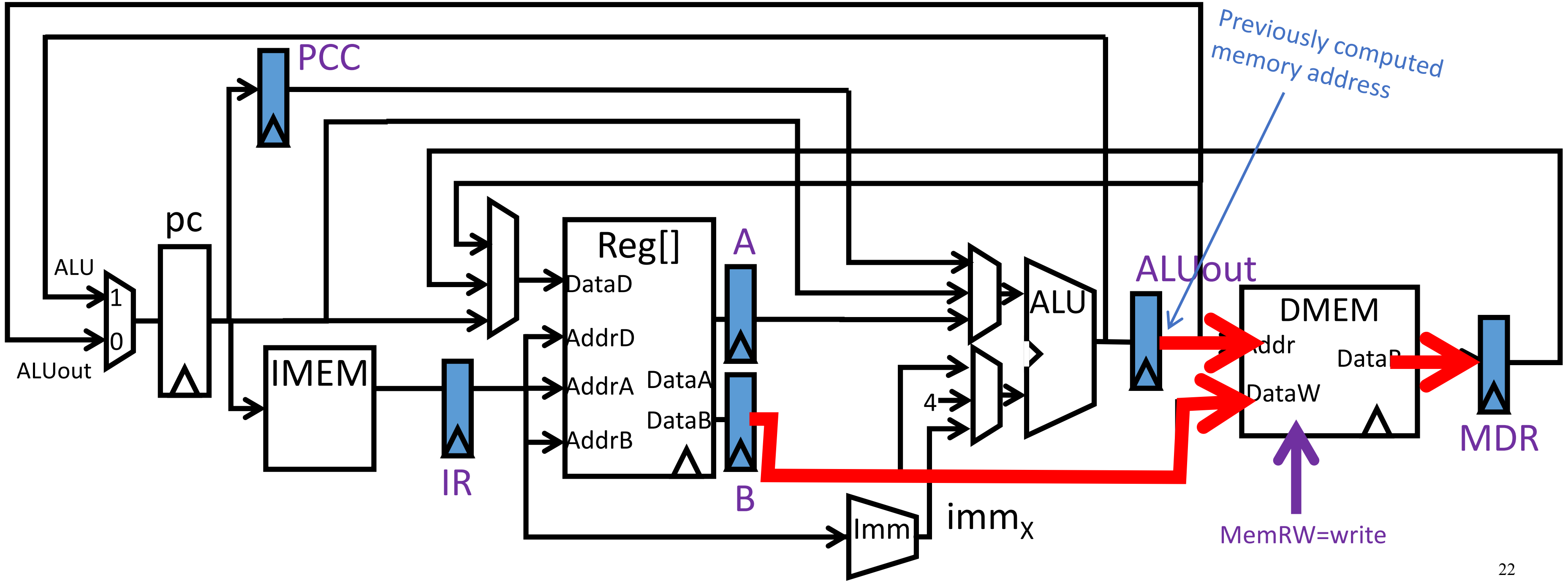
Memory cycle for Load

- MDR = Memory[ALUOut]



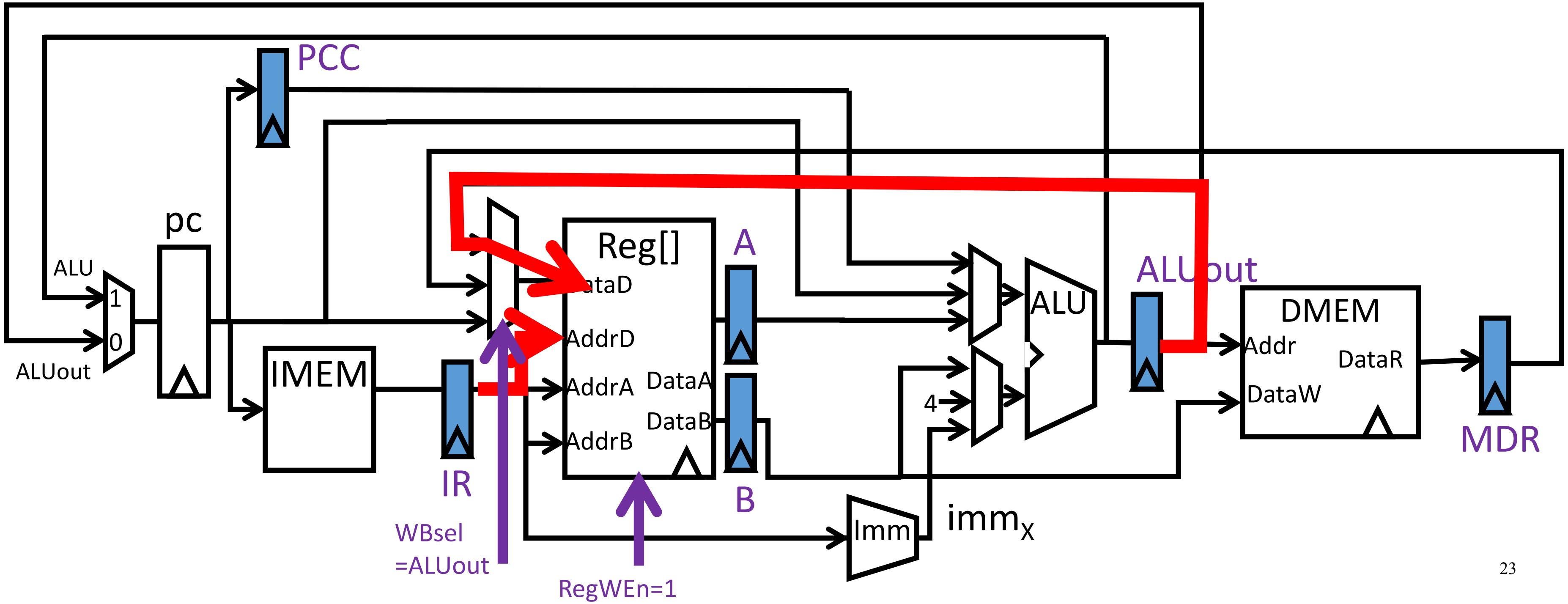
Memory cycle for Store

- Memory[ALUOut] = B



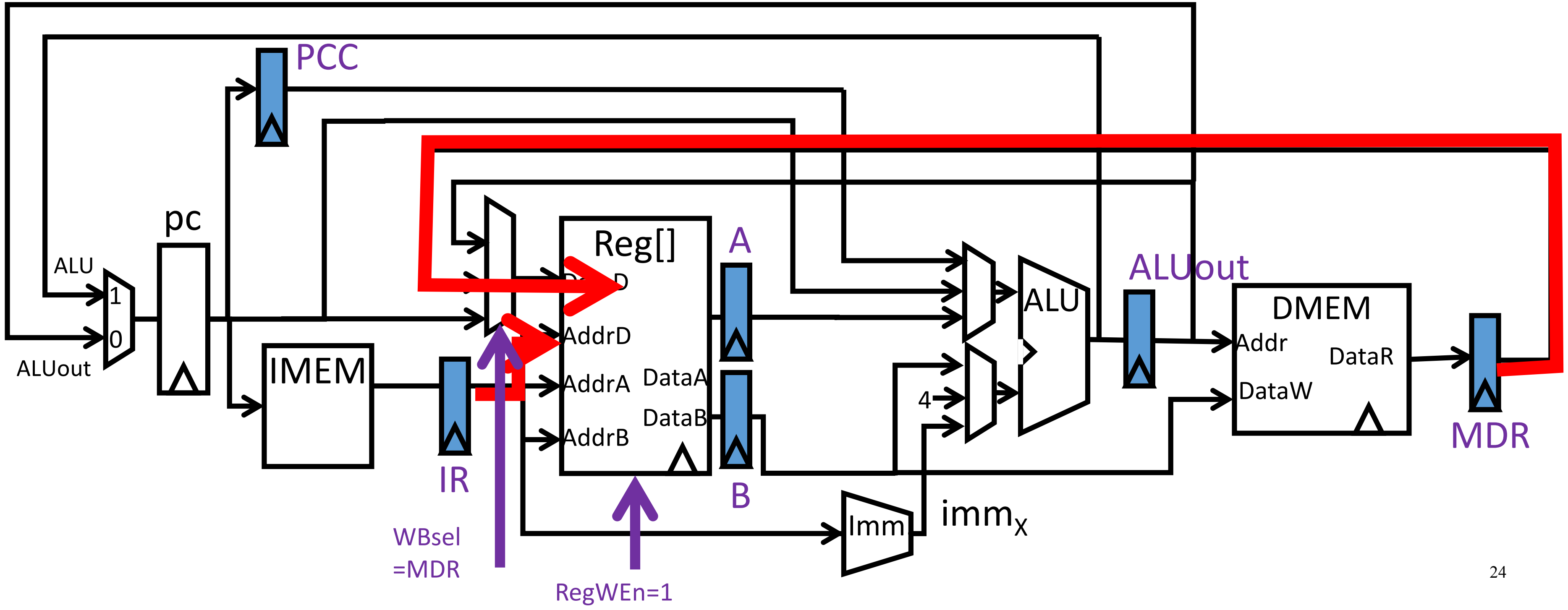
Write-back cycle for R-Type

- Reg[rd] = ALUOut



Write-back cycle for Load

- Reg[rd] = MDR



Summary

Step name	Action for R-type	Action for Load Instruction	Action for Store Instruction	Action for branches	Action for JAL
Instruction fetch	IR = Memory[PC] PC = PC + 4, PCC=PC				
Instruction decode / operand fetch	A = Reg [rs1] B = Reg [rs2] ALUOut = PCC + branch offset				
Execution, address computation, branch/ jump completion	ALUOut = A op B	ALUOut = A + immediate	ALUOut = A + immediate	if (A ==B) then PC = ALUOut	Reg[rd] =PC PC = PCC + immediate
Memory access or R-type write-back	Reg [rd] = ALUOut	MDR = Memory[ALUOut]	Memory [ALUOut] = B		
Load write-back		Reg [rd] = MDR			

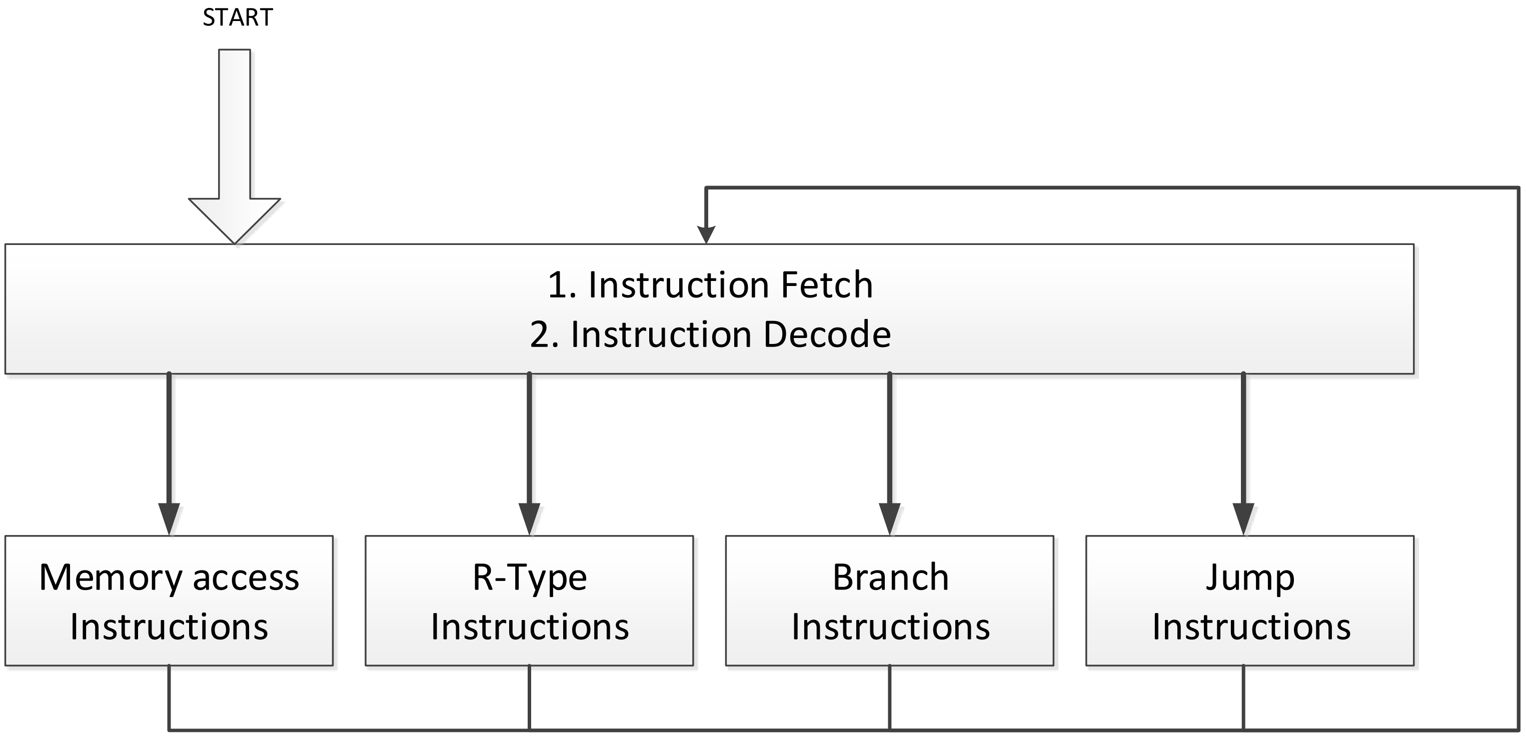
Example Questions

- How many cycles are needed to execute this code ?

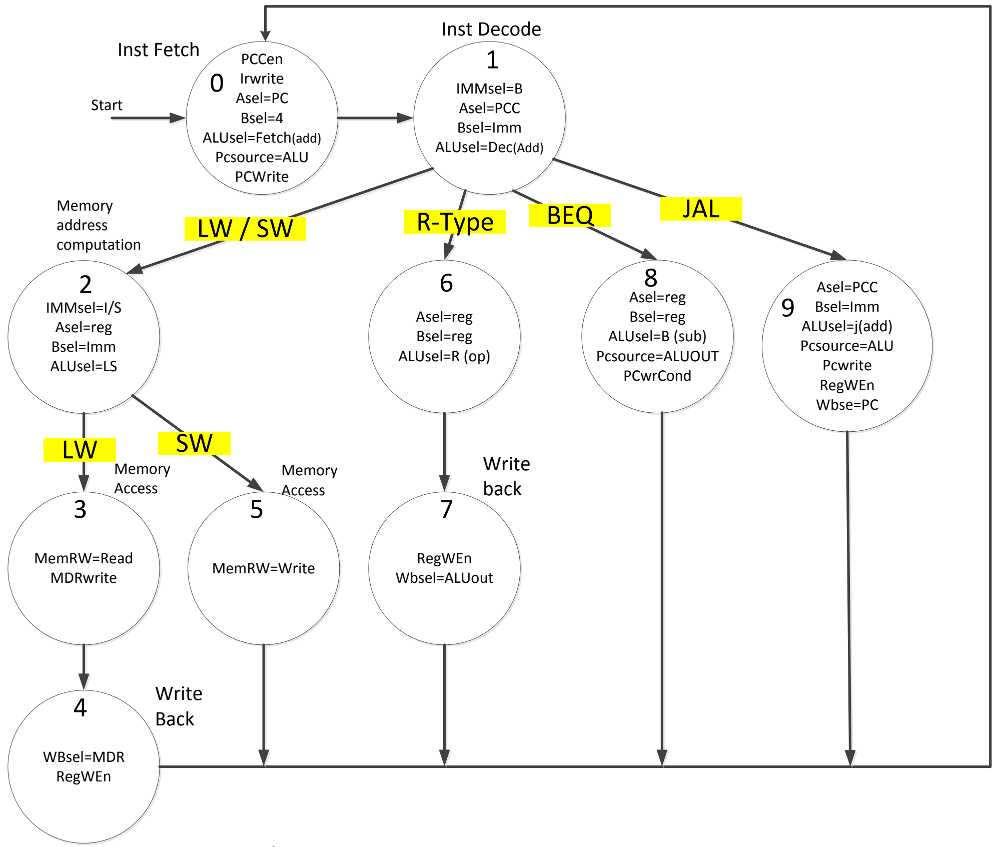
```
       lw    $s2, 0($s3)
       lw    $s3, 4($s3)
       beq   $s2, $s3, Label #assume not
       add   $s5, $s2, $s3
       sw    $s5, 8($s3)
Label:       ...
```

- What happens during the 8th cycle of execution ?
- In which cycle does the adding of `$s2+$s3` take place ?

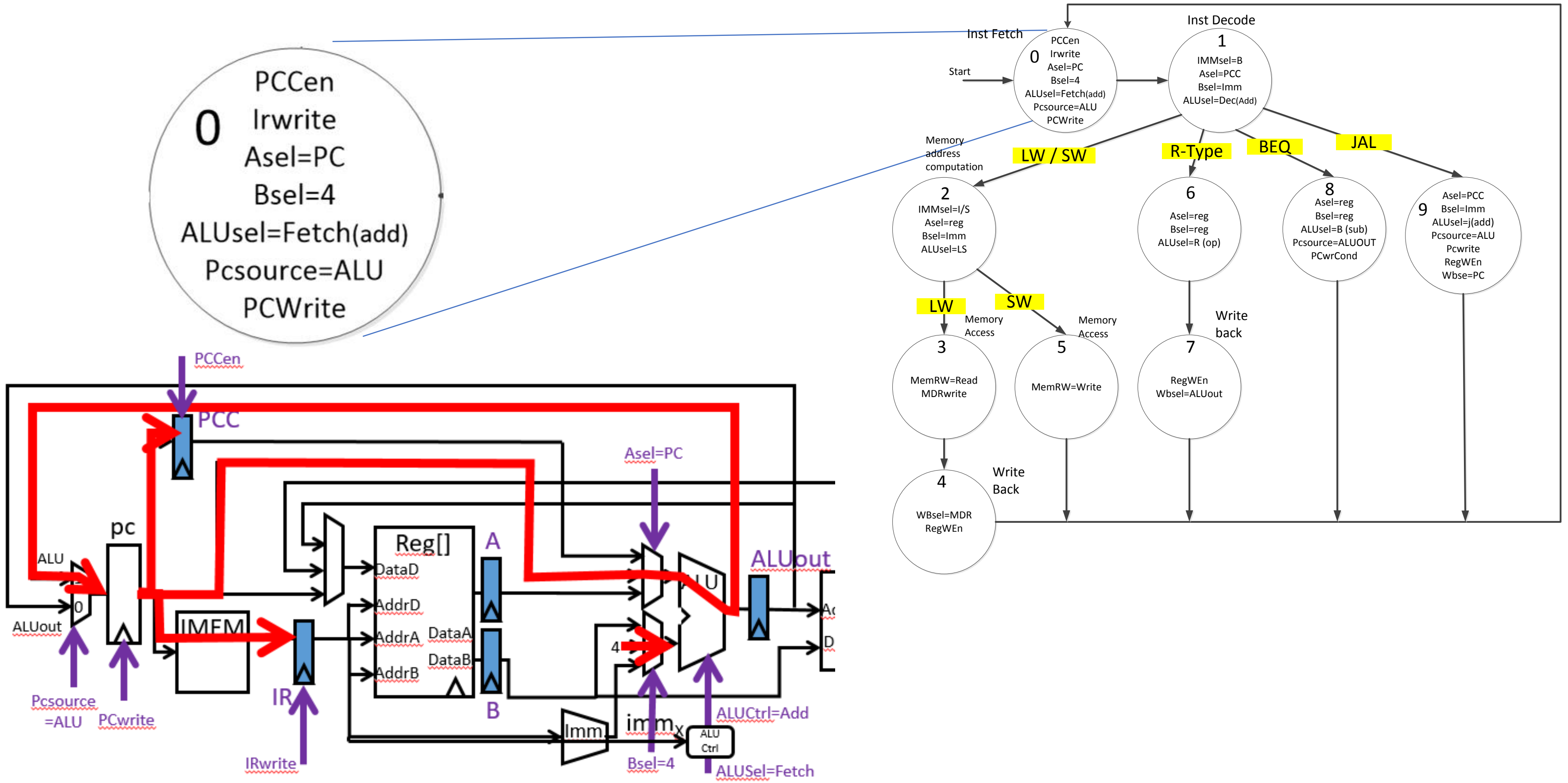
Controller State Machine



Controller State Machine

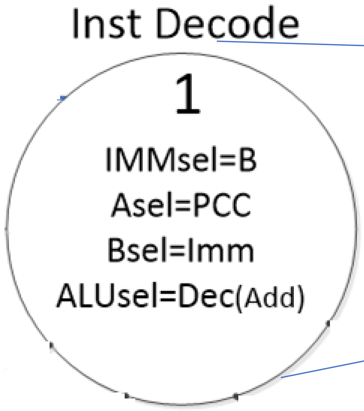


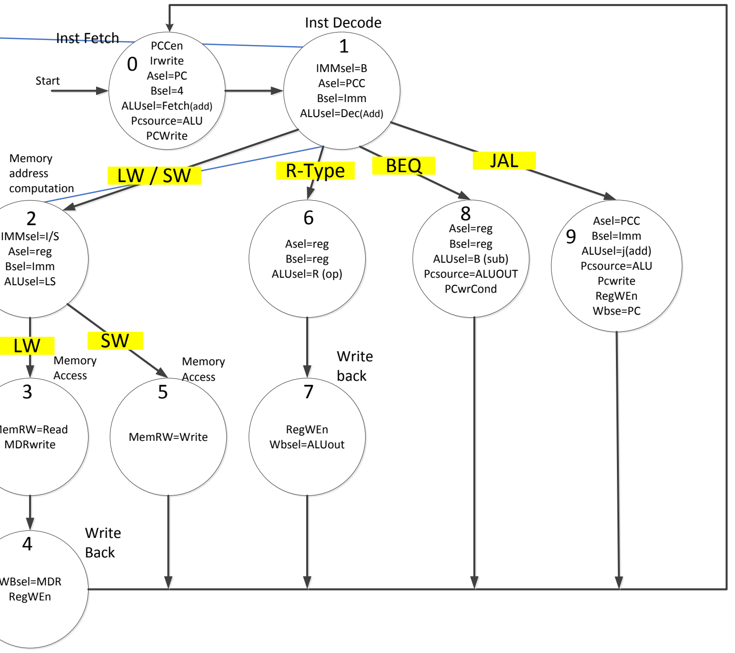
Fetch Cycle



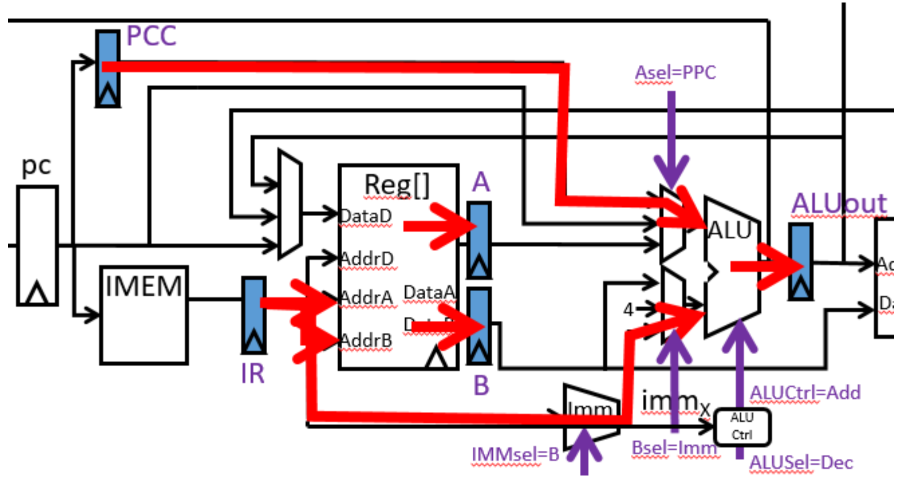
PCCen
Irwrite
Asel=PC
Bsel=4
ALUsel=Fetch(add)
Pcsource=ALU
PCWrite
Inst Fetch
Inst Decode
Start
0
PCCen
Irwrite
Asel=PC
Bsel=4
ALUsel=Fetch(add)
Pcsource=ALU
PCWrite
1
IMMsel=B
Asel=PCC
Bsel=Imm
ALUsel=Dec(Add)
Memory address computation
LW / SW
R-Type
BEQ
JAL
2
IMMsel=I/S
Asel=reg
Bsel=Imm
ALUsel=LS
6
Asel=reg
Bsel=reg
ALUsel=R (op)
8
Asel=reg
Bsel=reg
ALUsel=B (sub)
Pcsource=ALUOUT
PCwrCond
9
Asel=PCC
Bsel=Imm
ALUsel=j(add)
Pcsource=ALU
Pcwrite
RegWEn
Wbse=PC
LW
SW
Memory Access
Memory Access
Write back
3
MemRW=Read
MDRwrite
5
MemRW=Write
7
RegWEn
Wbsel=ALUout
4
WBsel=MDR
RegWEn
Write Back
PCCen
PCC
pc
ALU
ALUout
0
Pcsource =ALU
PCwrite
IMEM
IR
IRwrite
Reg[]
DataD
AddrD
AddrA
AddrB
DataA
DataB
A
B
Asel=PC
ALU
ALUout
4
Imm
immx
ALU Ctrl
Bsel=4
ALUCtrl=Add
ALUSel=Fetch

Instruction Decode / Register Fetch

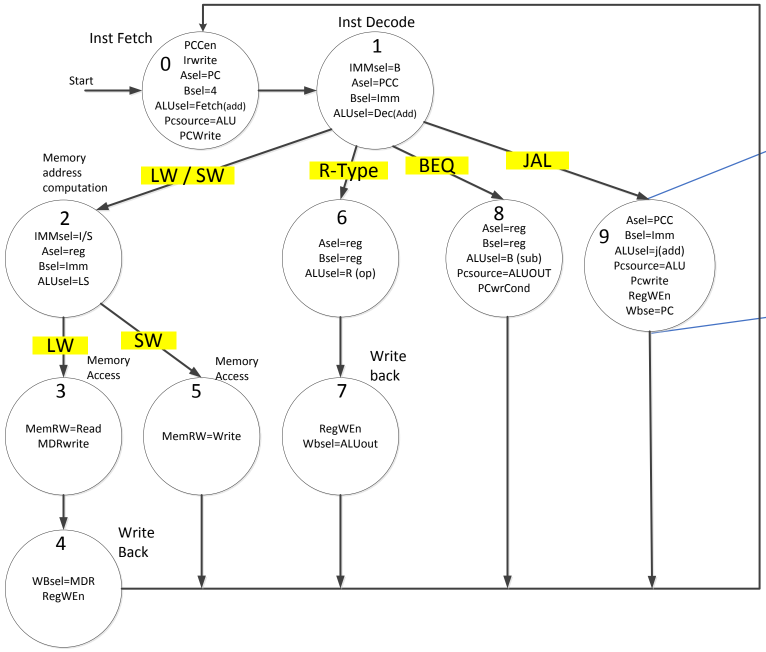


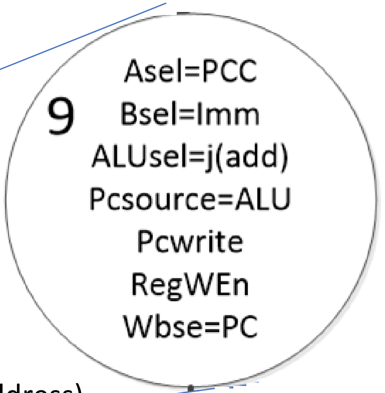


1. A = Reg[rs1], B = Reg[rs2]
2. ALUOut = PCC + branch offset

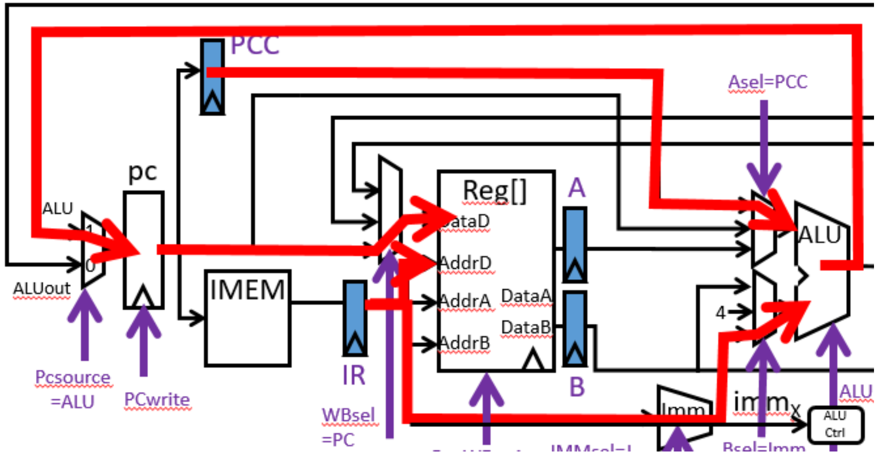


Execute cycle for JAL

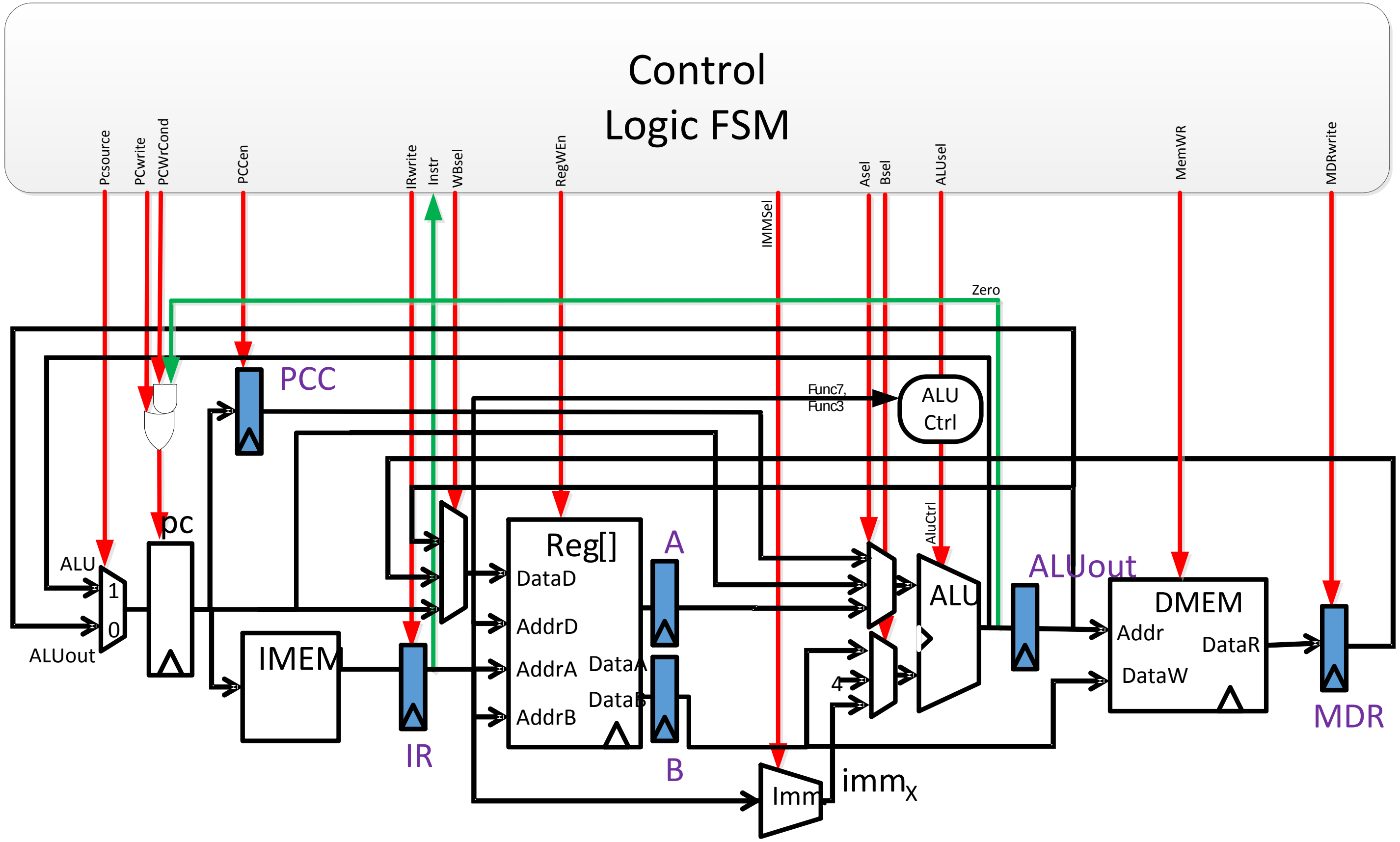




- Reg[rd] = PC+4 (the return address)
- PC = PCC + offset (PC-relative jump)



Multi-Cycle RISC-V Controller and Datapath



Agenda

- Multi-Cycle RISC-V Datapath
- Performance
- ROM Controller
- Micro-Coded Controller

Performance evaluation: Cycles-Per-Instruction

- In the single-cycle RISC-V, CPI=1
- In the multi-cycle RISC-V, CPI depends on instruction:
  - Load: 5 cycles
  - Store: 4 cycles
  - ALU: 4 cycles
  - Branch: 3 cycles
  - Jump: 3 cycles

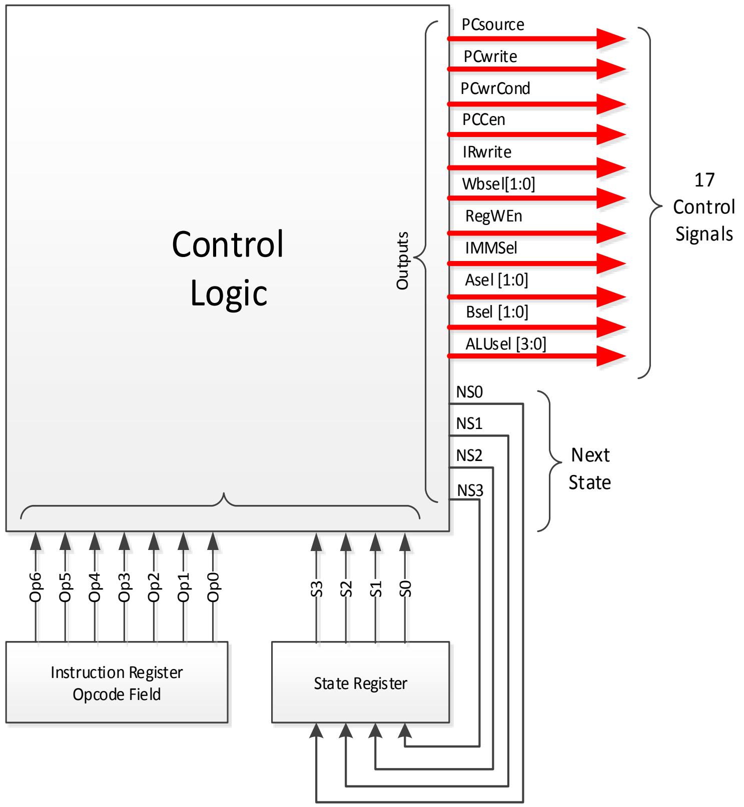
Performance improvement due to multi-cycle

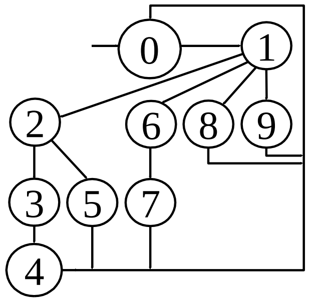
- By instruction dynamic distribution (see slide on Elastic Clock Cycle)
  - CPI = 5×24% + 4×12% + 4×44% + 3×18% + 3×2% = 4.02 cycles/instruction (on average)
- 4.02 is better than worst case of 5 (all instructions use 5 cycles)
- BUT apparently single-cycle RISC-V is "better" – CPI=1…
- Not really. Instruction TIME matters: Instruction Time = Cycle time × CPI
- What was Cycle Time in single-cycle RISC-V ? 800 ps
- What is Cycle Time in multi-cycle RISC-V ?
  - We split execution to five parts. Ideally, Cycle Time = 800 ps / 5 = 160 ps
  - But practice is never ideal. Hard to split exactly equal
  - Extra registers may also add certain overhead (setup time, clock-to-output time)
- If Cycle Time is indeed 800/5=160 ps, then Instruction Time = Cycle time × CPI = 160ps × 4.02 = 643.2 ps
- In this ideal case, improvement is 800/643.2 = 5/4.02 = 1.24 (multi-cycle 24% faster than single cycle)

Agenda

- Multi-Cycle RISC-V Datapath
- Performance
- ROM Controller
- Micro-Coded Controller

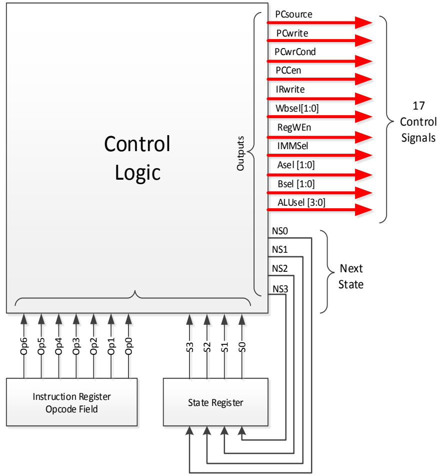
Implementing the FSM

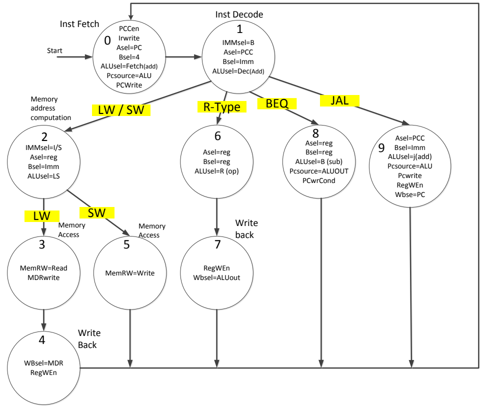




Controller implemented as a ROM

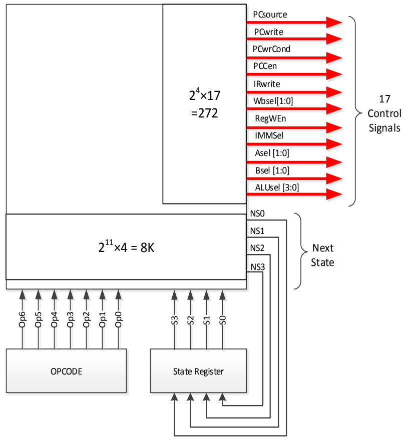
- Inputs
  - 7 opcode bits
  - 4 state bits
  - Total 11 bits → 2^{11}=2048 words
- Outputs
  - 17 control bits
  - 4 state bits
  - Total 21 bits per word
- ROM size 2048 × 21 = <u>42 Kbits</u>

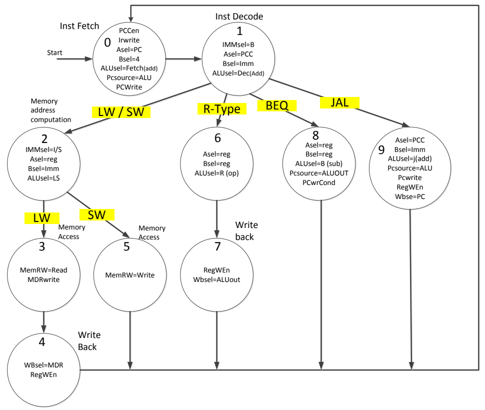




Reducing ROM size

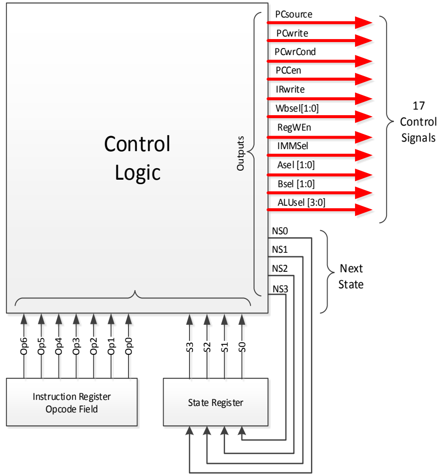
- Break ROM into two
  - Moore FSM→17 control outputs are function of only 4 state bits: (2^4 words)×(17 bits/word)=272 bits
  - 11 inputs (7 op + 4 state) determine next state: (2^{11} words)×(4 bits/word)=8K bits
  - Total 8.25 Kbits (cf. 42 Kbits)

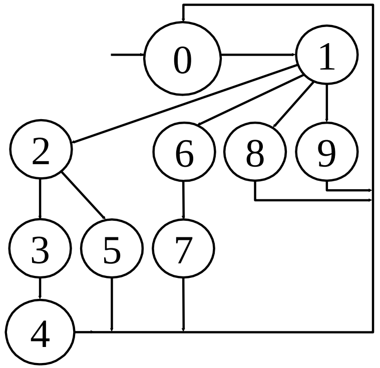




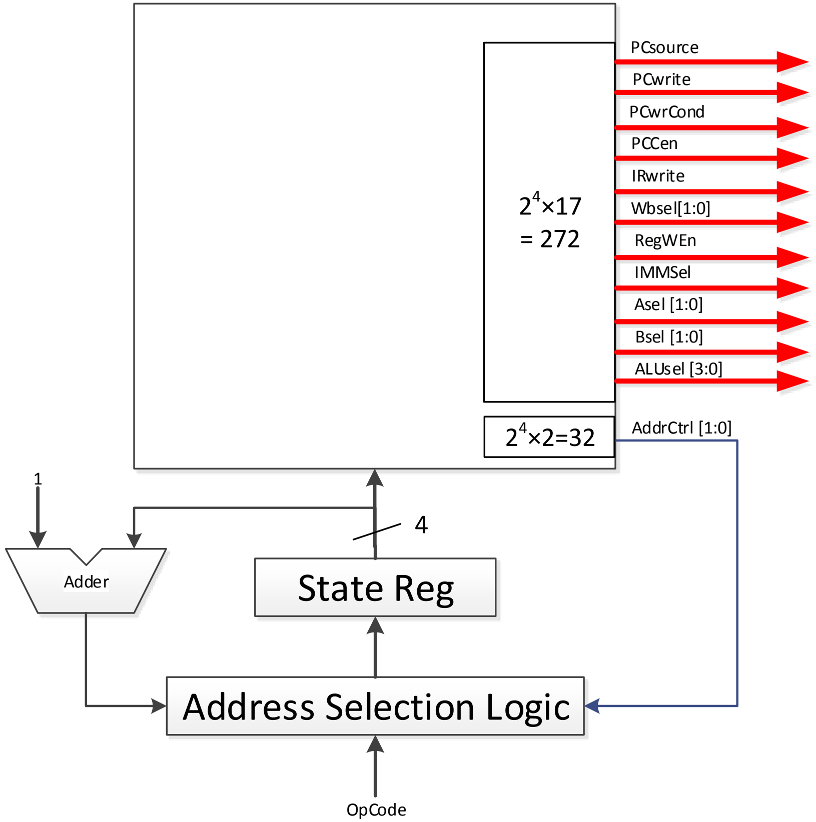
Optimizing FSM

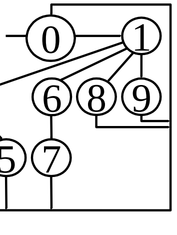
- ROM is non-optimal
  - Contains all combinations, some of them useless
  - Often we know exactly what next state is
  - Often, next state is next word in the ROM
  - In these cases, we don't need the next state bits
- We could have minimized the logic and use gates….
- Better idea: Use counter, increment +1 for next state (often)
- Often, but not always. So add a new control signal (use counter or not) [AddrCtrl]
- VERY similar to the PC
  - Often go to PC+4, otherwise jump/branch





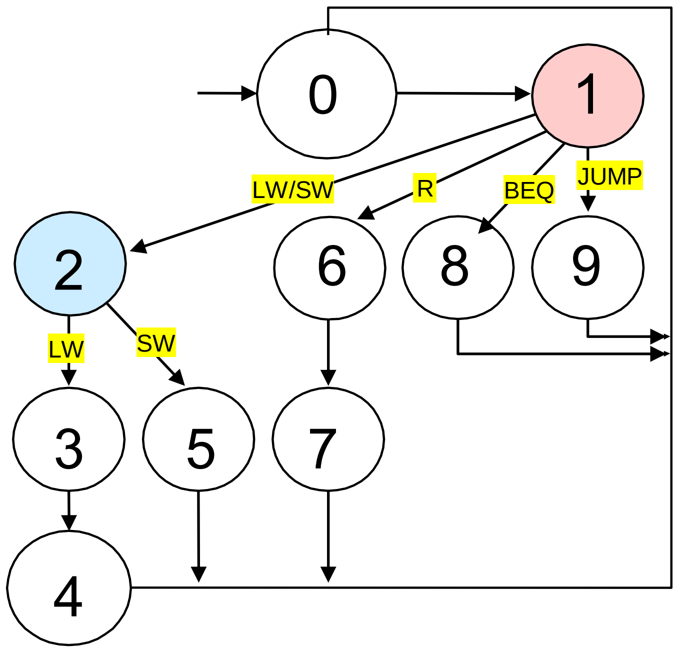
Added counter for next state



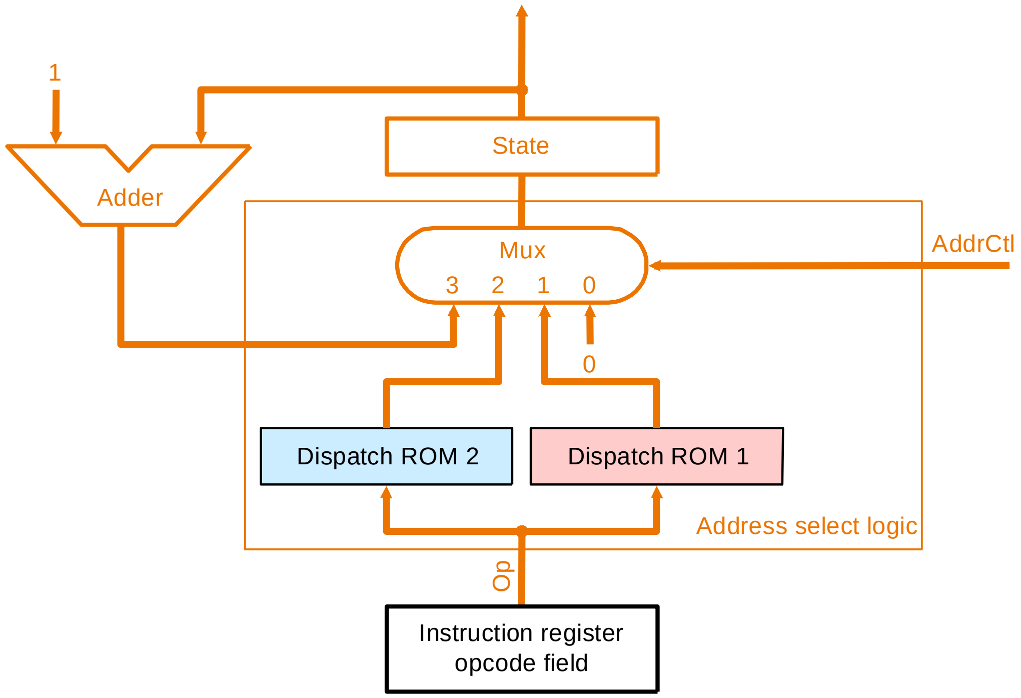


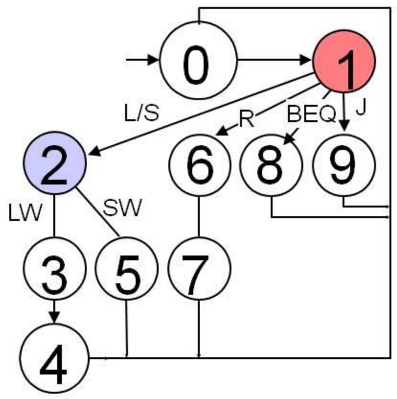
What if next state is not state+1 ?

- State diagram branches in 1, 2
- Use branch table for 1
- Use branch table for 2
- Go to zero after 4,5,7,8,9
- Go to state+1 after 0,3,6



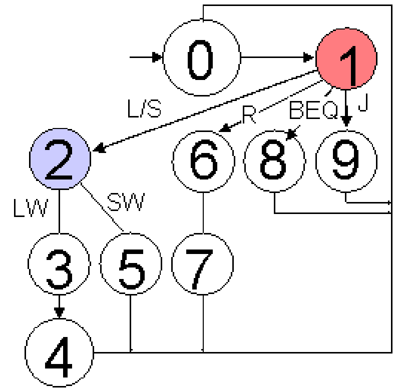
Address Selection





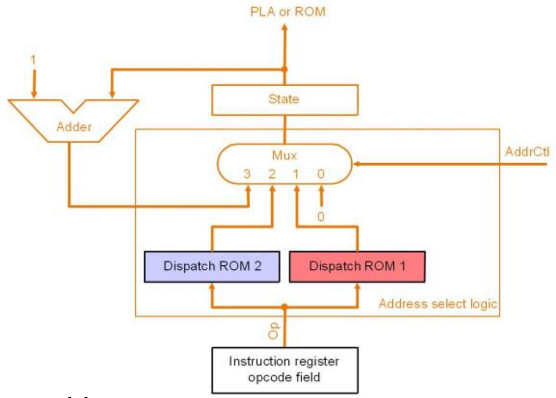
Address Selection Tables

- טבלה 1 לקפיצה ממצב מספר 1
- טבלה 2 לקפיצה ממצב מספר 2



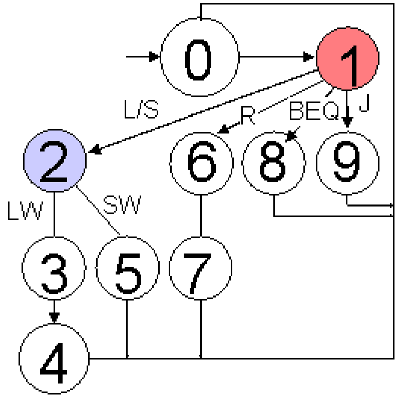
Dispatch ROM 1		
Op	Name	Value
x33	R-type	6
x67	JAL	9
x63	BEQ	8
x03	LW	2
x23	SW	2

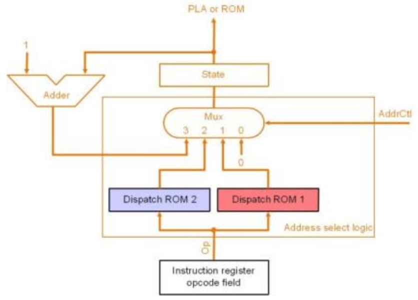
Dispatch ROM 2		
Op	Name	Value
x03	LW	3
x23	SW	5



Address Selection Control

State number	Address control action	Value of AddrCtl
0	Increment	3
1	Use dispatch ROM 1	1
2	Use dispatch ROM 2	2
3	Increment	3
4	Go to state 0	0
5	Go to state 0	0
6	Increment	3
7	Go to state 0	0
8	Go to state 0	0
9	Go to state 0	0



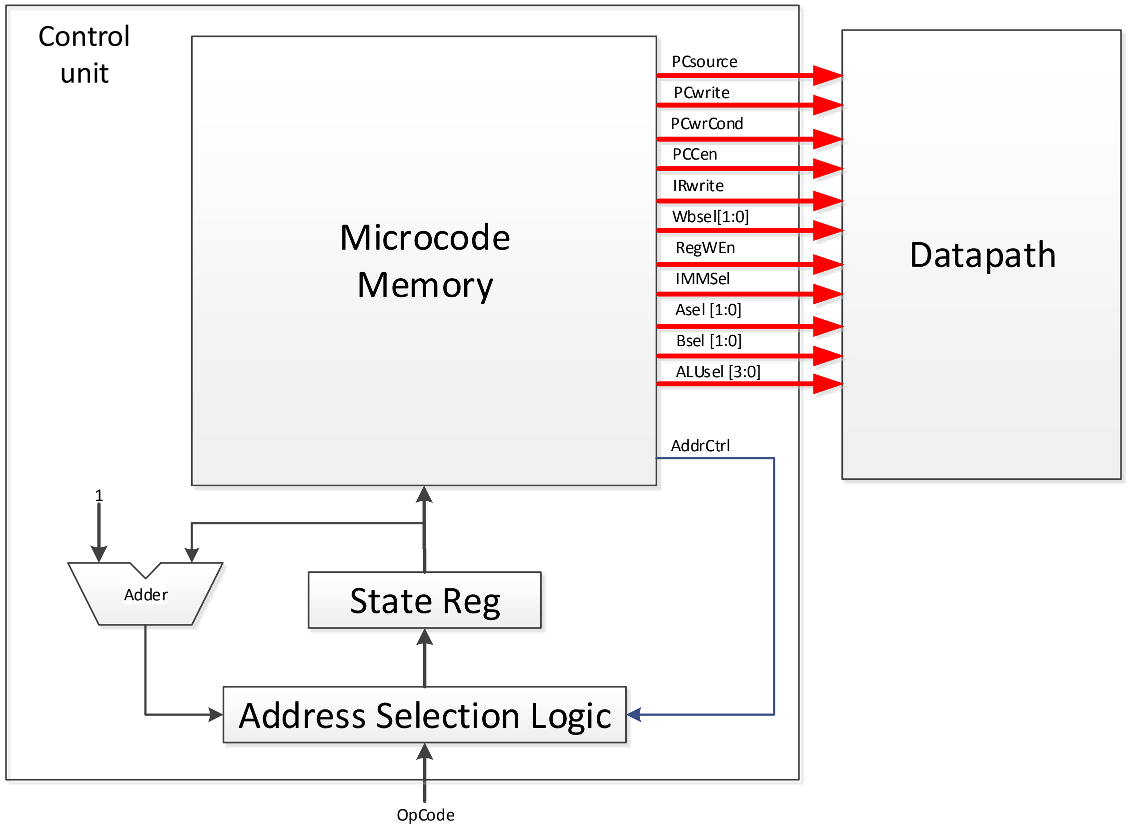


Result is smaller than ROM—no 4 bits/word for next state, only two bits

Agenda

- Multi-Cycle RISC-V Datapath
- Performance
- ROM Controller
- Micro-Coded Controller

Micro-code Controller



This is the modified ROM controller

Let’s change ROM to RAM

Now we can change control RAM

This is Microcode / Microprogram

Micro-code

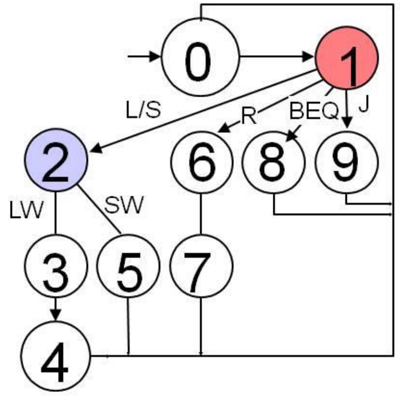
- μPC is analogous to PC
- μCode is analogous to Code
- μInstruction is analogous to Instruction

- Multiple μInstructions are (micro-) executed to implement one instruction
- Often, easier to make complex instructions by μCode section than by state diagram
  - Large state diagrams are hard to read (hence, error prone)
  - μCode is symbolic, translated by μAssembler. No need to manage bits
- Most complex computers implemented that way

Micro-coded RISC-V controller

- Symbolic µcode
- Tabulated

State	Label	ALUsel	Asel	Bsel	MemRW	PCsource	WBsel	Sequencing
0	Fetch	Fetch	PC	4		ALU		Seq
1		Dec	PCC	Imm				Dispatch 1
2	Mem1	LS	Reg	Imm				Dispatch 2
3	LW2				Read			Seq
4							MDR	Fetch
5	SW2				Write			Fetch
6	Rformat1	R	Reg	Reg				Seq
7								Fetch
8	BEQ1	B	Reg	Reg		ALUout		Fetch
9	JAL	J	PCC	Imm		ALU	PC	Fetch



Note
This table is only partial

μInstuction Format (partial)

ALU control	Fetch	ALUCtrl = 000	Cause the ALU to add for Fetch stage.
	Dec	ALUCtrl = 001	Cause the ALU to add for decode stage.
	LS	ALUCtrl = 010	Cause the ALU to add for Load/Store instructions
	B	ALUCtrl = 011	Cause the ALU to subtract; this implements the compare for branches.
	J	ALUCtrl = 100	Cause the ALU to add for jump target address computation
	R	ALUCtrl = 101	Use the instruction's function code (Func3/7) to determine ALU control.
	…		
Asel	PC	Asel = 01	Use the PC as the first ALU input.
	PCC	Asel = 10	Use the PCC as the first ALU input.
	Reg	Asel = 00	Register A is the first ALU input.
Bsel	B	BSel = 00	Register B is the second ALU input.
	4	BSel = 01	Use 4 as the second ALU input.
	Imm	BSel = 10	Use output of the sign extension unit as the second ALU input.
Register control	Read	RegWEn=0	Read two registers using the rs1 and rs2 fields of the IR as the register numbers and putting the data into registers A and B.
	Write ALUout	RegWEn= 1, WBsel = 00	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	RegWEn= 1, WBsel = 01	Write a register using the rd field of the IR as the register number and the contents of the MDR as the data.
	Write PC	RegWEn= 1, WBsel = 10	Write a register using the rd field of the IR as the register number and the contents of the PC as the data.
Memory (DMEM)	Read DMEM	MemRW=0 MDRwrite=1	Read memory using address calculated in EXE stage and sittinh in ALUout Save the read data into MDR
	Write DMEM	MemRW=1 MDRwrite=0	Write to memory using the ALUOut as address and B register as data
PC write control	ALU	PCSource = 1, PCWrite=1	Write the output of the ALU into the PC. For PC+4, JAL address, etc.
	ALUOut	PCSource = 0, PCWrite=1	If the Zero output of the ALU is active, write the PC with the contents of the register ALUOut.
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.

Conclusions

- Single cycle RISC-V
  - Lots of hardware (dedicated to functionality)
  - Slow
  - Combinational control
- Multi-cycle RISC-V
  - Hardware savings (but added registers)
  - Faster
  - Some speculation – more energy consumed
  - Sequential control (ROM / micro-code)
  - Flexible: see homework. We later exploit this to show interrupts