

# Hardwired, Non-pipelined ISA Implementation

*Hyun Ryong (Ryan) Lee*

Computer Science & Artificial Intelligence Lab  
M.I.T.

# Administrivia

---

- TA: Ryan Lee
- Contact: 6823-tas@lists.csail.mit.edu  
or hrlee@csail.mit.edu
- Please use Piazza extensively for questions!
- Office Hours: Wed. 4-5:30pm, 32-G725 or by appointment
- Tutorials every week (optional)
  - First two will cover background materials
  - Cover lab tools (Intel Pin, Murphi)
  - Go over problem sets, quiz prep
  - Two sessions will be reserved for Quizzes. Do not miss them!

# Last Lecture...

---

- Computer Architecture as designing under constraints
  - Spans much of the computing stack nowadays
- Looked at some early computers and their instruction sets (e.g., ENIAC)
  - Instruction set tightly coupled to the technology
  - Need for compatibility, well-defined interfaces
    - > Instruction Set Architecture (ISA)
    - > Will see these in detail next lecture!
- Today: single-cycle implementation of a processor

# Processor Performance

---

$$\frac{\text{Time}}{\text{Program}} = \frac{\text{Instructions}}{\text{Program}} * \frac{\text{Cycles}}{\text{Instruction}} * \frac{\text{Time}}{\text{Cycle}}$$

- Instructions per program depends on source code, compiler technology and ISA
- Cycles per instructions (CPI) depends upon the ISA and the microarchitecture
- Time per cycle depends upon the microarchitecture and the base technology

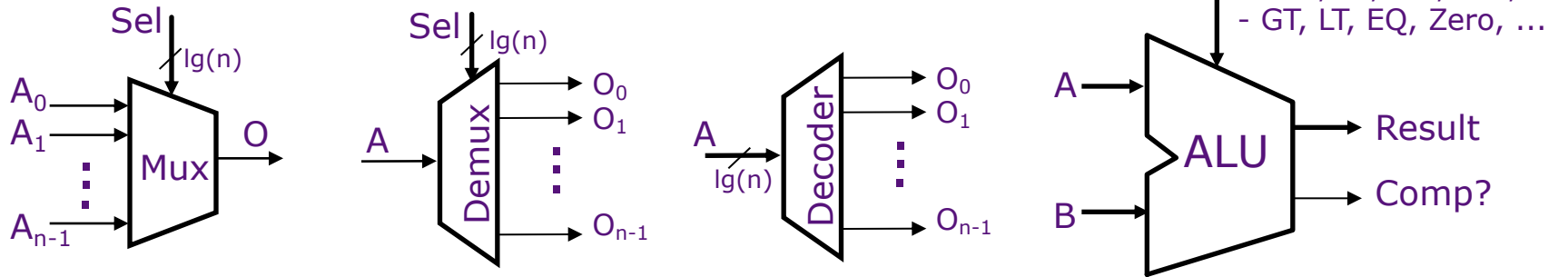
rest of  
this lecture  
→

Microarchitecture	CPI	cycle time
Microcoded	> 1	short
Single-cycle unpipelined	1	long
Pipelined	1	short

# Hardware Elements

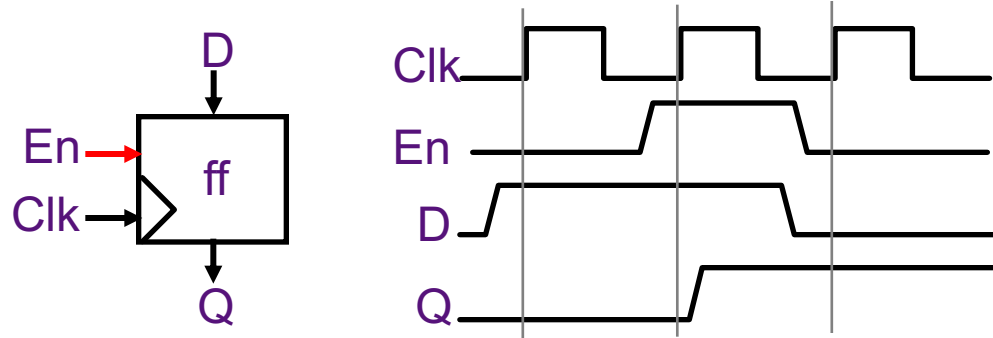
- Combinational circuits

- Mux, Demux, Decoder, ALU, ...



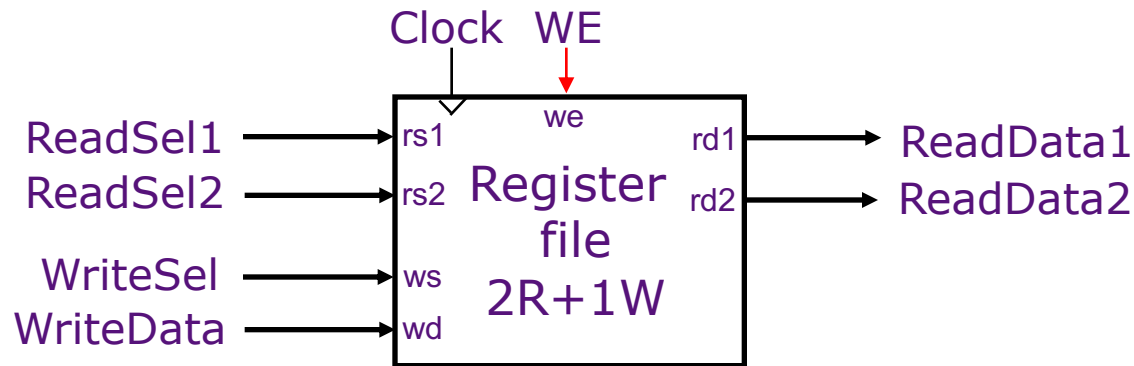
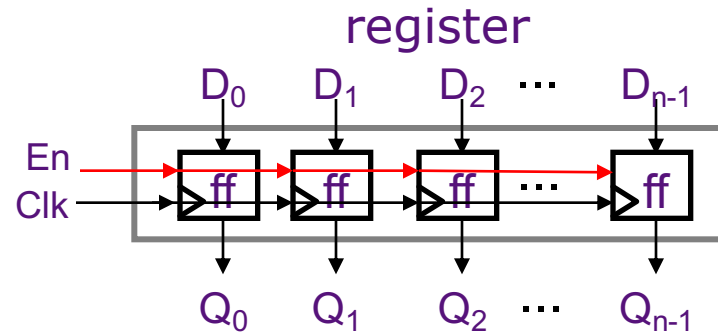
- Synchronous state elements

- Flipflop, Register, Register file, SRAM, DRAM



*Edge-triggered: Data is sampled at the rising edge*

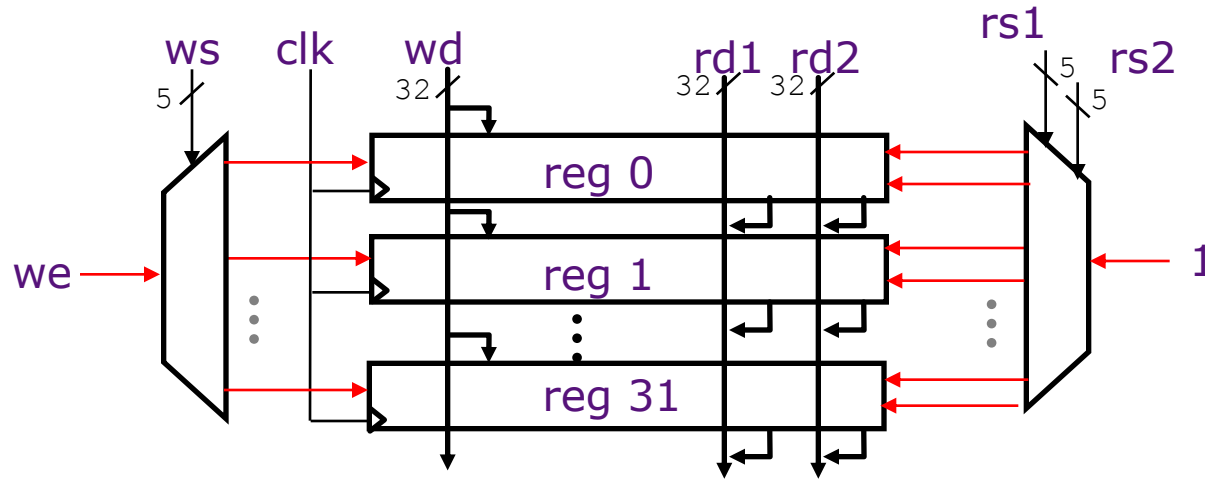
# Register Files



No timing issues when reading and writing the same register  
(writes happen at the end of the cycle)

# Register File Implementation

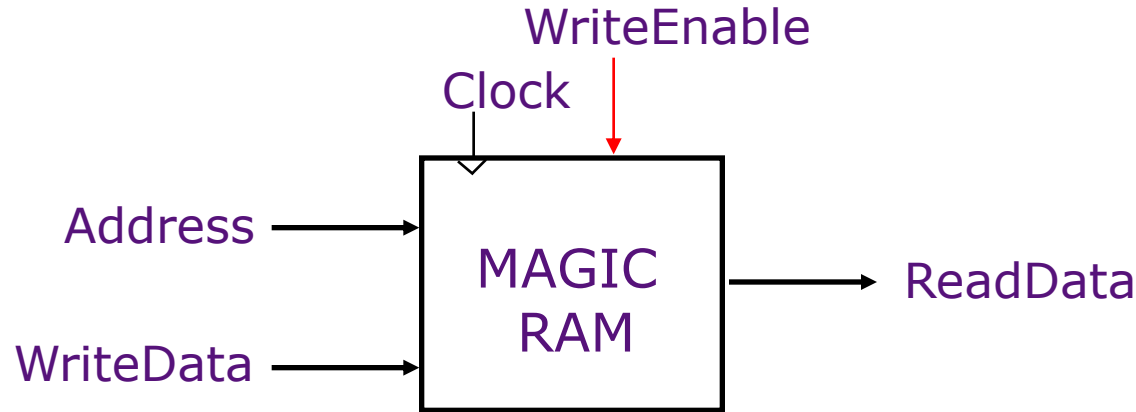
---



- Register files with a large number of ports are difficult to design
  - Area scales with ports<sup>2</sup>
  - Almost all Alpha instructions have exactly 2 register source operands
  - *Intel's Itanium GPR File has 128 registers with 8 read ports and 4 write ports!!*

# A Simple Memory Model

---



- Reads and writes are always completed in one cycle
  - A Read can be done any time (i.e., combinational)
  - If enabled, a Write is performed at the rising clock edge  
(*the write address and data must be stable at the clock edge*)

*Later in the course we will present a more realistic model of memory*



# Implementing MIPS: Single-cycle per instruction datapath & control logic

# The MIPS ISA

---

## Processor State

- 32 32-bit GPRs, R0 always contains a 0
- 32 single precision FPRs, may also be viewed as
  - 16 double precision FPRs
- FP status register, used for FP compares & exceptions
- PC, the program counter
- Some other special registers

## Data types

- 8-bit byte, 16-bit half word
- 32-bit word for integers
- 32-bit word for single precision floating point
- 64-bit word for double precision floating point

## Load/Store style instruction set

- Data addressing modes: immediate & indexed
- Branch addressing modes: PC relative & register indirect
- Byte-addressable memory, big-endian mode

**All instructions are 32 bits**

# Instruction Execution

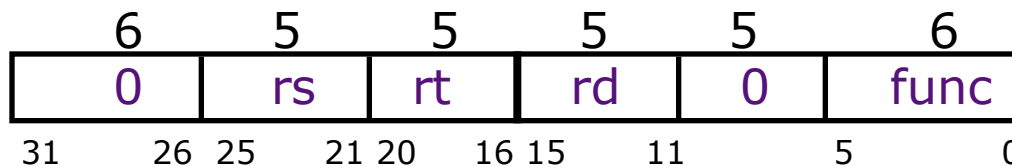
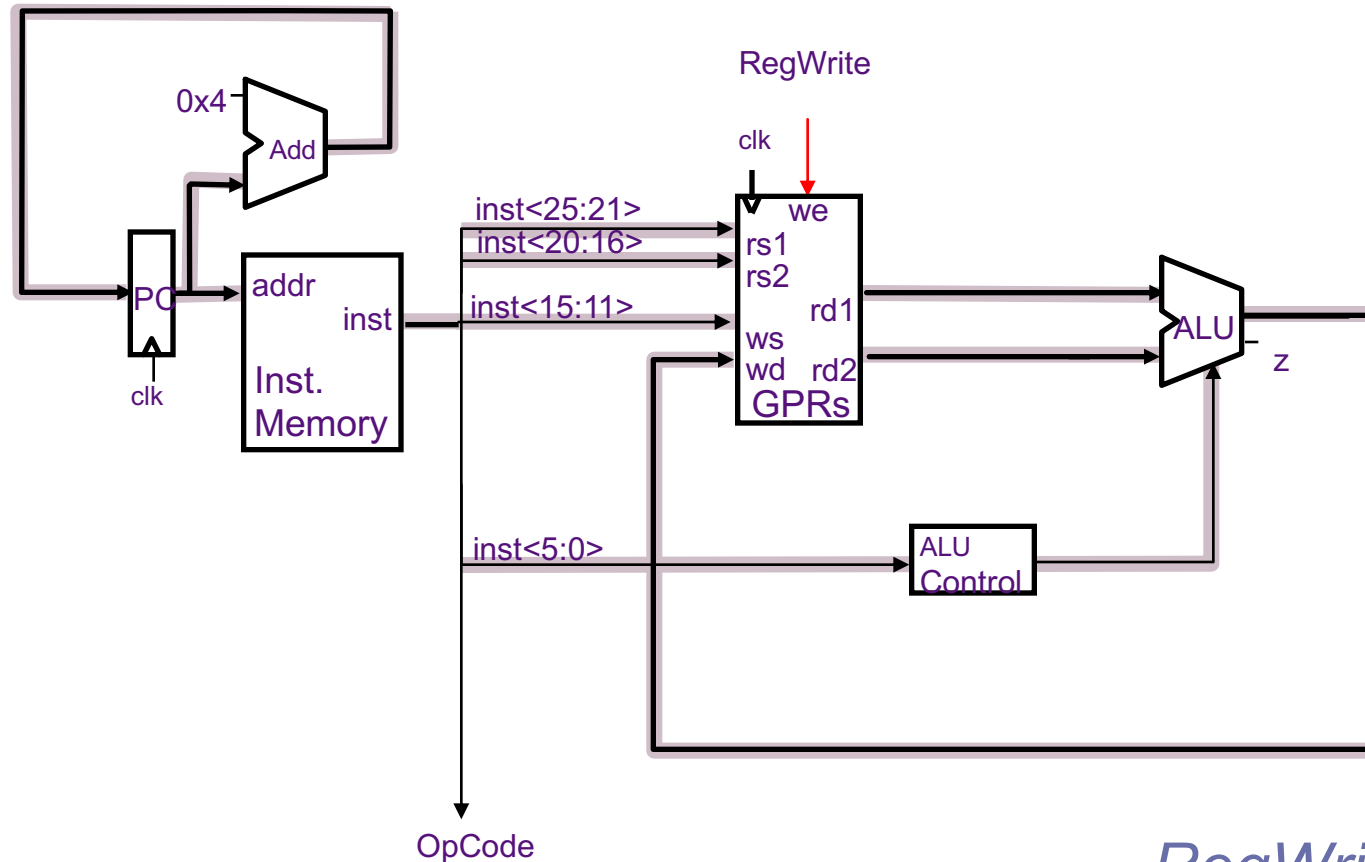
---

Execution of an instruction involves

1. Instruction fetch
2. Decode
3. Register fetch
4. ALU operation
5. Memory operation (optional)
6. Write back

And computing the address of the  
*next instruction (next PC)*

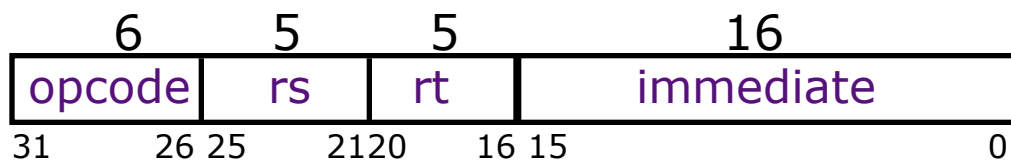
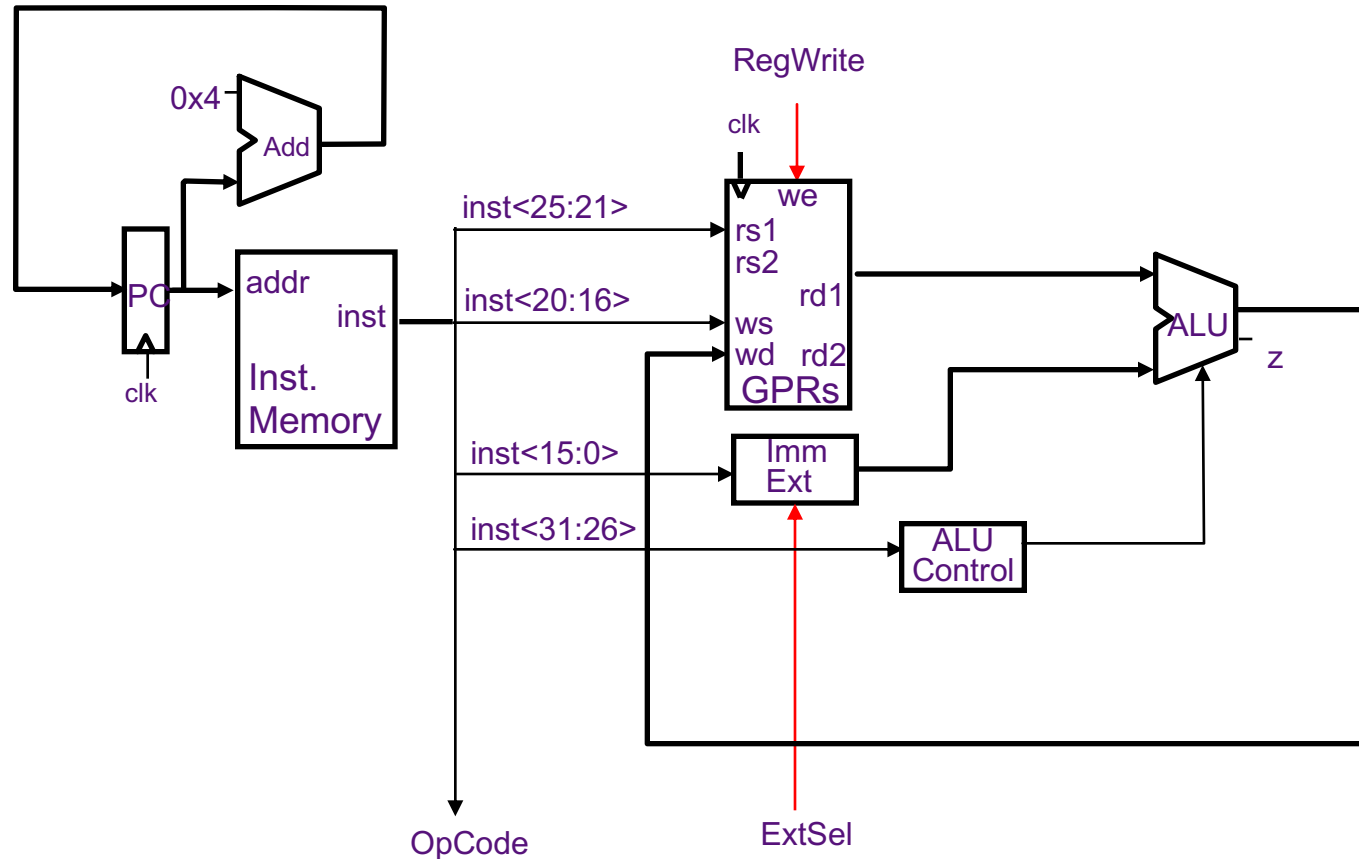
# Datapath: Reg-Reg ALU Instructions



*RegWrite Timing?*

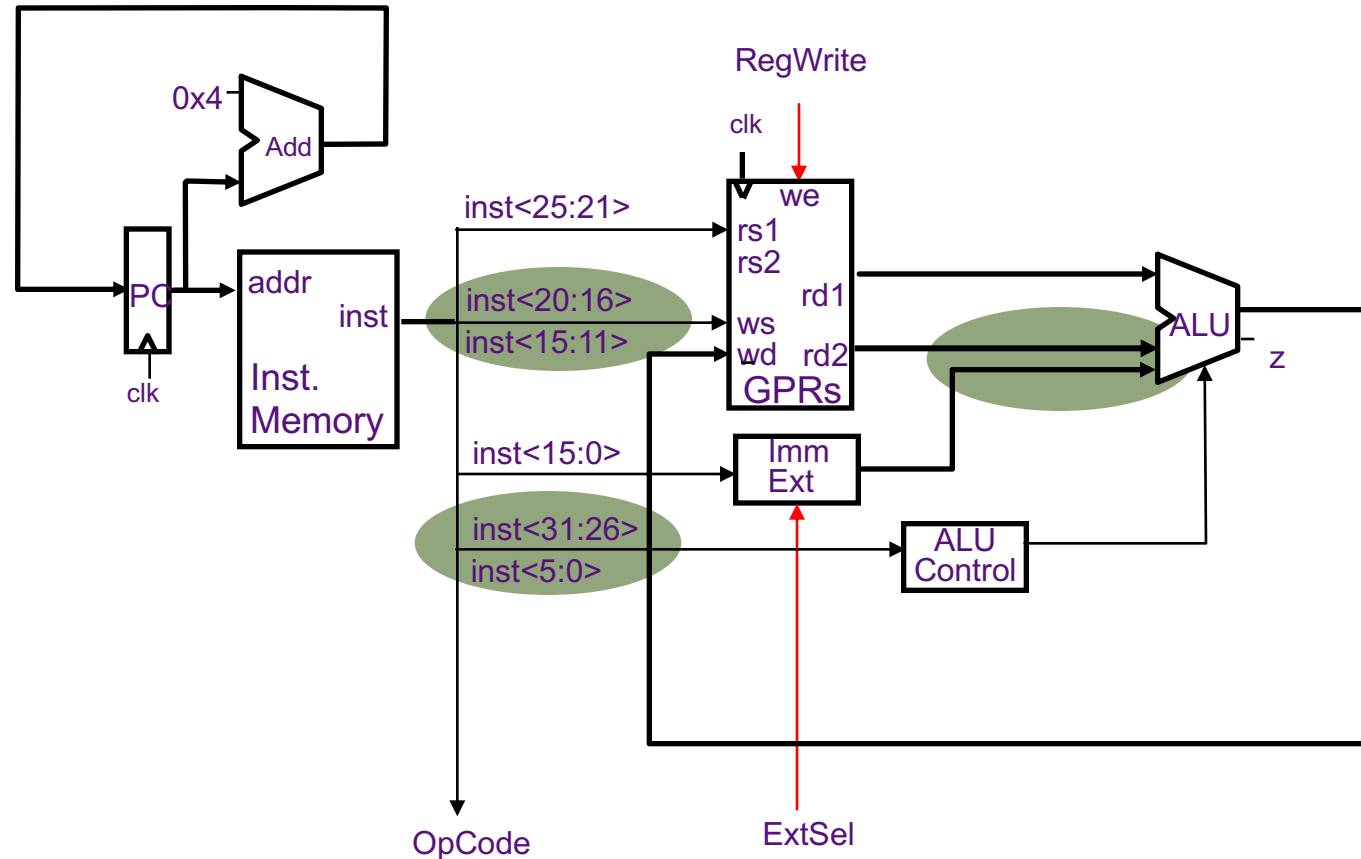
$$rd \leftarrow (rs) \text{ func } (rt)$$

# Datapath: Reg-Imm ALU Instructions

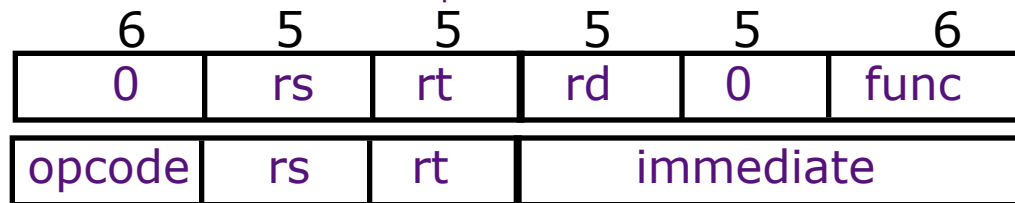


$rt \leftarrow (rs) \text{ op } \text{immediate}$

# Conflicts in Merging Datapath



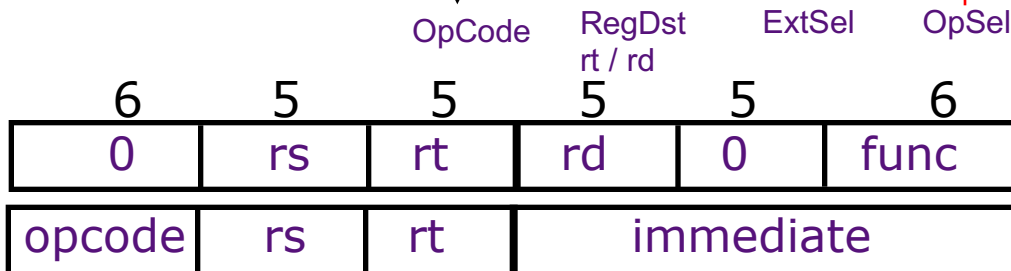
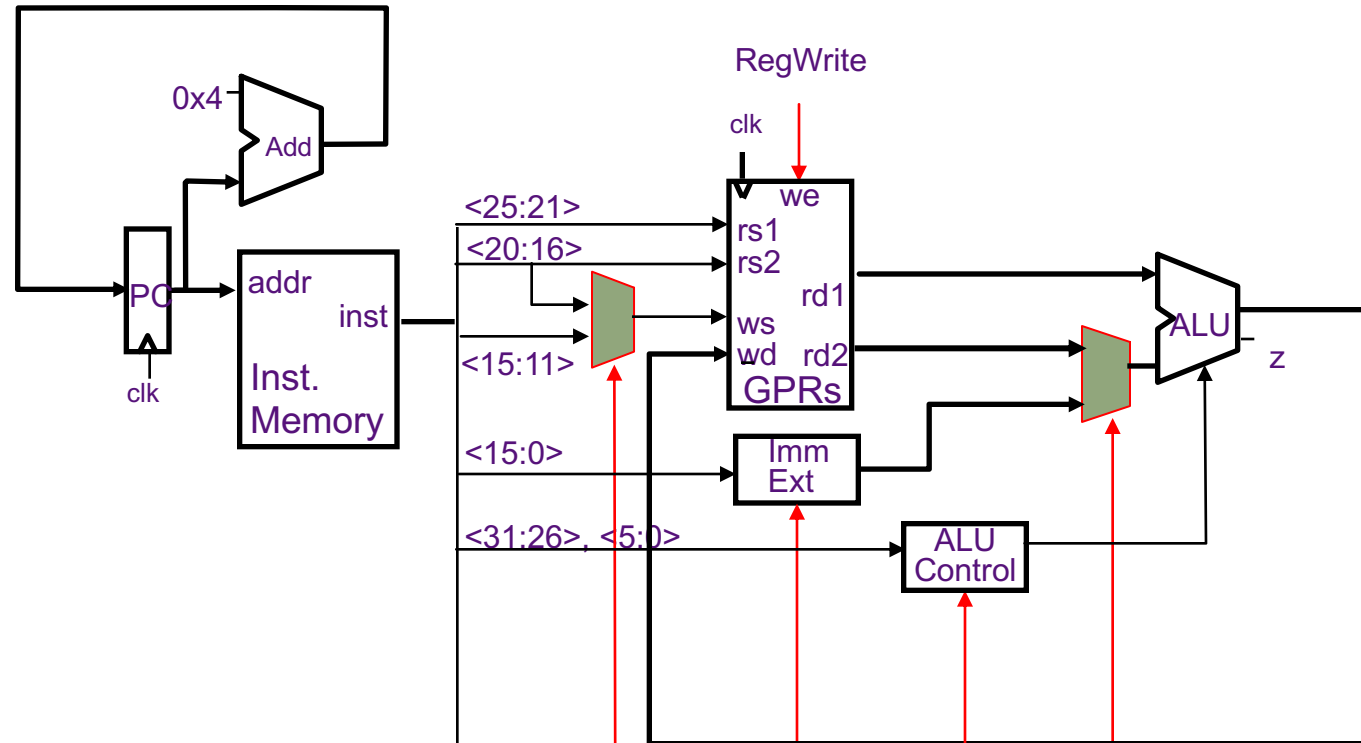
Introduce  
muxes



$rd \leftarrow (rs) \text{ func } (rt)$

$rt \leftarrow (rs) \text{ op } \text{immediate}$

# Datapath for ALU Instructions



$rd \leftarrow (rs) \text{ func } (rt)$

$rt \leftarrow (rs) \text{ op } \text{immediate}$

# Datapath for Memory Instructions

---

Should program and data memory be separate?

*Harvard style: separate* (Aiken and Mark 1 influence)

- read-only program memory
- read/write data memory

- Note:

There must be a way to load the program memory

*Princeton style: the same* (von Neumann's influence)

- single read/write memory for program and data

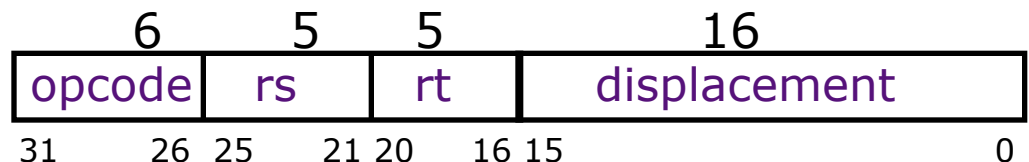
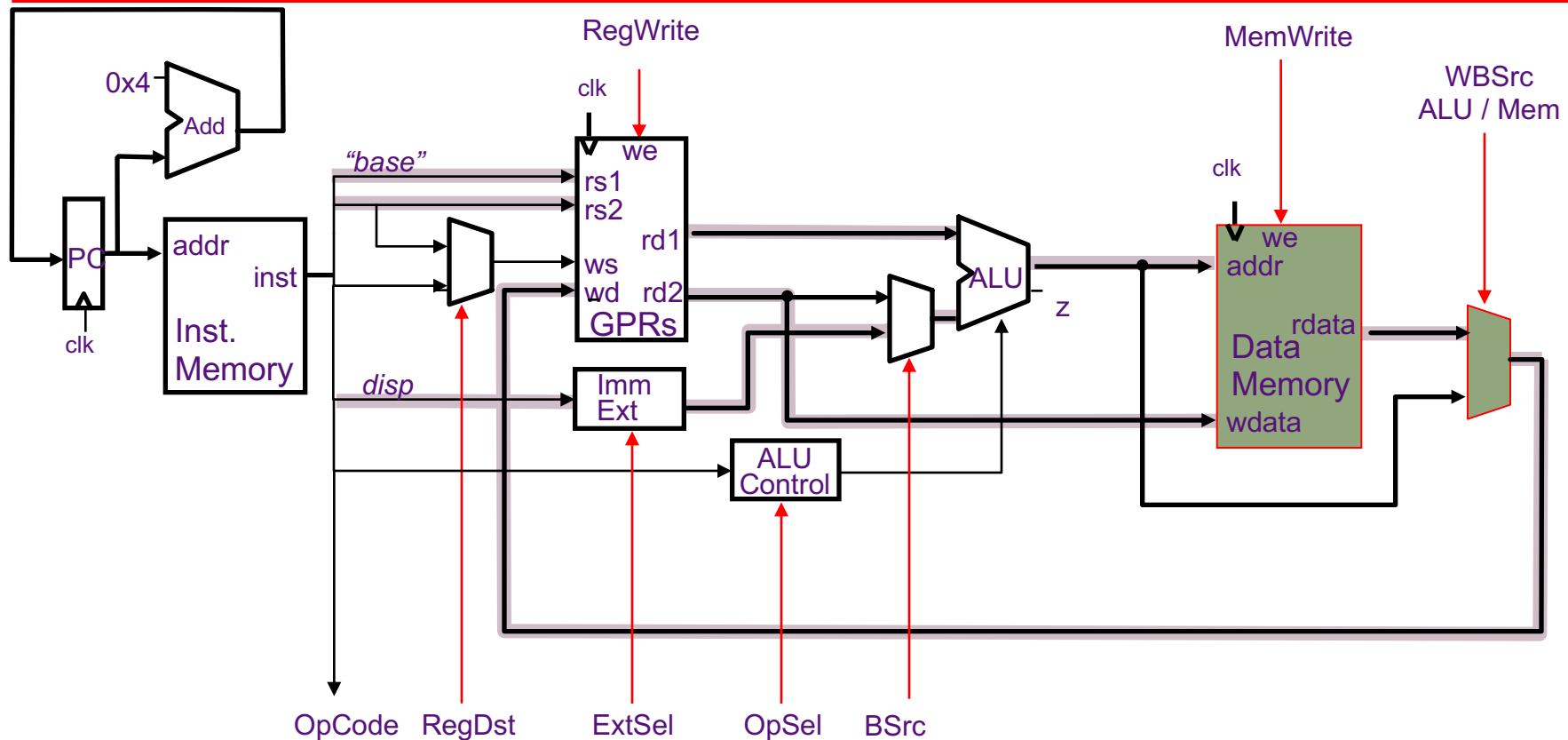
- Note:

Executing a Load or Store instruction requires accessing the memory more than once



# Load/Store Instructions

## Harvard Datapath



addressing mode  
(rs) + displacement

rs is the base register

rt is the destination of a Load or the source for a Store

# MIPS Control Instructions

---

Conditional (on GPR) PC-relative branch



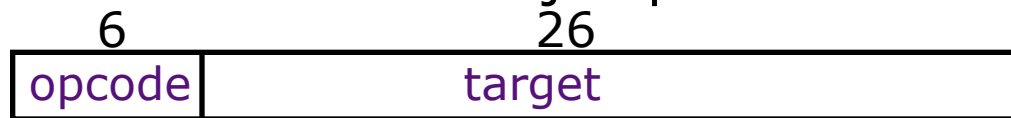
BEQZ, BNEZ

Unconditional register-indirect jumps



JR, JALR

Unconditional absolute jumps



J, JAL

Target PC

Condition

---

BEQZ

BNEZ

---

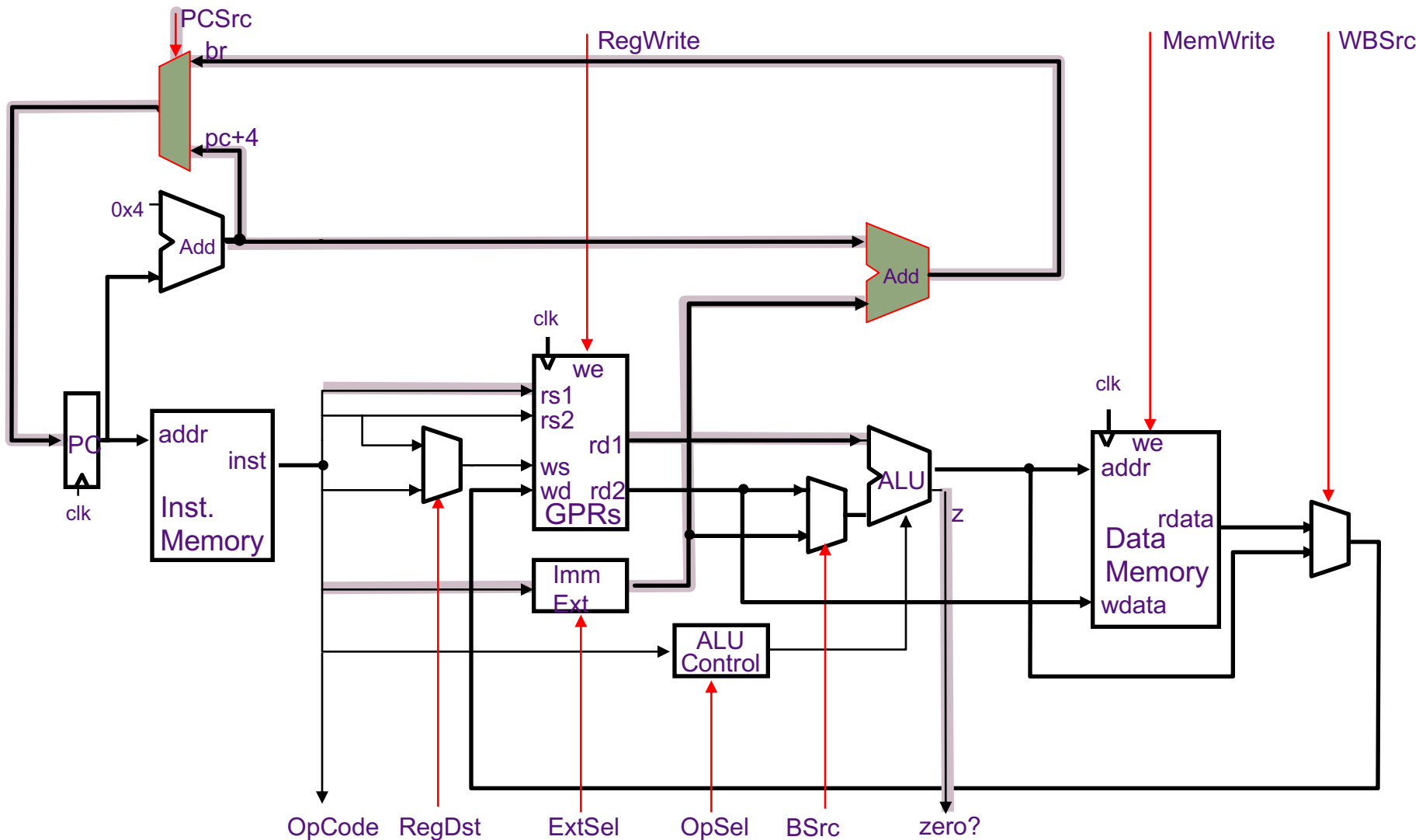
JR, JALR

---

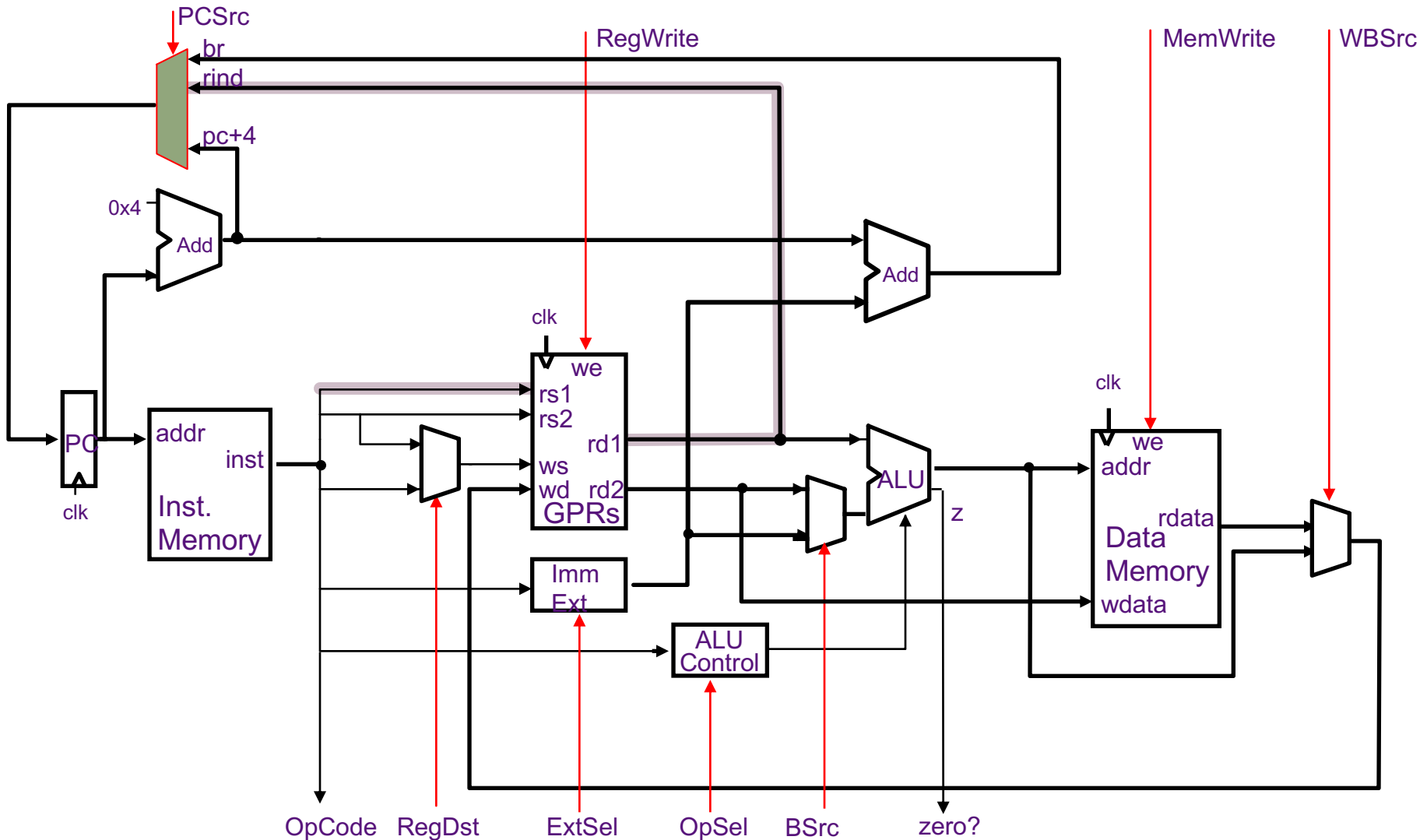
J, JAL

- Jump-&-link stores PC+4 into the link register (R31)
- Control transfers are not delayed  
*we will worry about the branch delay slot later*

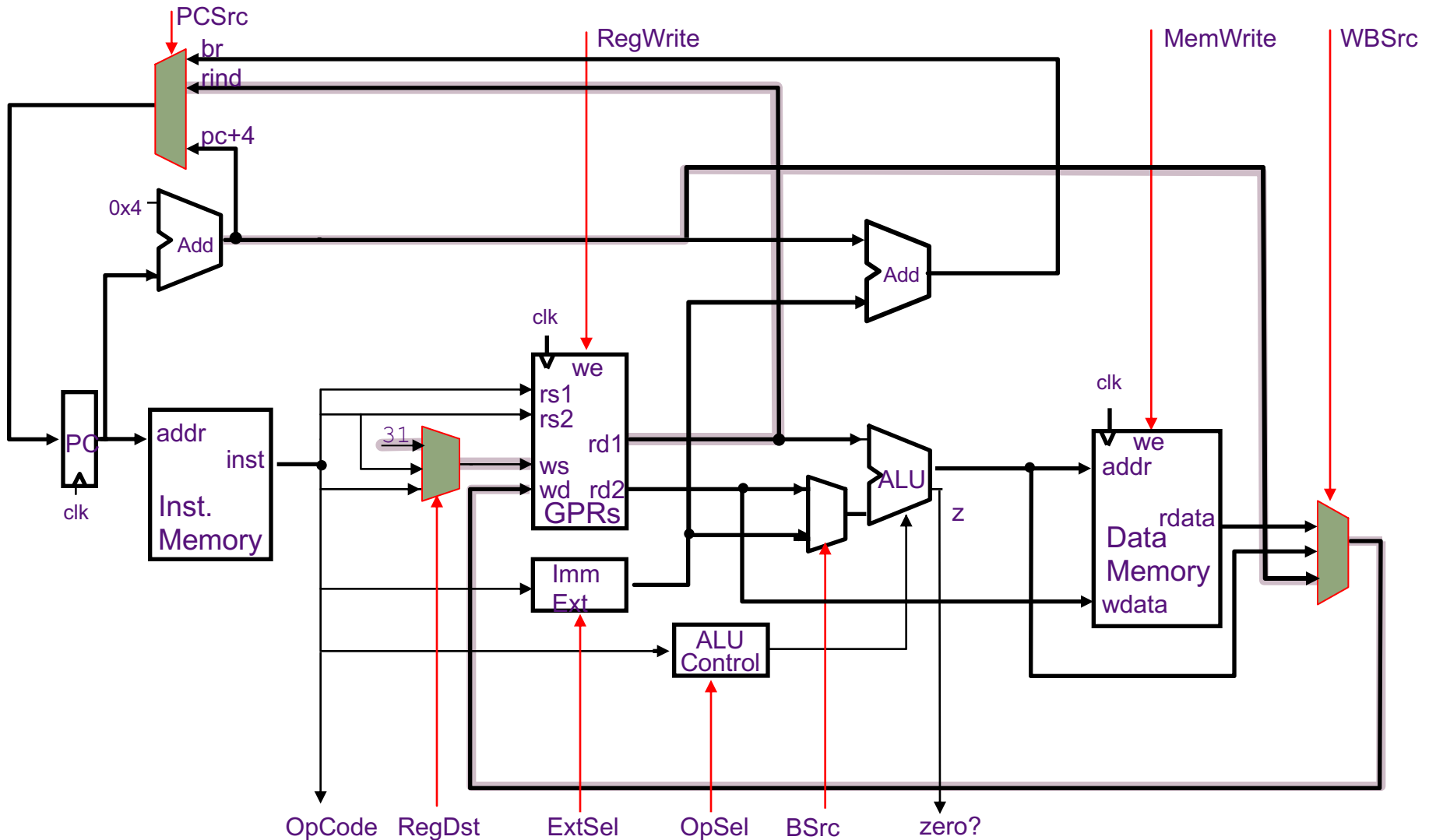
# Conditional Branches (BEQZ, BNEZ)



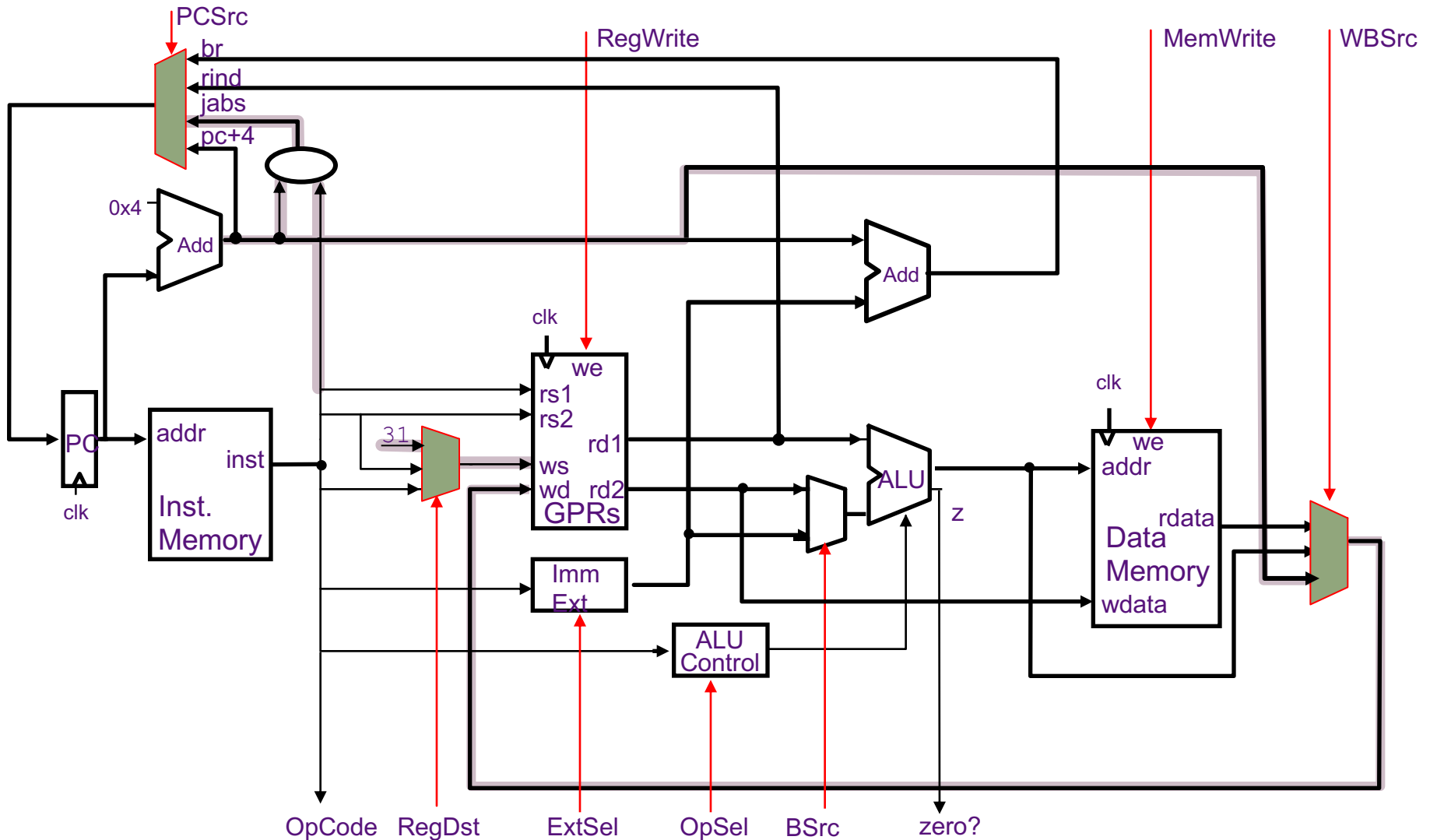
# Register-Indirect Jumps (JR)



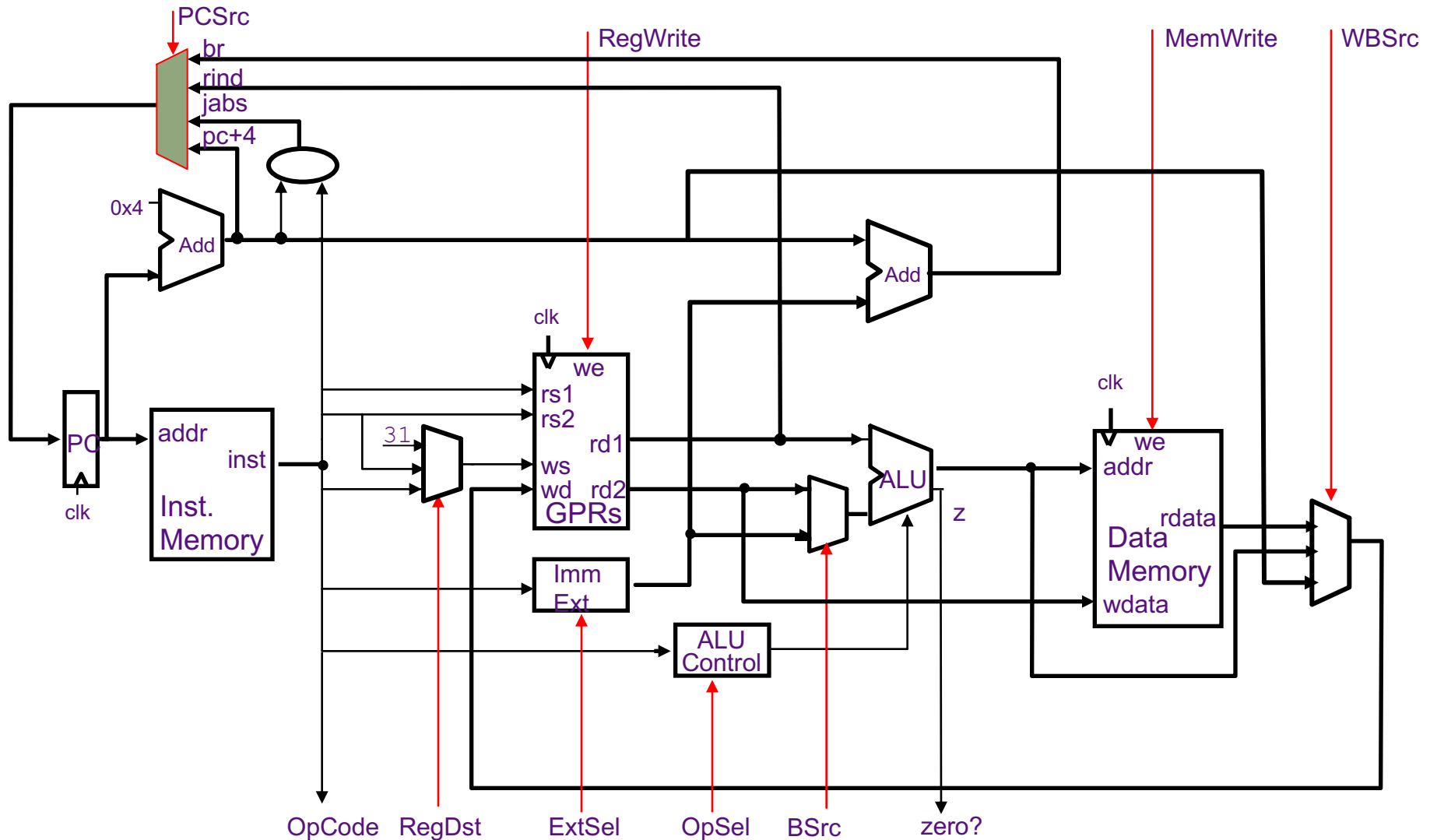
# Register-Indirect Jump-&-Link (JALR)



# Absolute Jumps (J, JAL)

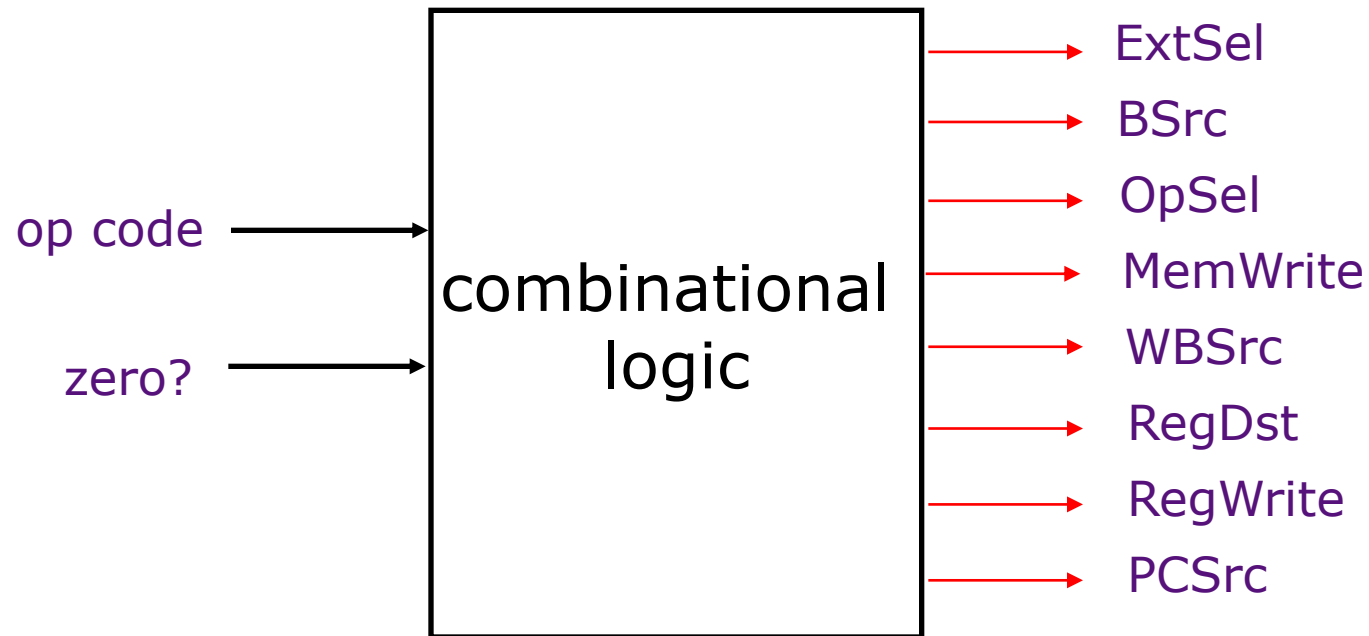


# Harvard-Style Datapath for MIPS



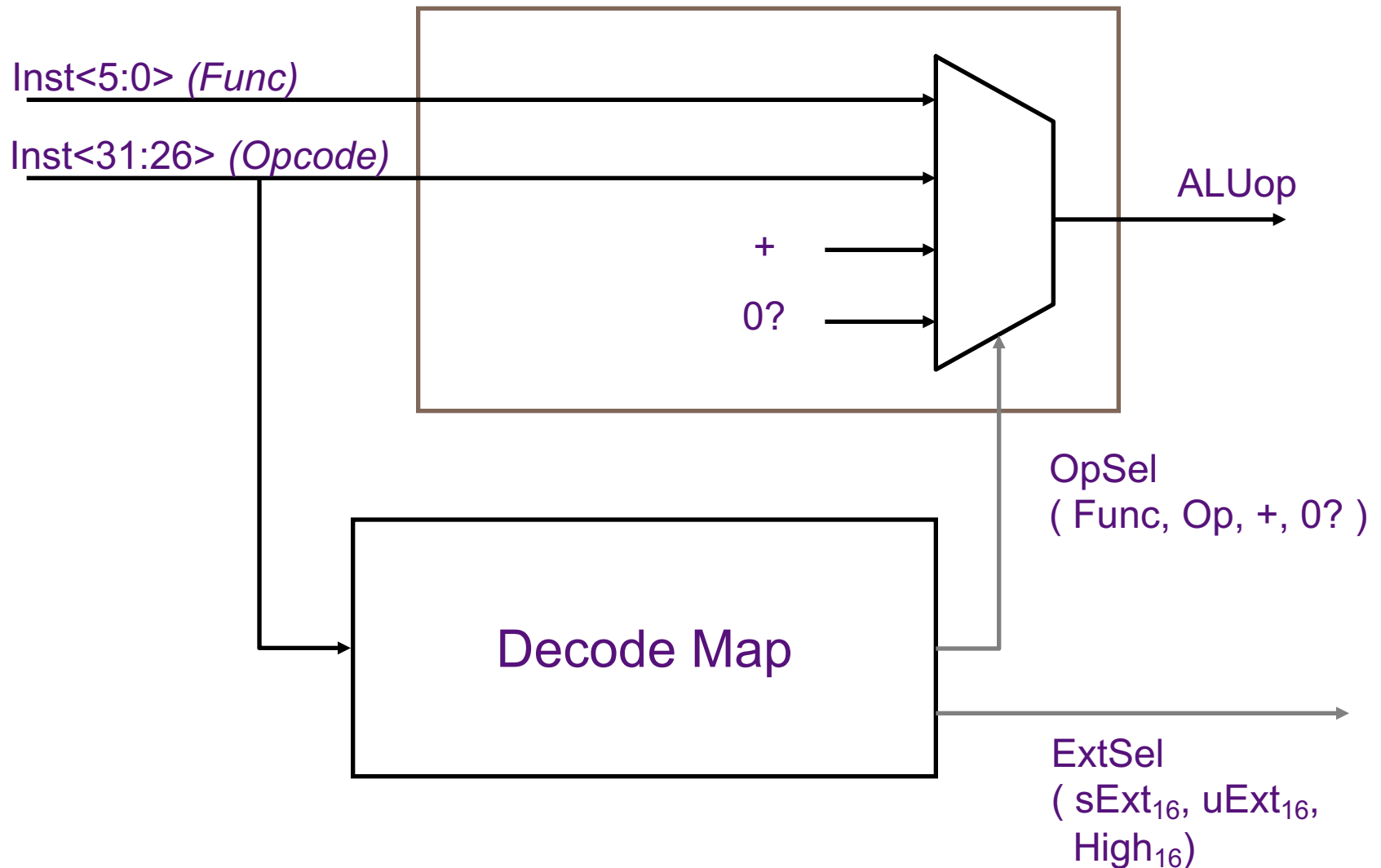
# Hardwired Control is pure Combinational Logic

---





# ALU Control & Immediate Extension



# Hardwired Control Table

Opcode	ExtSel	BSrc	OpSel	MemW	RegW	WBSrc	RegDst	PCSrc
ALU								
ALUi								
ALUiu								
LW								
SW								
BEQ <sub>Z=0</sub>								
BEQ <sub>Z=1</sub>								
J								
JAL								
JR								
JALR								

BSrc = Reg / Imm  
 RegDst = rt / rd / R31

WBSrc = ALU / Mem / PC  
 PCSrc = pc+4 / br / rind / jabs

# Single-Cycle Hardwired Control:

## *Harvard architecture*

---

We will assume

- Clock period is sufficiently long for all of the following steps to be “completed”:

1. instruction fetch
2. decode and register fetch
3. ALU operation
4. data fetch if required
5. register write-back setup time

$$\Rightarrow t_C > t_{IFetch} + t_{RFetch} + t_{ALU} + t_{DMem} + t_{RWB}$$

- At the rising edge of the following clock, the PC, the register file and the memory are updated

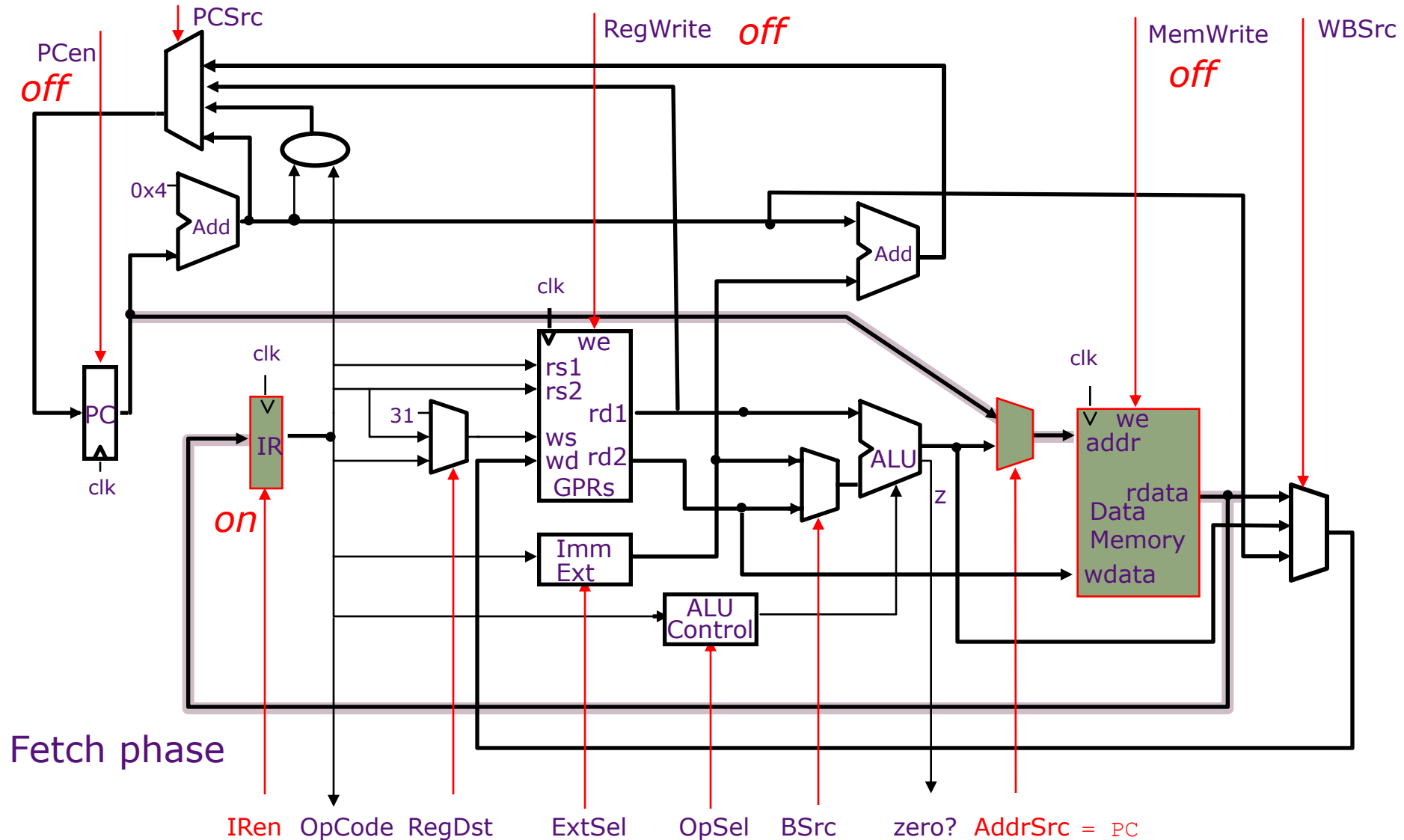
# Princeton challenge

---

- What problem arises if instructions and data reside in the same memory?

# Princeton Microarchitecture

## Datapath & Control

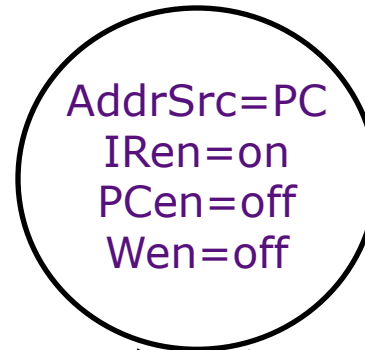


# Two-State Controller:

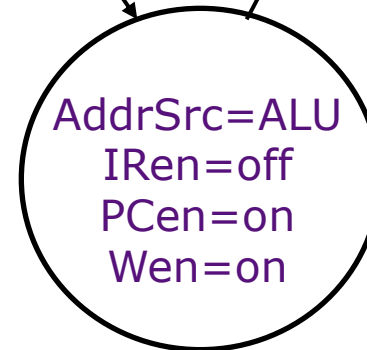
## *Princeton Architecture*

---

*fetch phase*



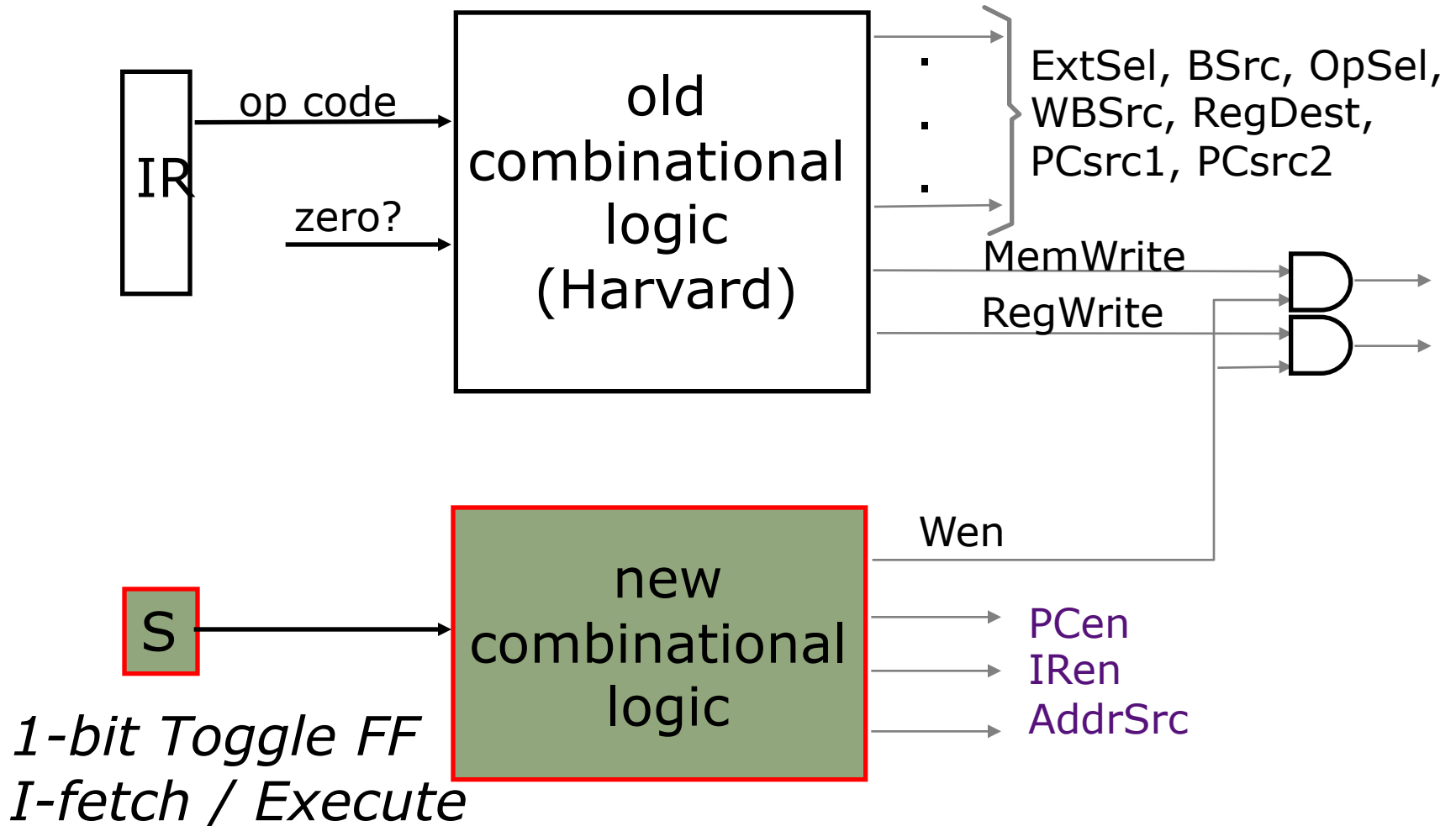
*execute phase*



A flipflop can be used to remember the phase

# Hardwired Controller:

## *Princeton Architecture*



# Clock Rate vs CPI

---

$$t_{\text{C-Princeton}} > \max \{t_M, t_{\text{RF}} + t_{\text{ALU}} + t_M + t_{\text{WB}}\}$$

$$t_{\text{C-Princeton}} > t_{\text{RF}} + t_{\text{ALU}} + t_M + t_{\text{WB}}$$

$$t_{\text{C-Harvard}} > t_M + t_{\text{RF}} + t_{\text{ALU}} + t_M + t_{\text{WB}}$$

Suppose  $t_M \gg t_{\text{RF}} + t_{\text{ALU}} + t_{\text{WB}}$

$$t_{\text{C-Princeton}} = 0.5 * t_{\text{C-Harvard}}$$

$$\text{CPI}_{\text{Princeton}} = 2$$

$$\text{CPI}_{\text{Harvard}} = 1$$

*No difference in performance!*

Is it possible to design a controller for the Princeton architecture with  $\text{CPI} < 2$  ?

*CPI = Clock cycles Per Instruction*

*Stay tuned!*