Hardwired, Non-pipelined ISA Implementation

Hyun Ryong (Ryan) Lee
Computer Science & Artificial Intelligence Lab
M.I.T.

Administrivia

- TAs: **Ryan Lee** and Nikola Samardzic
- Contact: 6823-tas@lists.csail.mit.edu
  or hrlee@csail.mit.edu, nsamar@csail.mit.edu
- Please use Piazza extensively for questions!
- Office Hours: Wed. 4-5:30pm, 32-G7 common area 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...

Last Lecture...

- Computer Architecture as designing under constraints
  - Spans much of the computing stack nowadays

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

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!

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

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

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

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 →

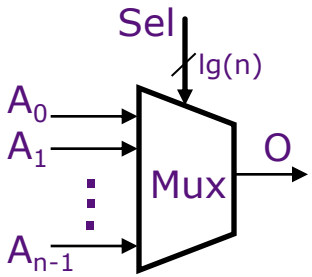
Microarchitecture	CPI	cycle time
Microcoded	>1	short
Single-cycle unpipelined	1	long
Pipelined	1	short

Hardware Elements

- Combinational circuits
  - Mux, Demux, Decoder, ALU, ...

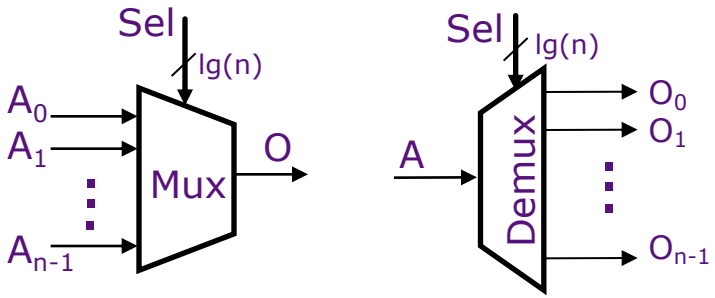
Hardware Elements

- Combinational circuits
  - Mux, Demux, Decoder, ALU, ...



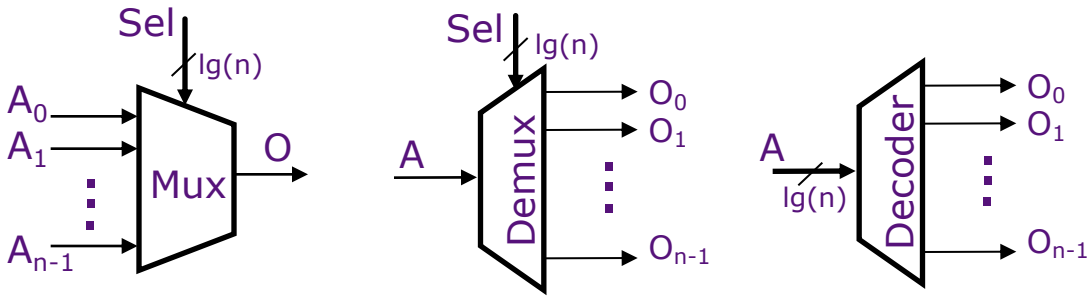
Hardware Elements

- Combinational circuits
  - Mux, Demux, Decoder, ALU, ...



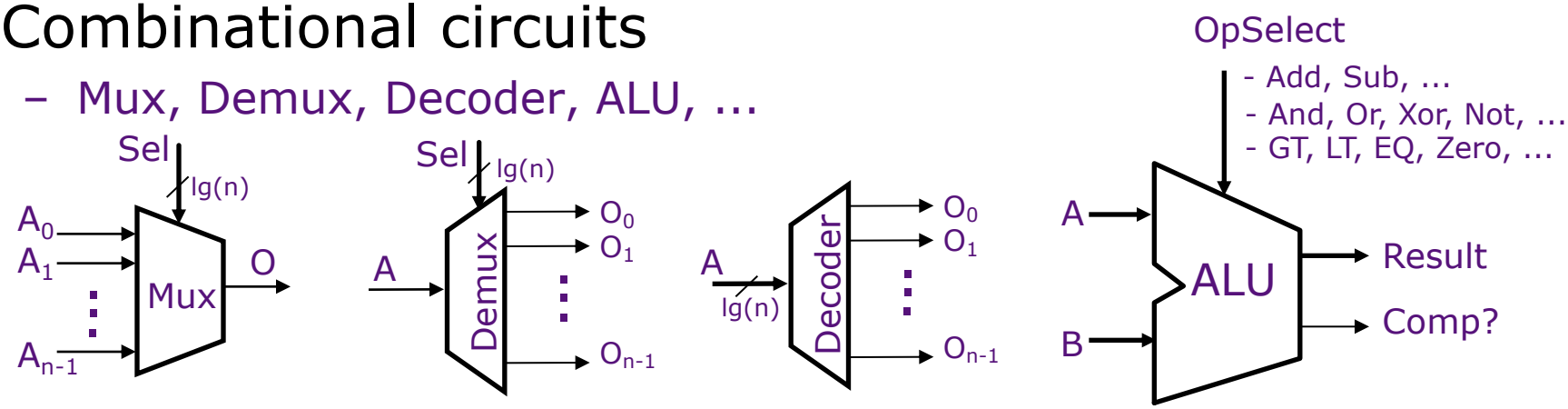
Hardware Elements

- Combinational circuits
  - Mux, Demux, Decoder, ALU, ...



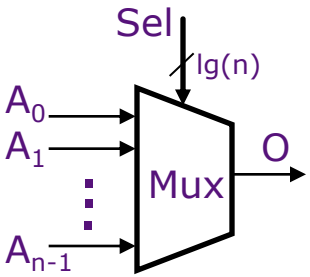
Hardware Elements

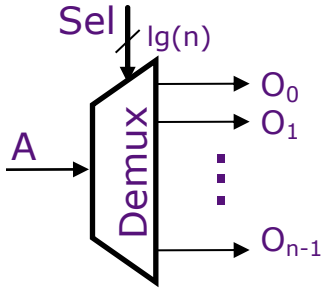
- Combinational circuits
  - Mux, Demux, Decoder, ALU, ...

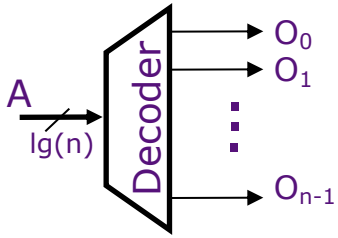


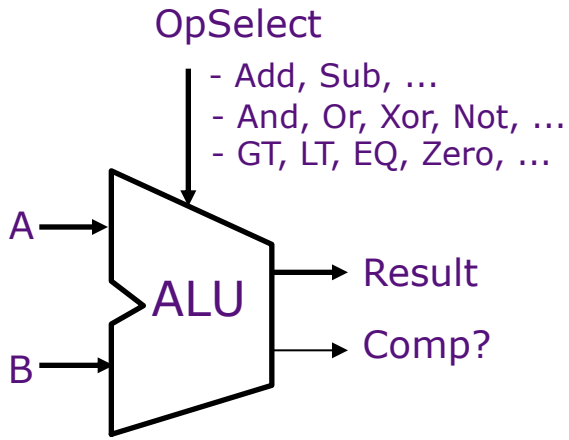
Hardware Elements

- Combinational circuits
  - Mux, Demux, Decoder, ALU, ...





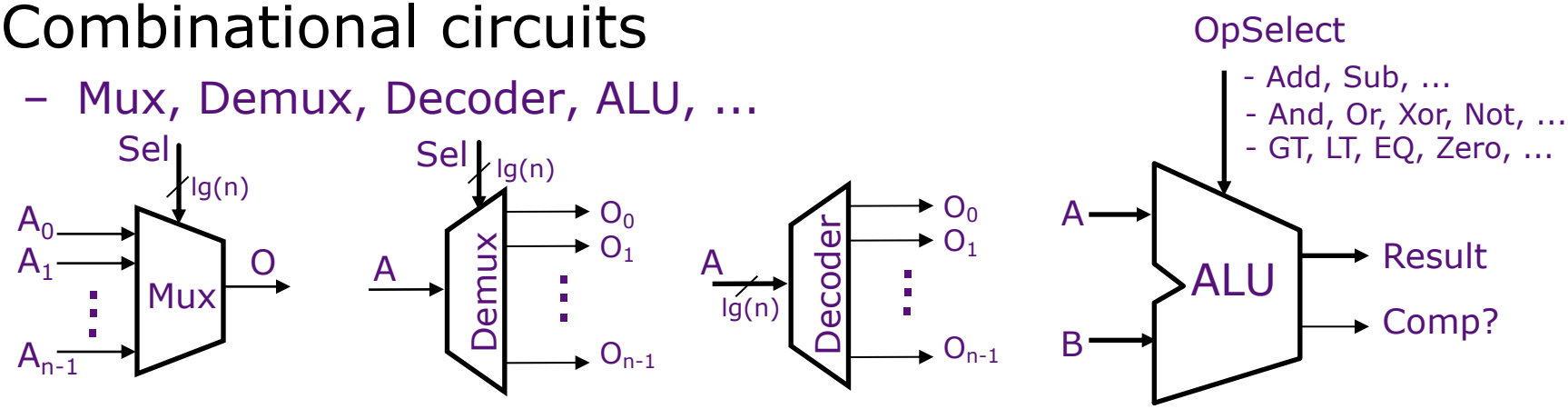




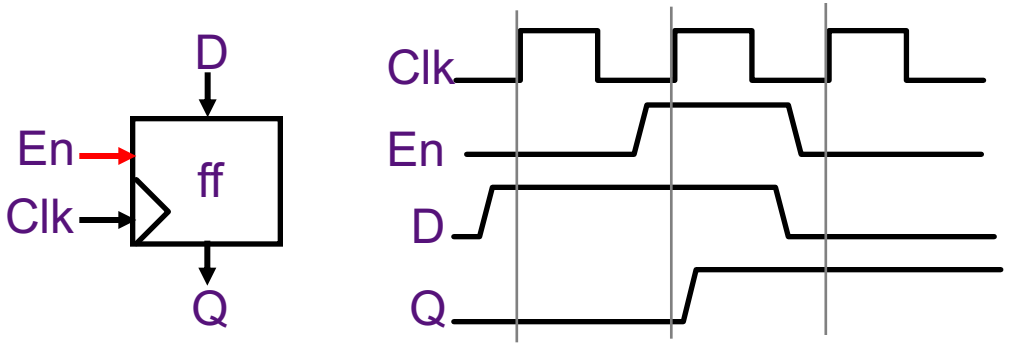
- Synchronous state elements
  - Flipflop, Register, Register file, SRAM, DRAM

Hardware Elements

- Combinational circuits
  - Mux, Demux, Decoder, ALU, ...



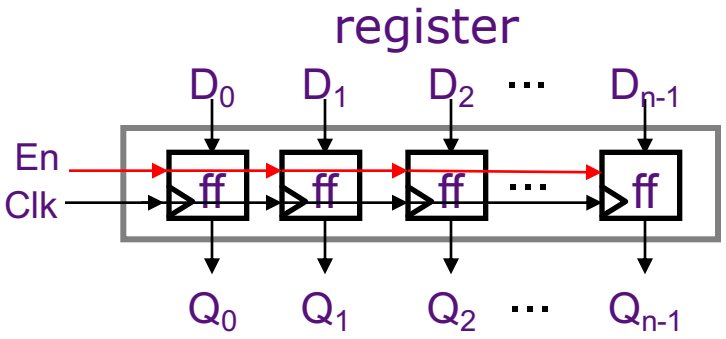
- Synchronous state elements
  - Flipflop, Register, Register file, SRAM, DRAM



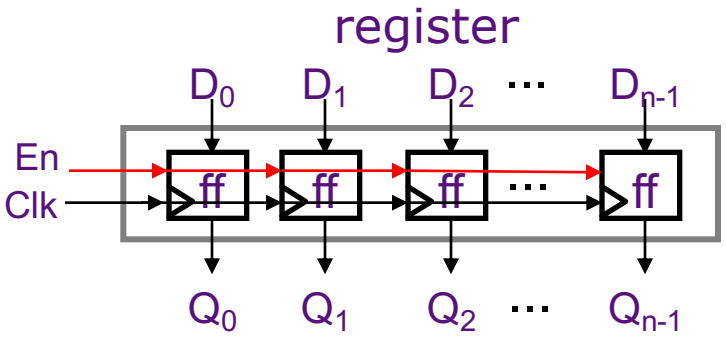
Edge-triggered: Data is sampled at the rising edge

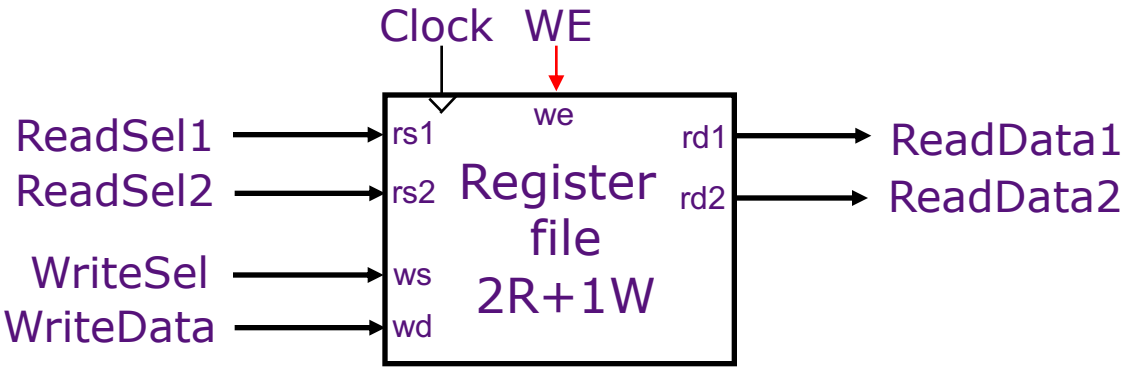
Register Files

Register Files

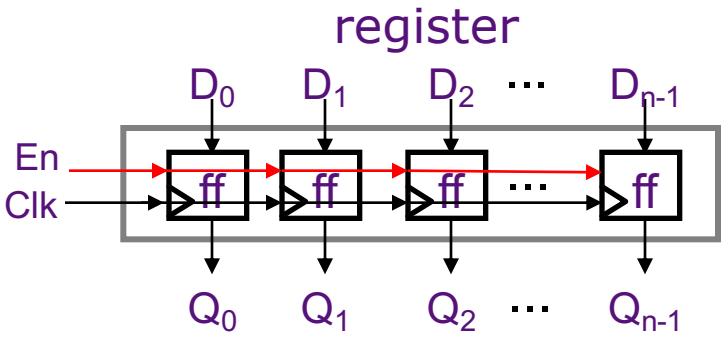


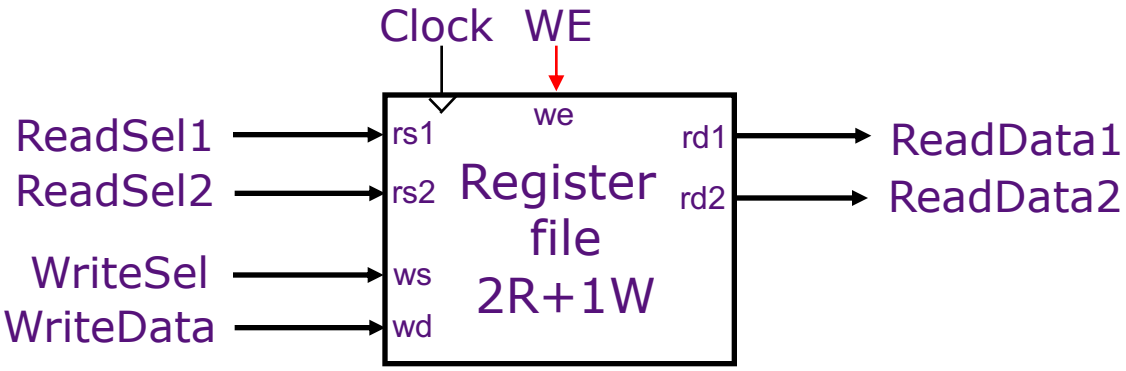
Register Files





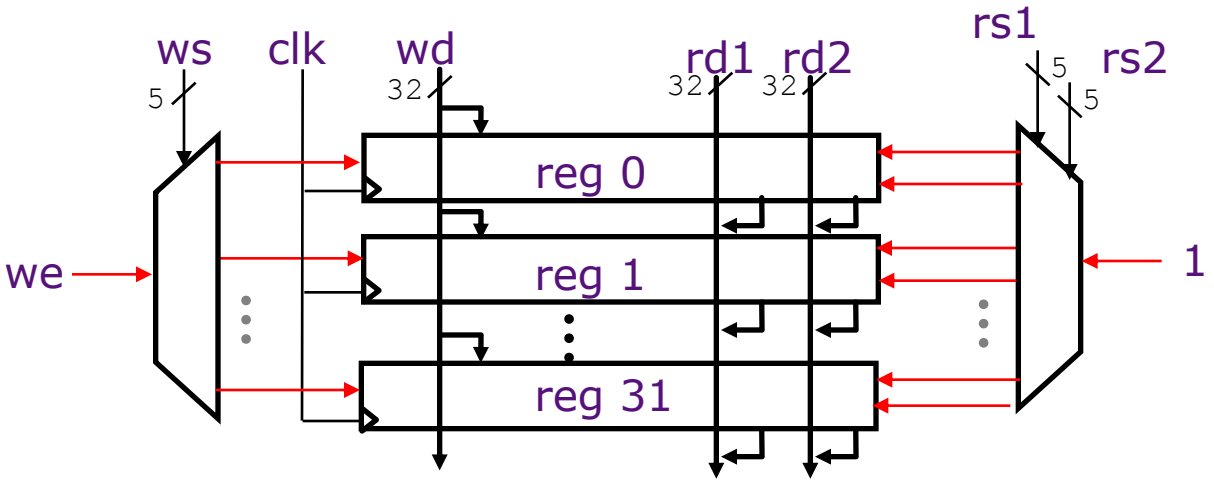
Register Files





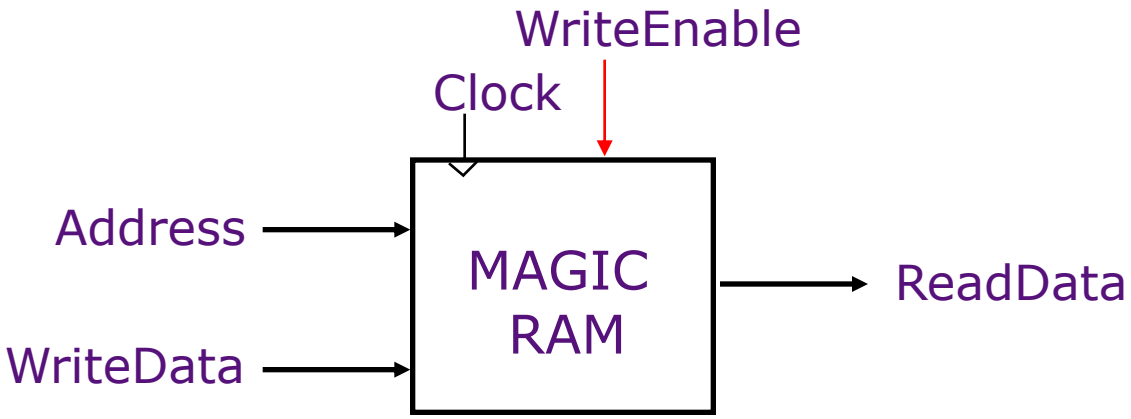
No timing issues when reading and writing the same register (writes happen at the end of the cycle)

Register File Implementation



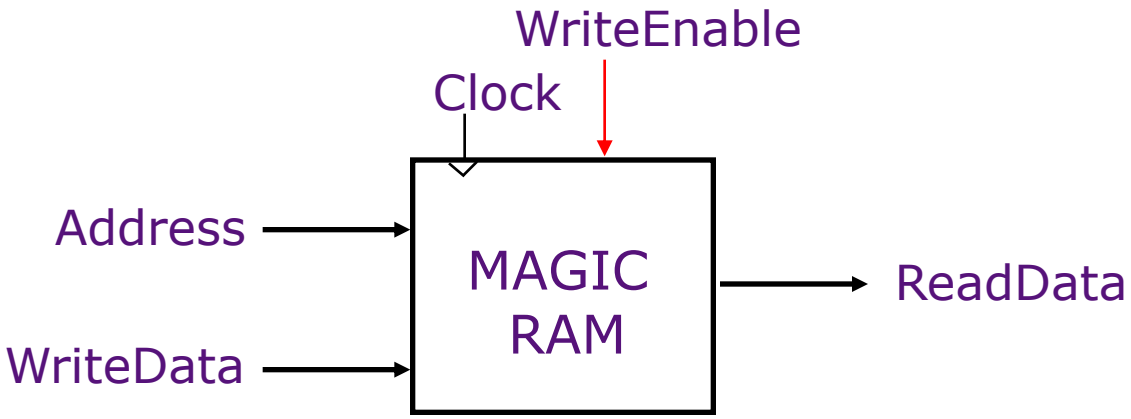
- Register files with a large number of ports are difficult to design
  - Area scales with ports^2
  - *Intel's Itanium GPR File has 128 registers with 8 read ports and 4 write ports!!*

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)*

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 RISC-V:

Single-cycle per instruction datapath & control logic

The RISC-V 32-bit ISA

Processor State

- 32 32-bit GPRs, x0 always contains a 0
- PC, the program counter
- Single- and double-precision floating point extensions
- Some other special registers

Data types

- 8-bit byte, 16-bit half word, 32-bit word
- Integers are signed and in 2's complement

Load/Store style instruction set

- Data addressing modes: immediate + register
- Branch addressing modes: PC-relative for branches, absolute or register-indirect for jumps
- Byte-addressable memory, little-endian mode

All instructions are 32 bits

Instruction Execution

Execution of an instruction involves

Instruction Execution

Execution of an instruction involves

1. Instruction fetch

Instruction Execution

Execution of an instruction involves

1. Instruction fetch
2. Decode

Instruction Execution

Execution of an instruction involves

1. Instruction fetch
2. Decode
3. Register fetch

Instruction Execution

Execution of an instruction involves

1. Instruction fetch
2. Decode
3. Register fetch
4. ALU operation

Instruction Execution

Execution of an instruction involves

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

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

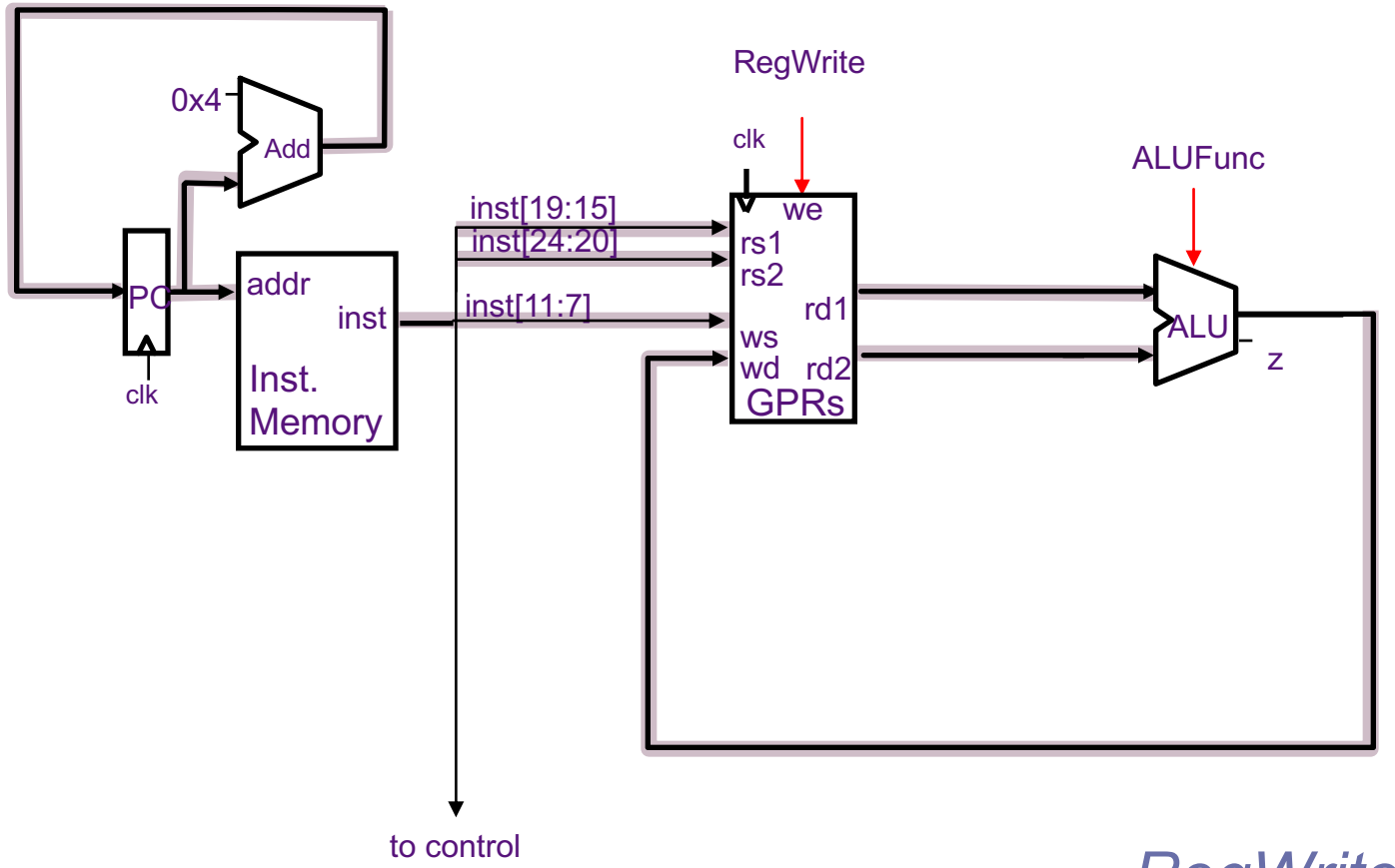
Instruction Execution

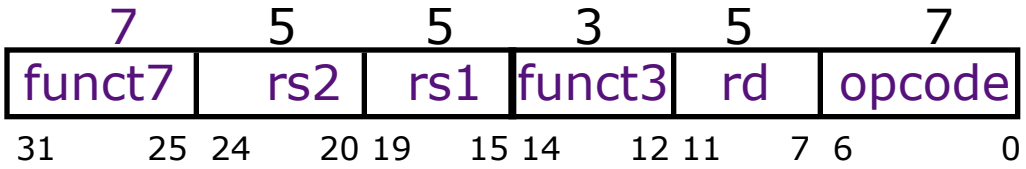
Execution of an instruction involves

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

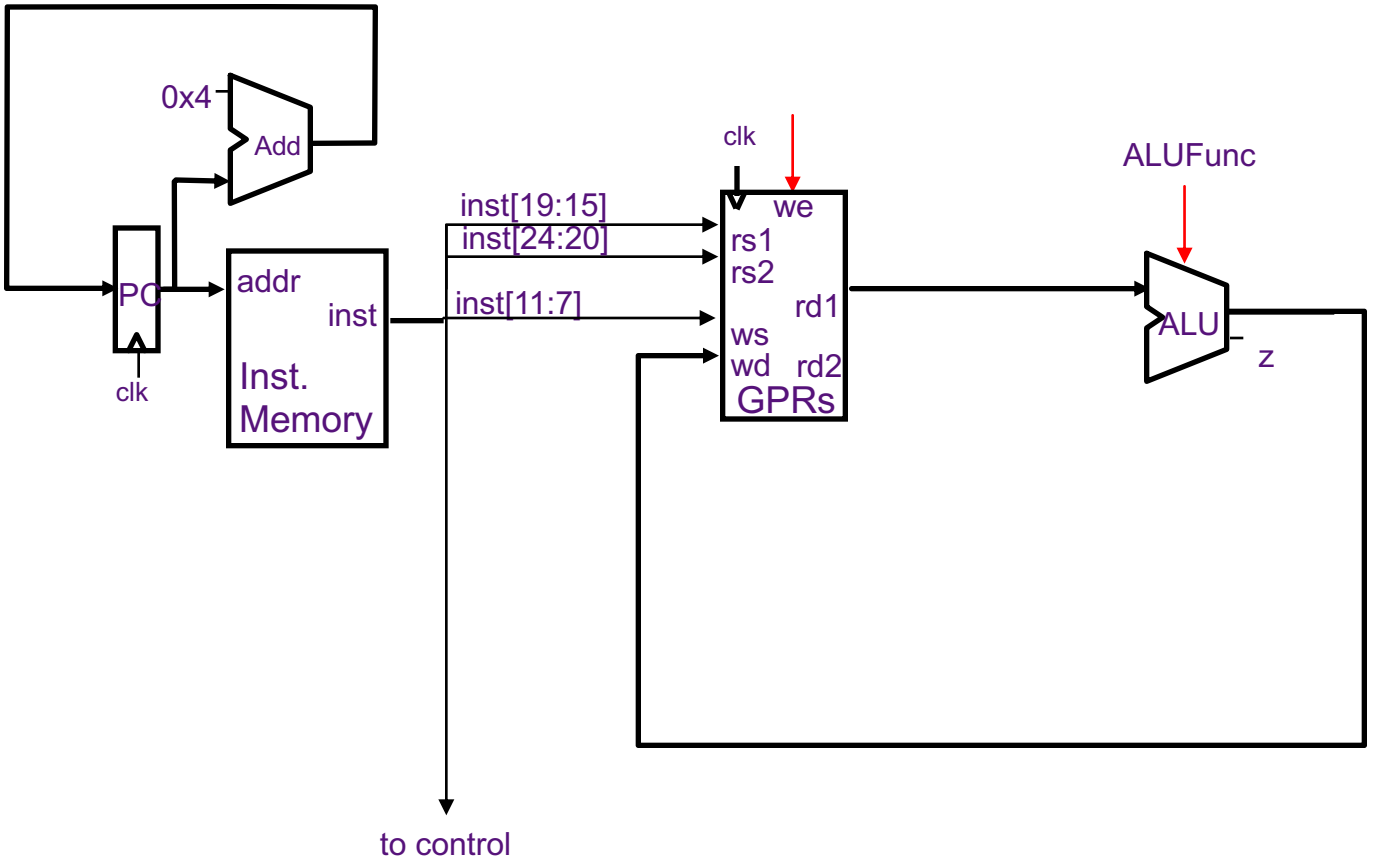


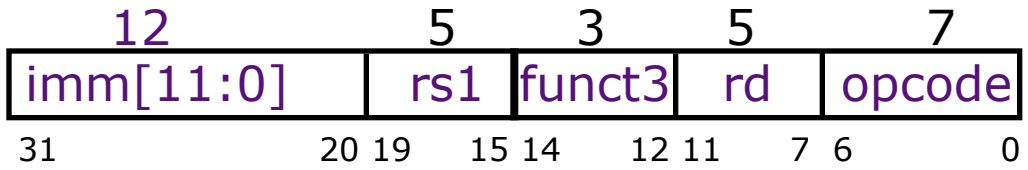


RegWrite Timing?

rd ← (rs1) funct (rs2)

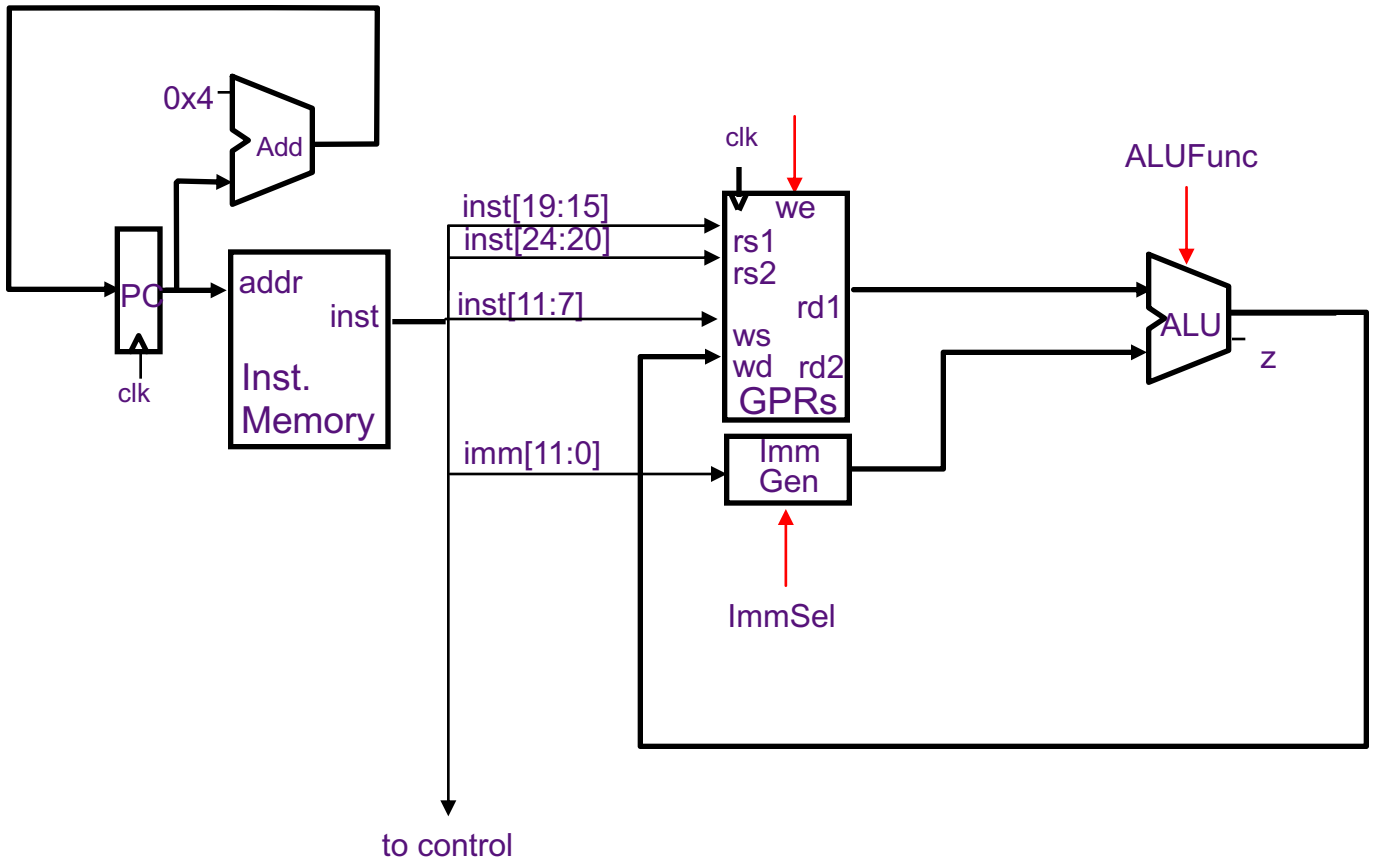
Datapath: Reg-Imm ALU Instructions

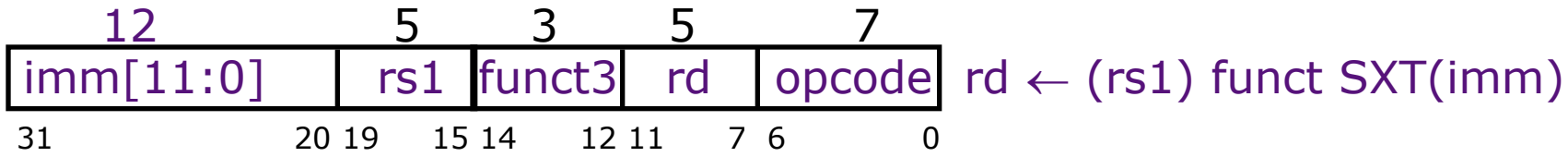




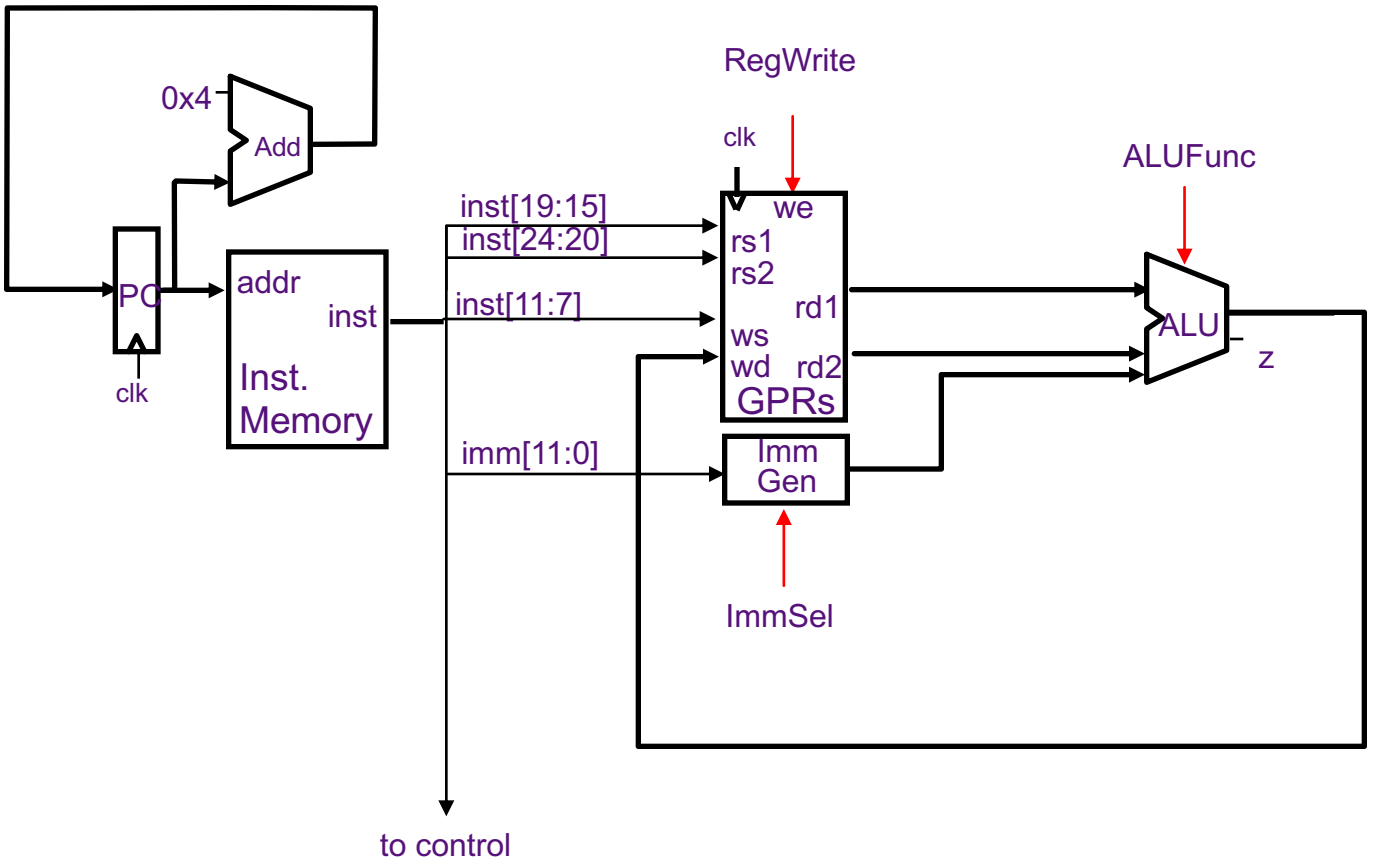
rd ← (rs1) funct SXT(imm)

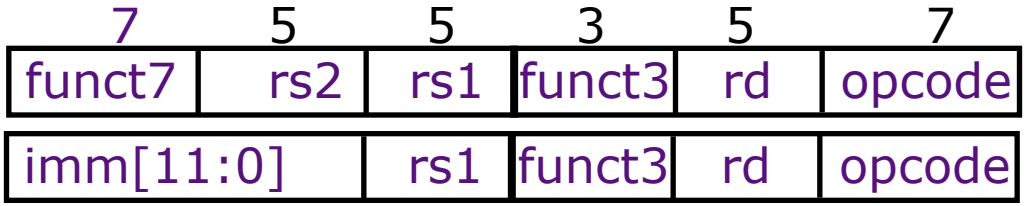
Datapath: Reg-Imm ALU Instructions





Conflicts in Merging Datapath

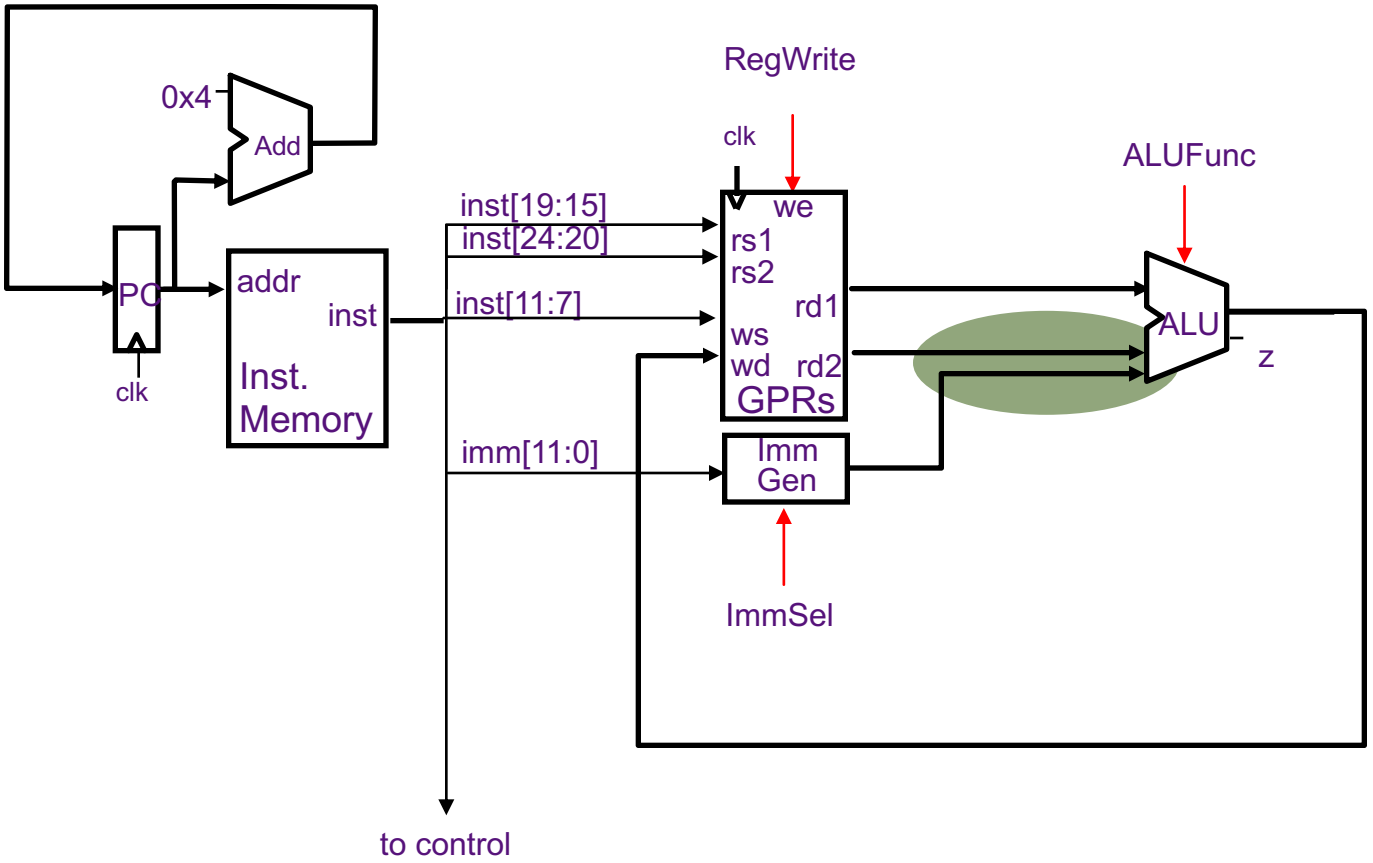


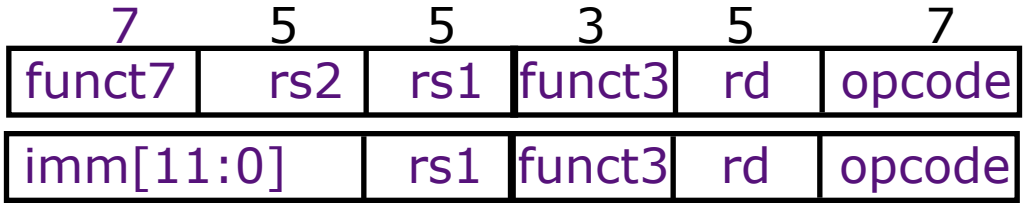


rd ← (rs1) funct (rs2)

rd ← (rs1) funct SXT(imm)

Conflicts in Merging Datapath

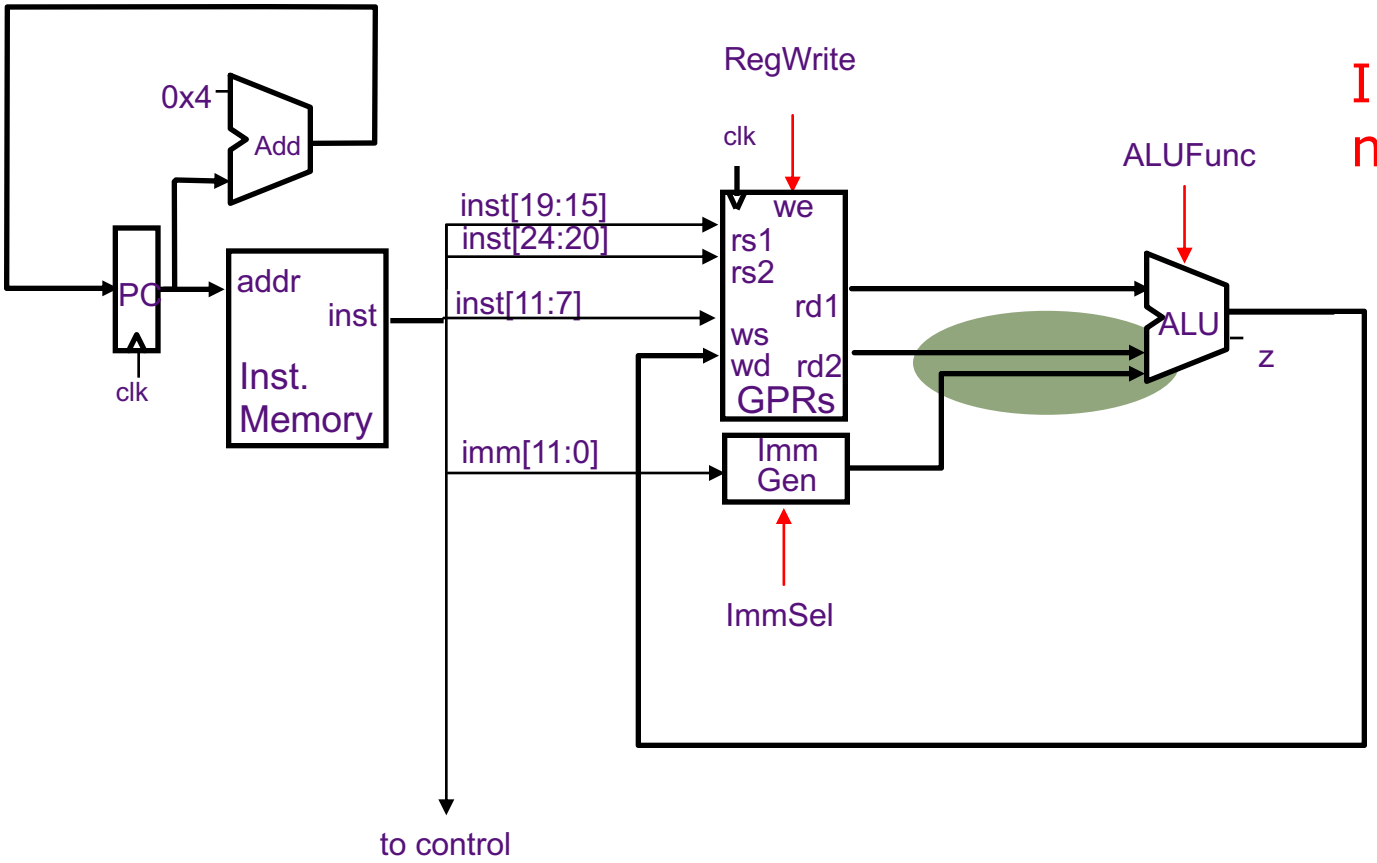


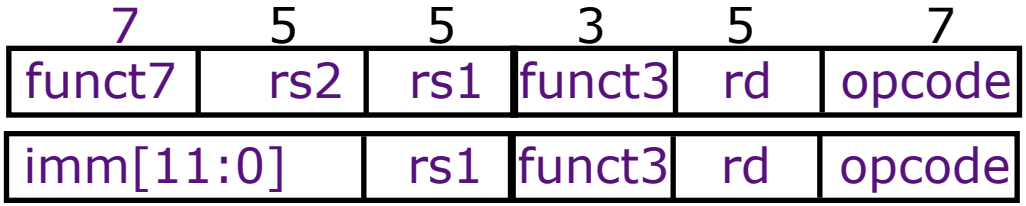


rd ← (rs1) funct (rs2)

rd ← (rs1) funct SXT(imm)

Conflicts in Merging Datapath

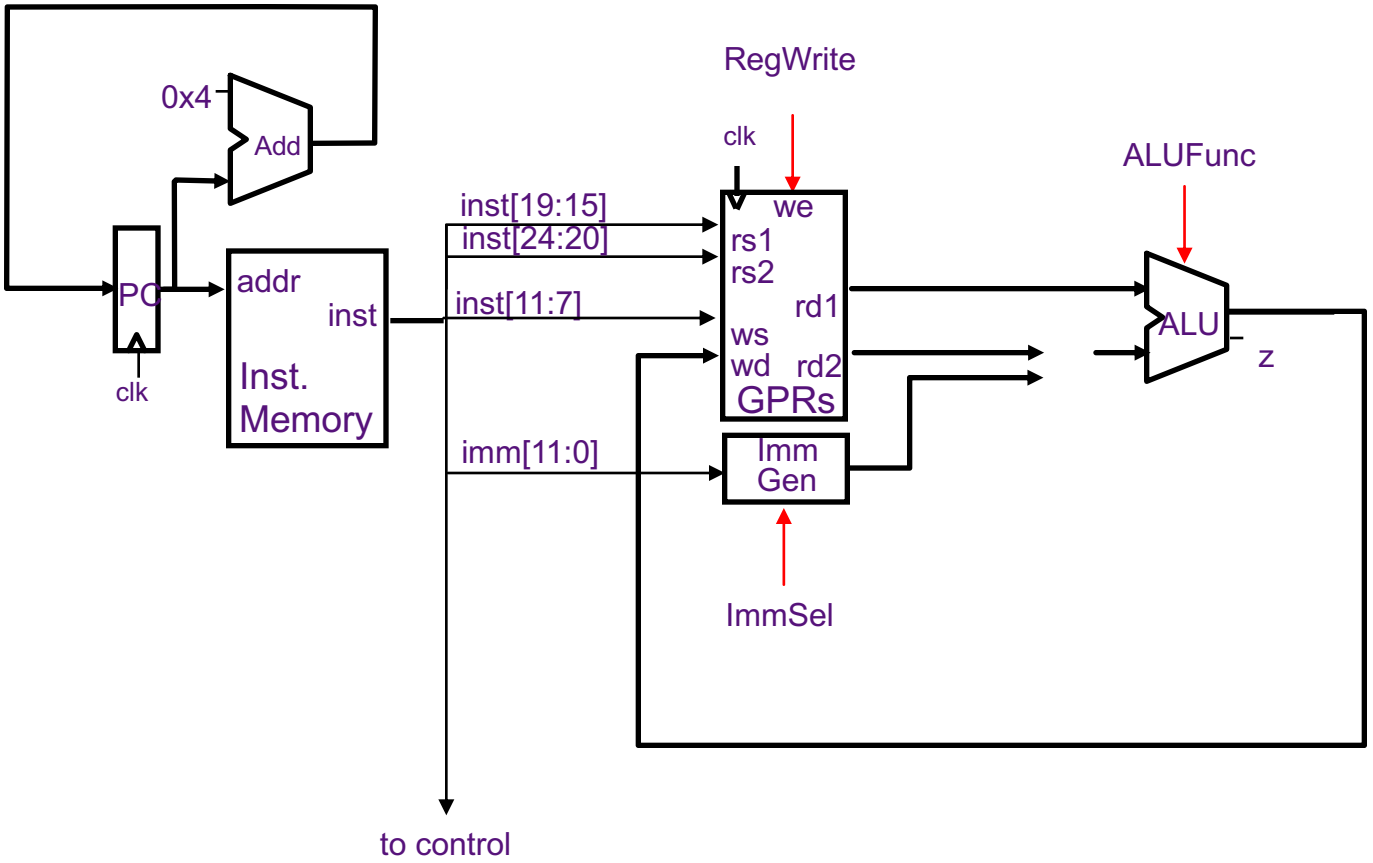


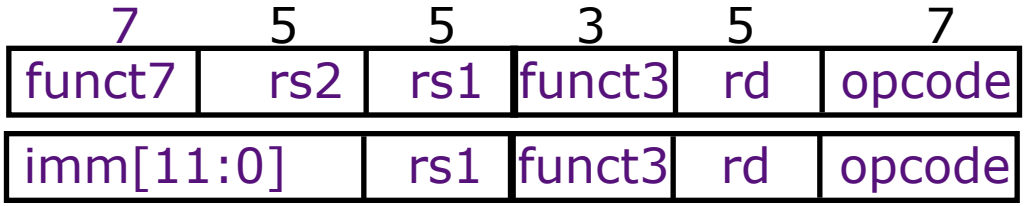


rd ← (rs1) funct (rs2)

rd ← (rs1) funct SXT(imm)

Datapath for ALU Instructions

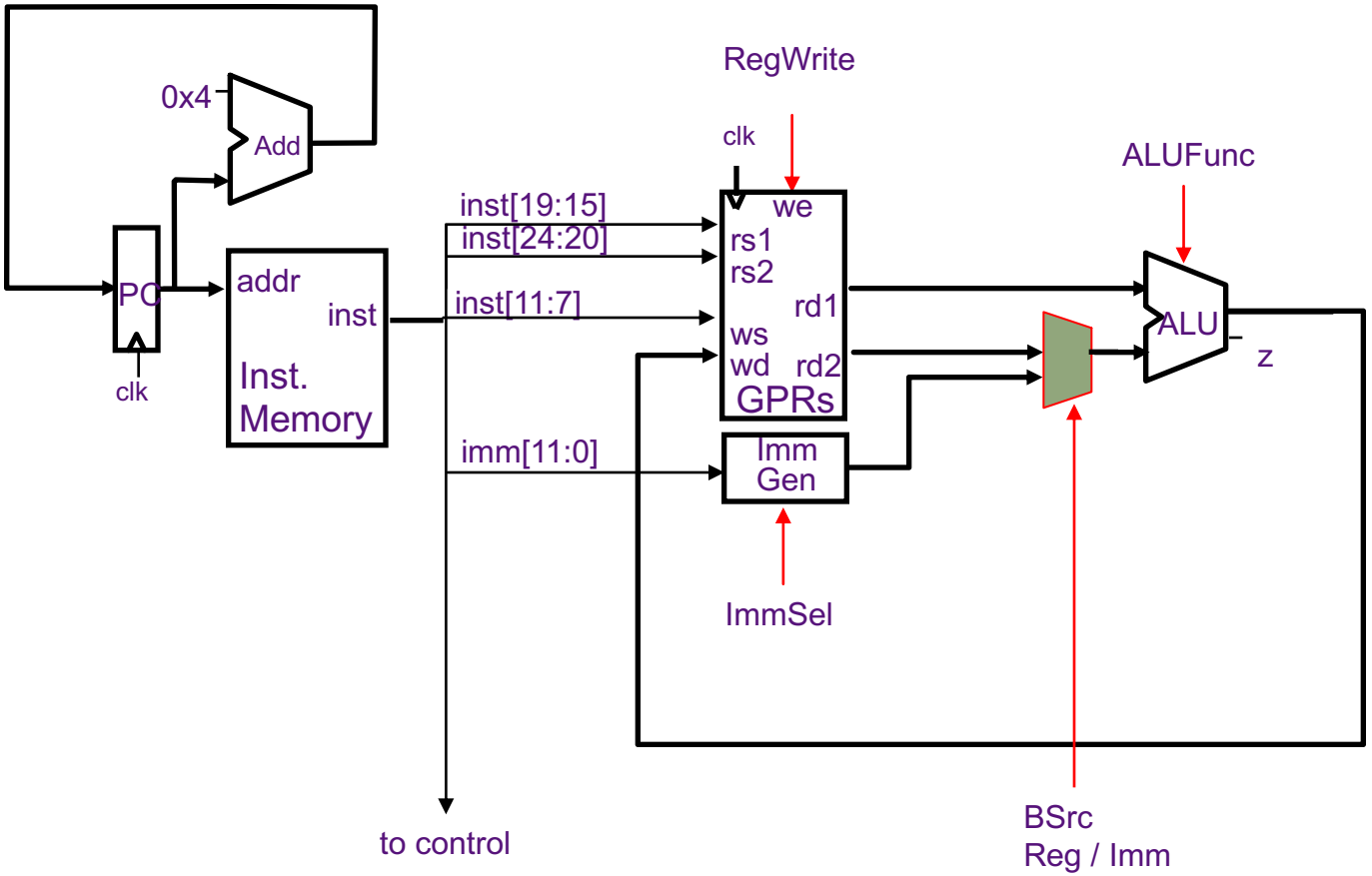




rd ← (rs1) funct (rs2)

rd ← (rs1) funct SXT(imm)

Datapath for ALU Instructions



7	5	5	3	5	7
funct7	rs2	rs1	funct3	rd	opcode
imm[11:0]		rs1	funct3	rd	opcode

rd ← (rs1) funct (rs2)

rd ← (rs1) funct SXT(imm)

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

Princeton style: the same (von Neumann's influence)
- single read/write memory for program and data

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

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

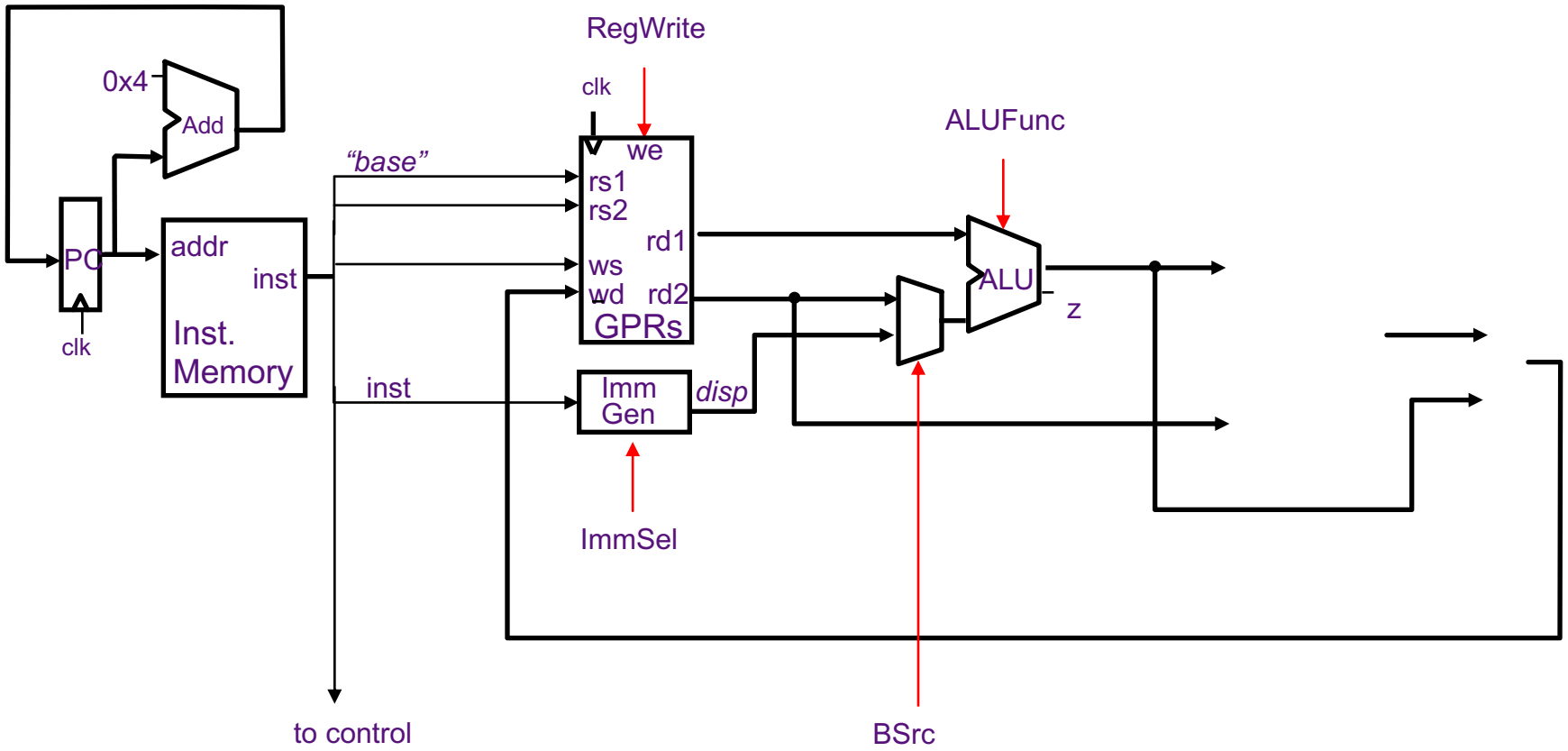
- Note:
  There must be a way to load the program memory

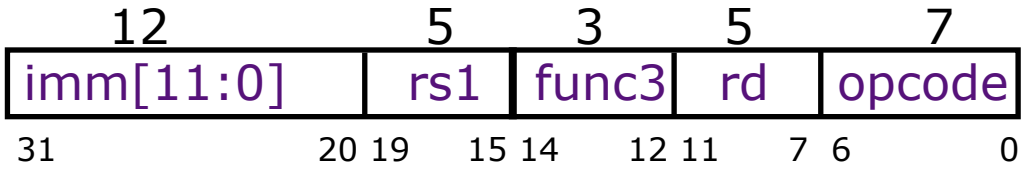
Princeton style: the same (von Neumann's influence)
- single read/write memory for program and data

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

Load Instructions

Harvard Datapath



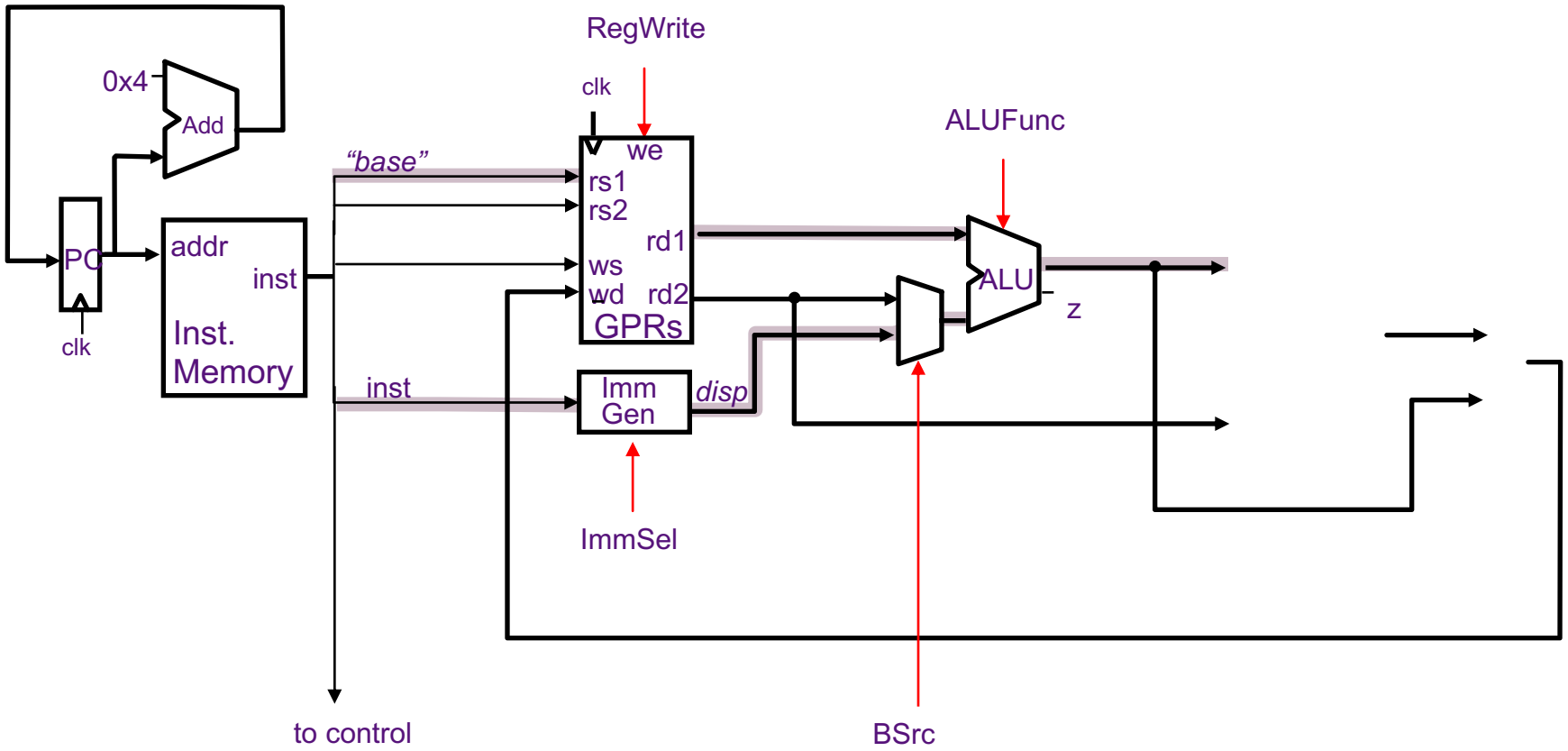


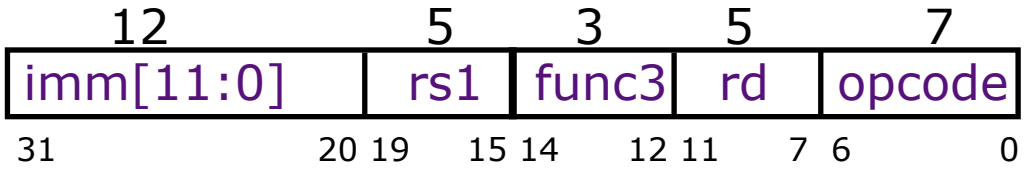
addressing mode
rs1 + imm. displacement

Same encoding as immediate instructions
rs1 contains base address, rd is the destination.

Load Instructions

Harvard Datapath



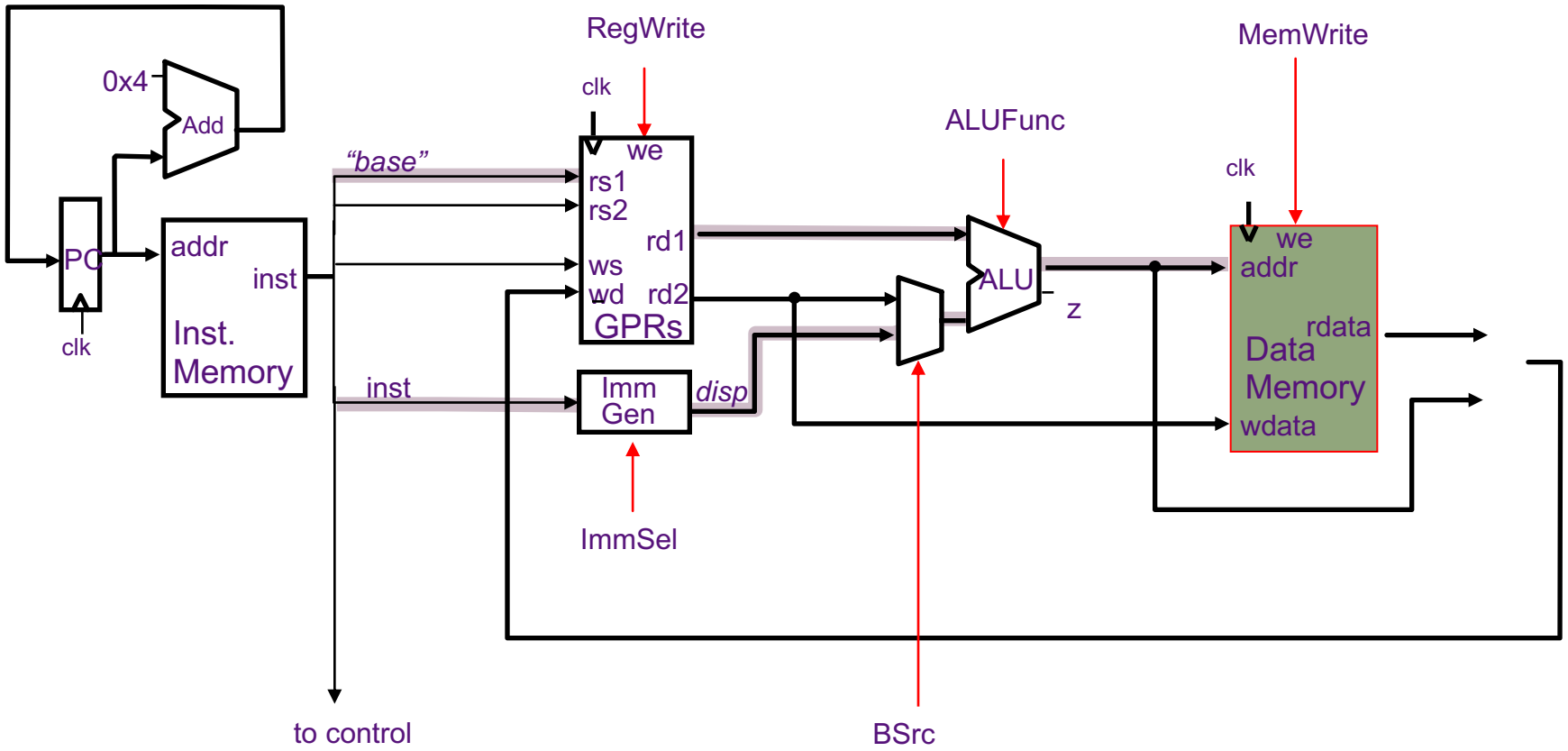


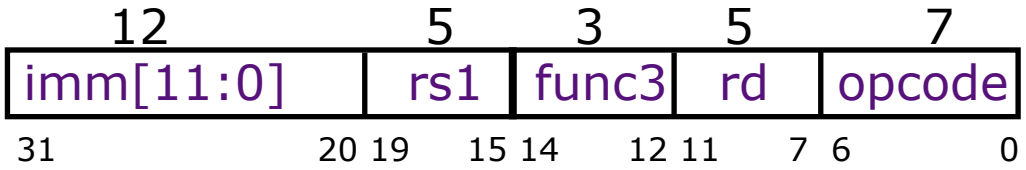
addressing mode
rs1 + imm. displacement

Same encoding as immediate instructions
rs1 contains base address, rd is the destination.

Load Instructions

Harvard Datapath



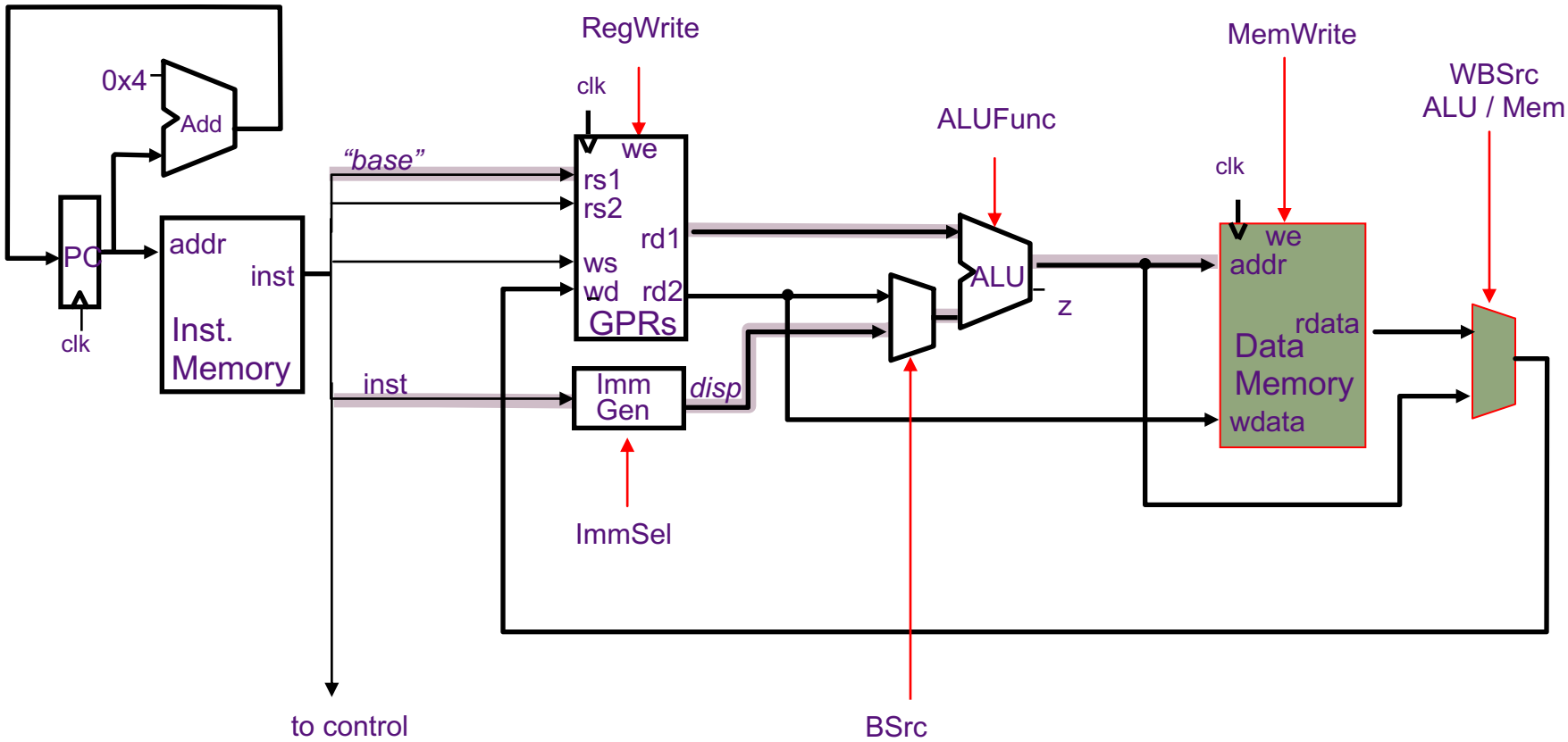


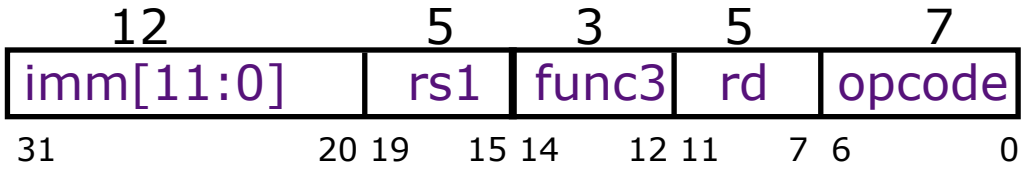
addressing mode
rs1 + imm. displacement

Same encoding as immediate instructions
rs1 contains base address, rd is the destination.

Load Instructions

Harvard Datapath



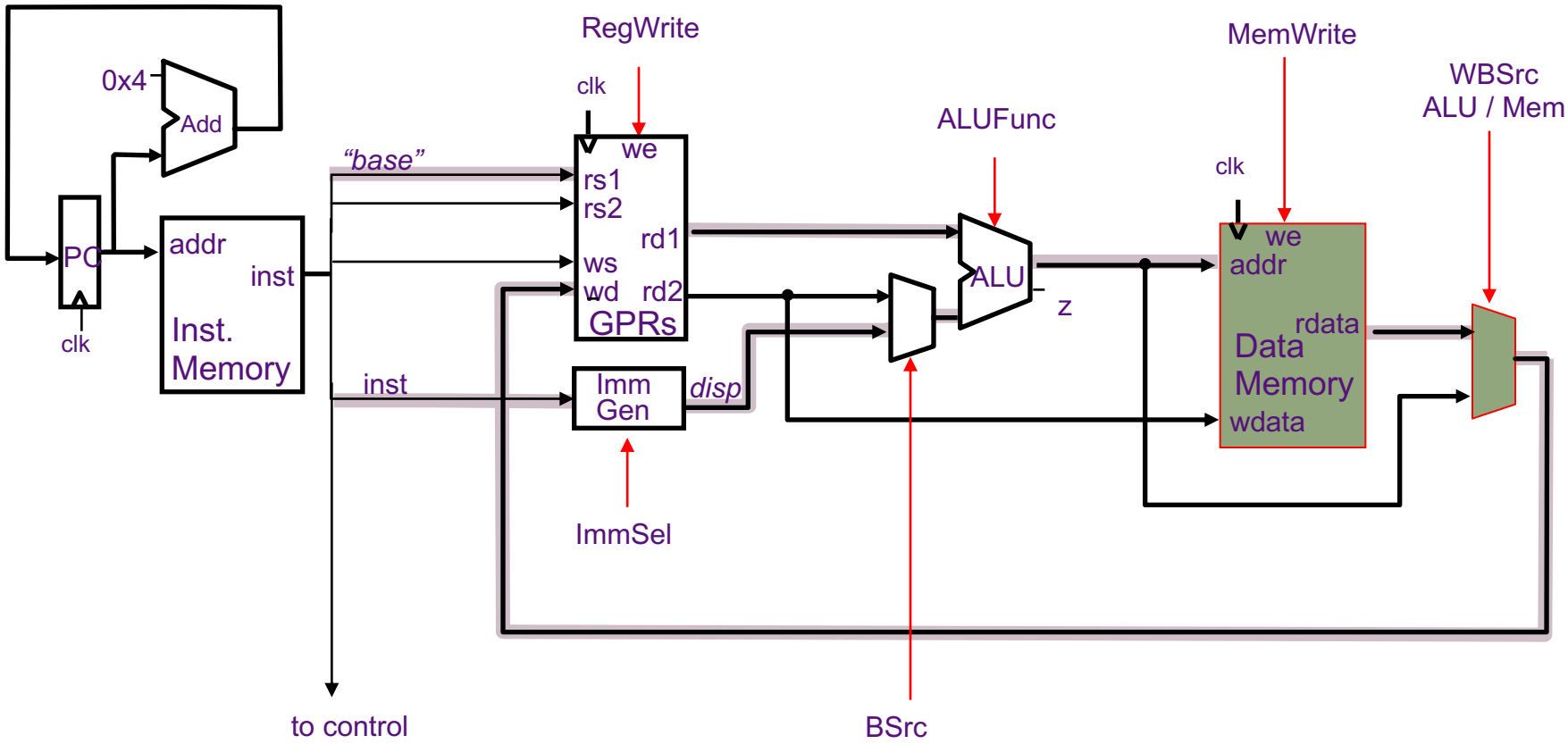


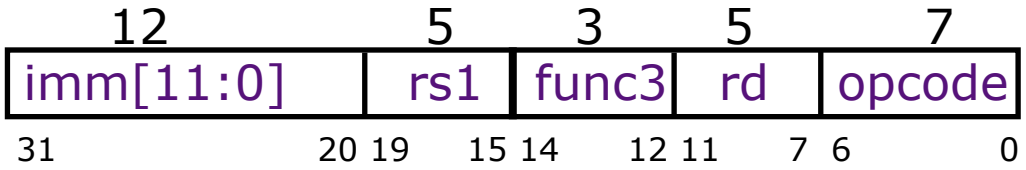
addressing mode
rs1 + imm. displacement

Same encoding as immediate instructions
rs1 contains base address, rd is the destination.

Load Instructions

Harvard Datapath



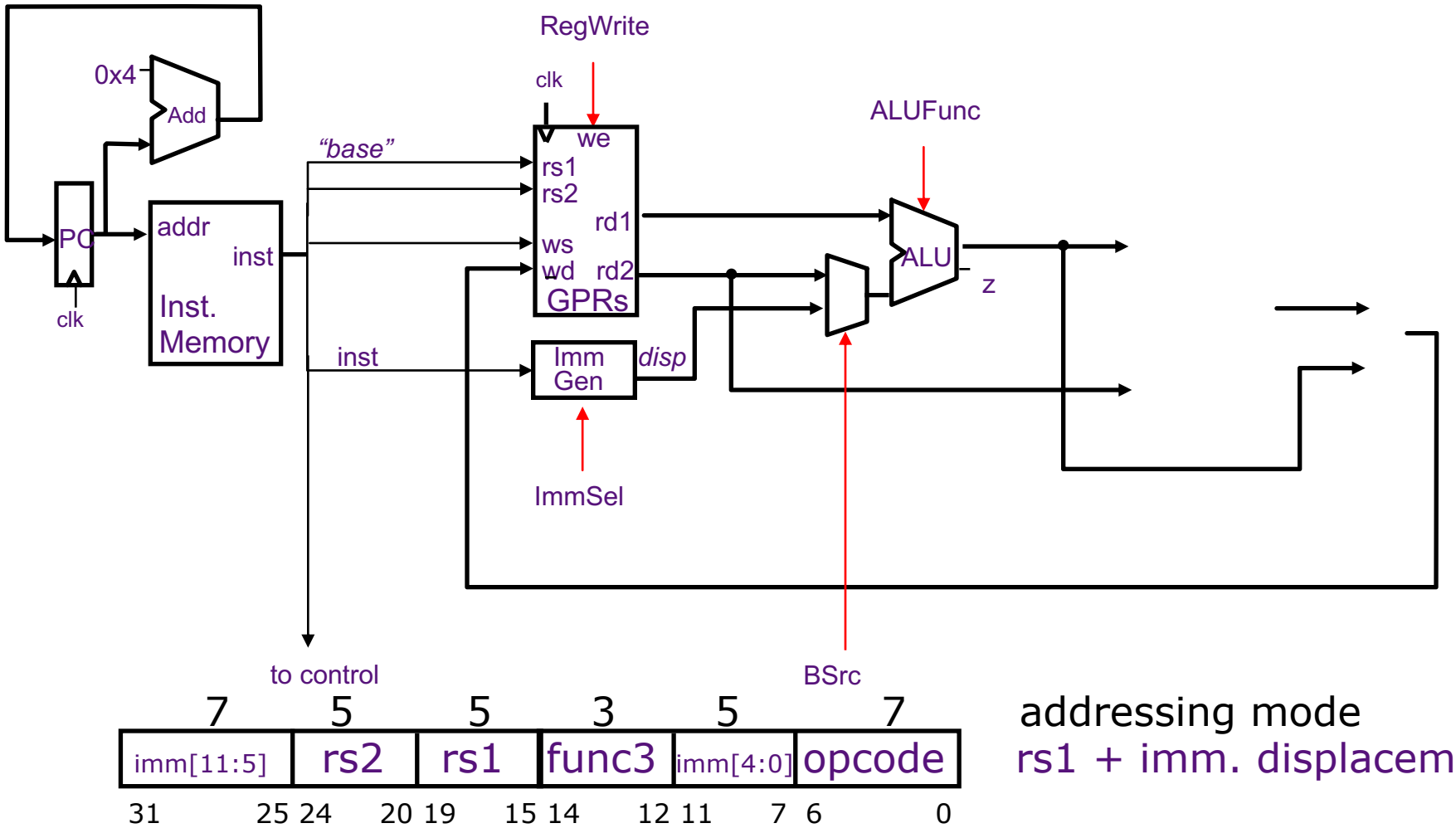


addressing mode
rs1 + imm. displacement

Same encoding as immediate instructions
rs1 contains base address, rd is the destination.

Store Instructions

Harvard Datapath

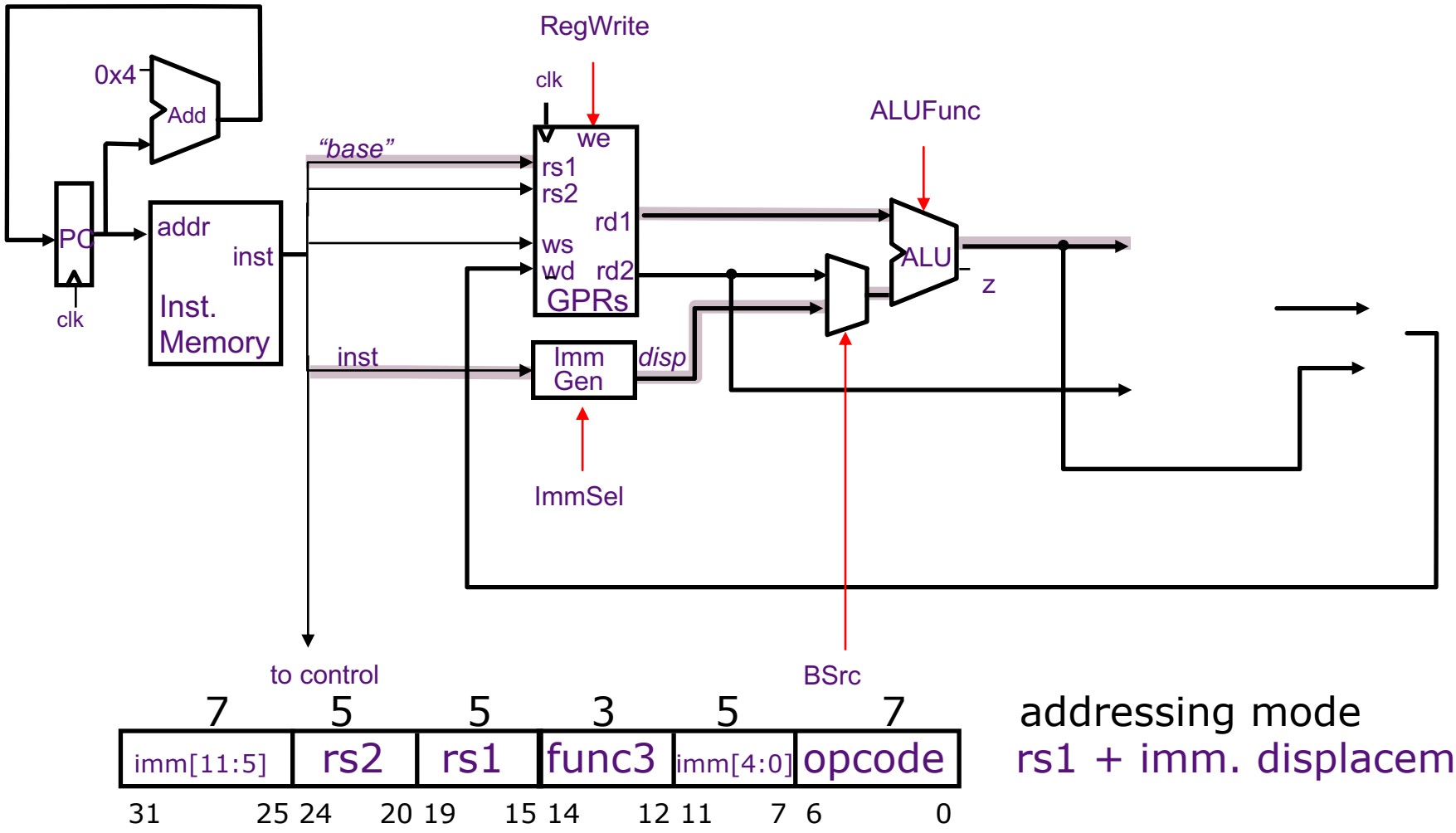


addressing mode
rs1 + imm. displacement

Keep rs1 and rs2 bits in same place, move lower bits of immediate to where rd was

Store Instructions

Harvard Datapath

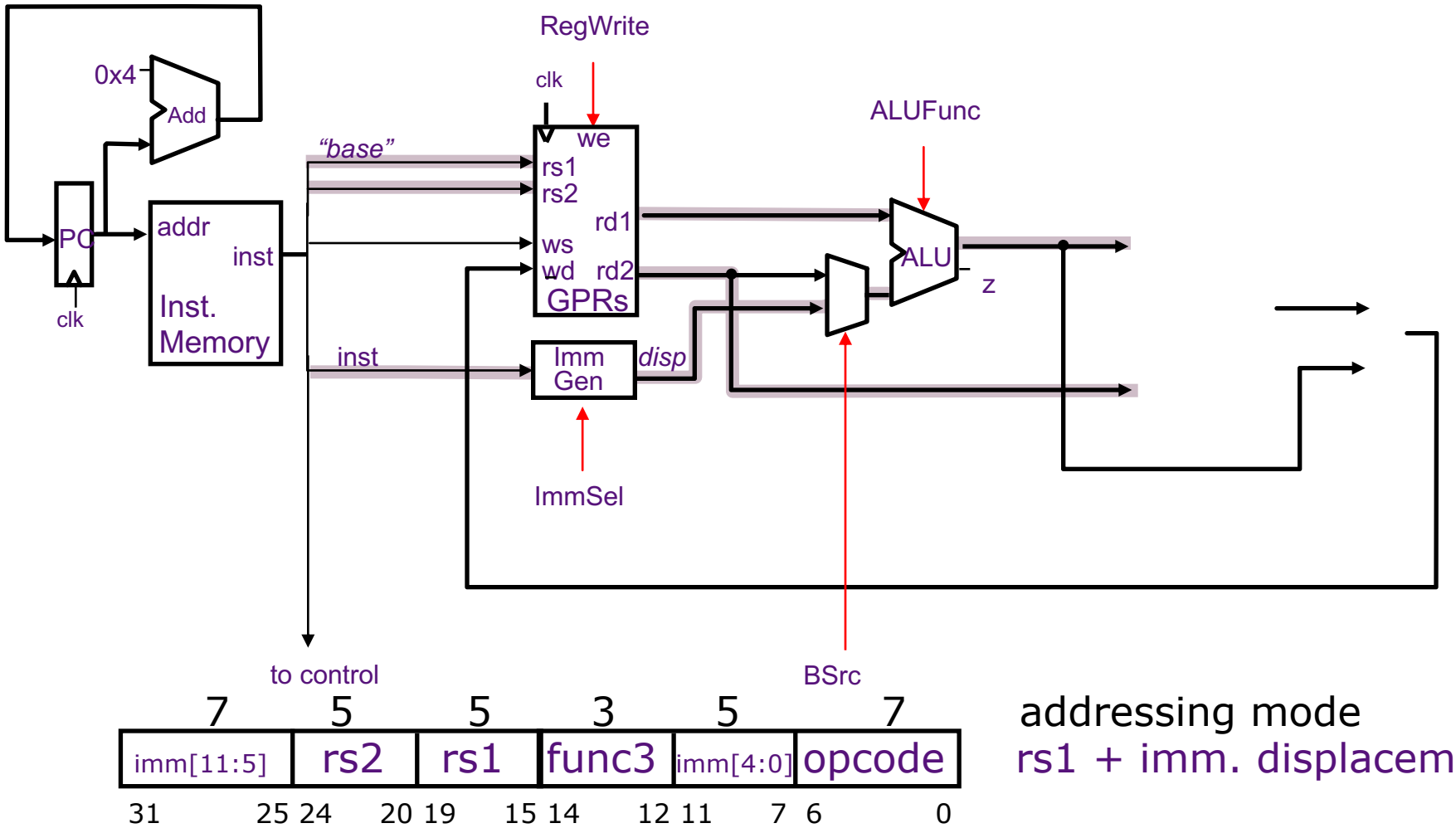


addressing mode
rs1 + imm. displacement

Keep rs1 and rs2 bits in same place, move lower bits of immediate to where rd was

Store Instructions

Harvard Datapath

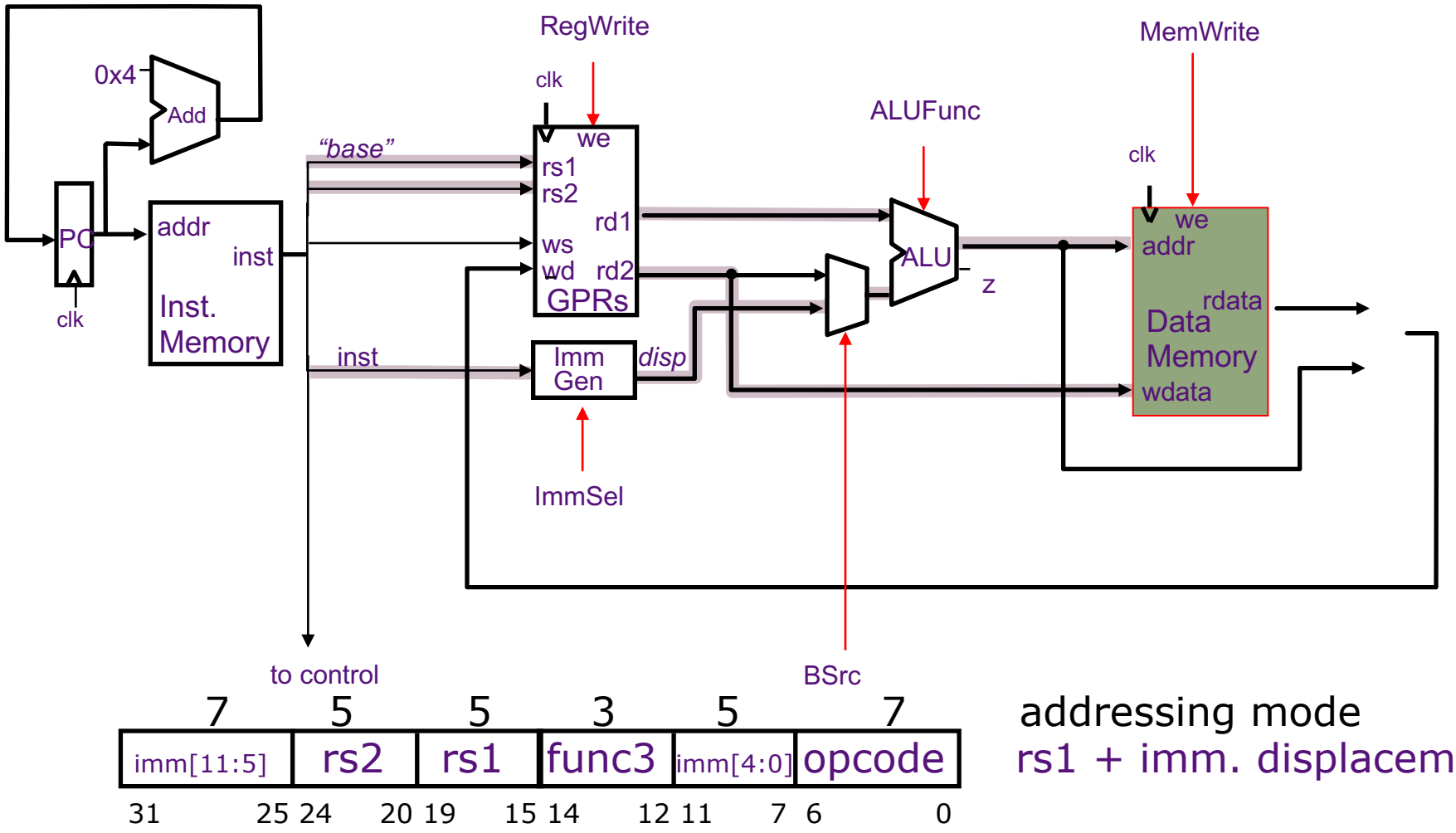


addressing mode
rs1 + imm. displacement

Keep rs1 and rs2 bits in same place, move lower bits of immediate to where rd was

Store Instructions

Harvard Datapath

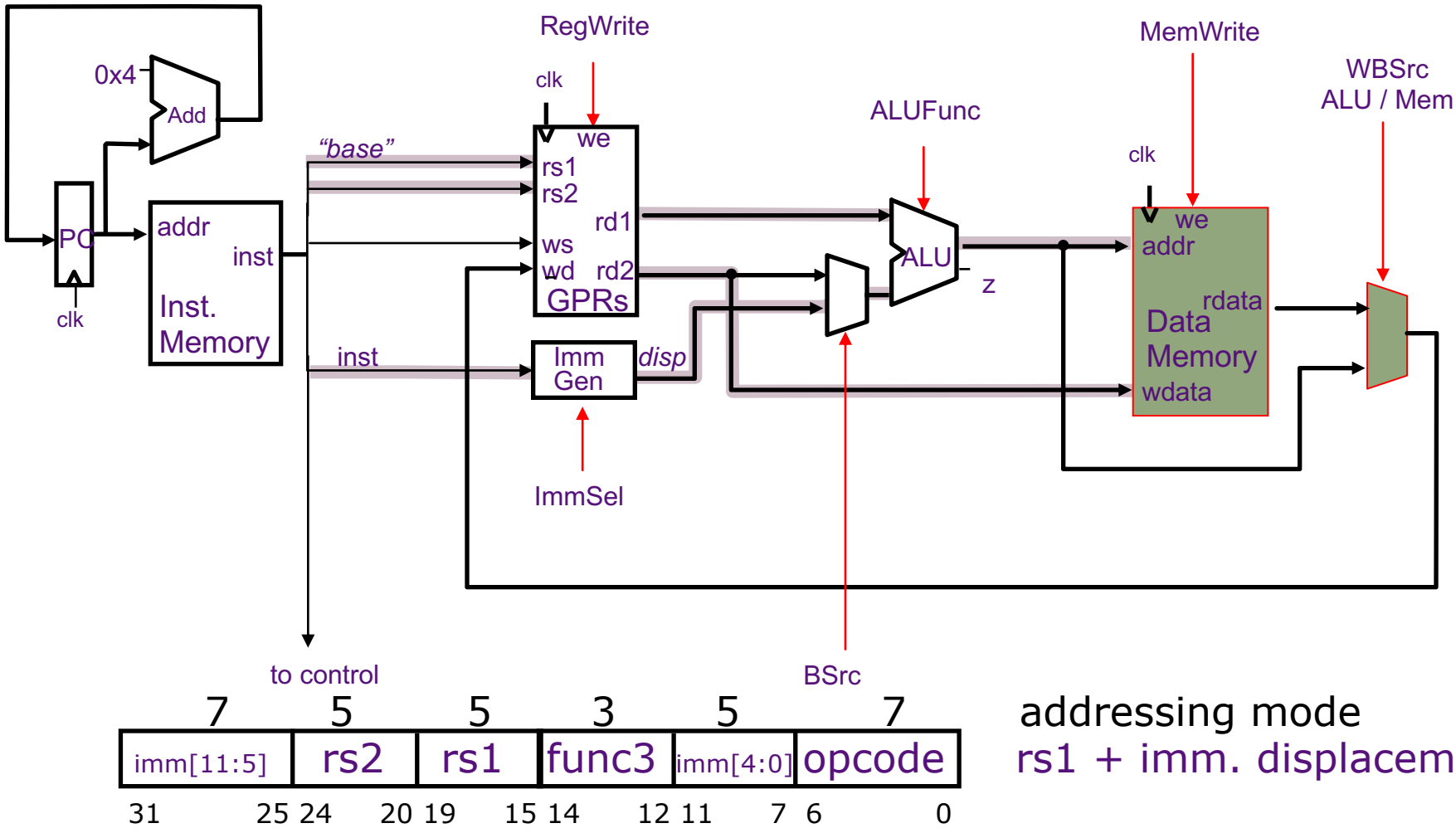


addressing mode
rs1 + imm. displacement

Keep rs1 and rs2 bits in same place, move lower bits of immediate to where rd was

Store Instructions

Harvard Datapath

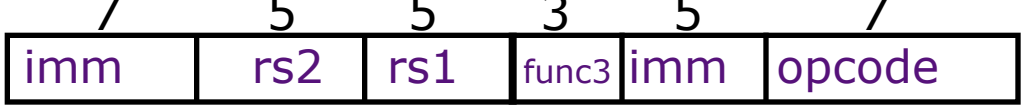


addressing mode
rs1 + imm. displacement

Keep rs1 and rs2 bits in same place, move lower bits of immediate to where rd was

RISC-V Control Instructions

Conditional (on GPR) PC-relative branch



BEQ, BNE

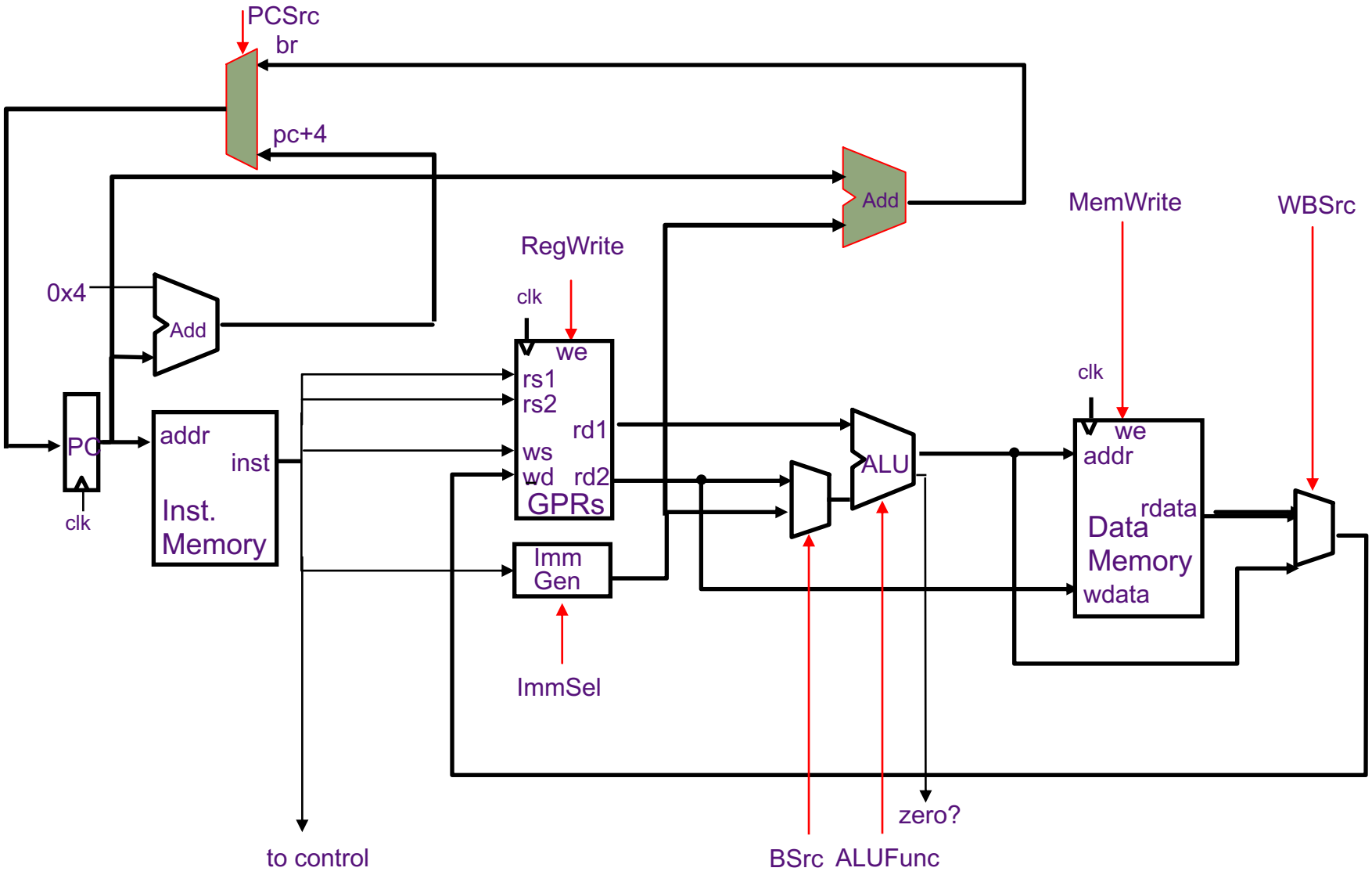
RISC-V Control Instructions

Conditional (on GPR) PC-relative branch

BEQ, BNE

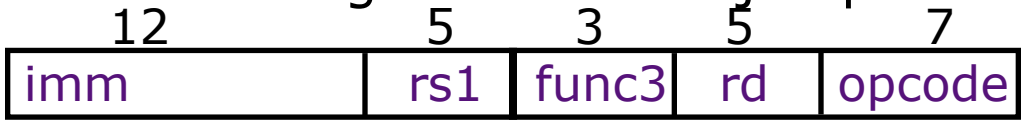
Instruction	Target PC	Condition
BEQ	immB=SXT({imm[12:1],1'b0}) PC+immB	(rs1)==(rs2)
BNE		(rs1)!=(rs2)
BLT		(rs1)<(rs2)
BGE		(rs1)>=(rs2)
BLTU		(rs1)<(rs2) (unsigned)
BGEU		(rs1)>=(rs2) (unsigned)

Conditional Branches



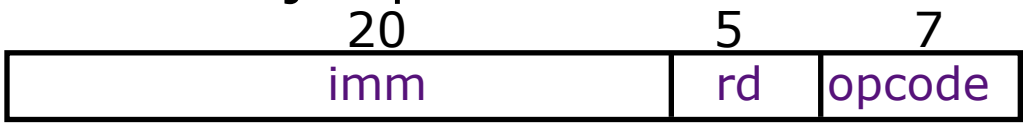
RISC-V Control Instructions

Unconditional register-indirect jumps



JALR

Unconditional jumps

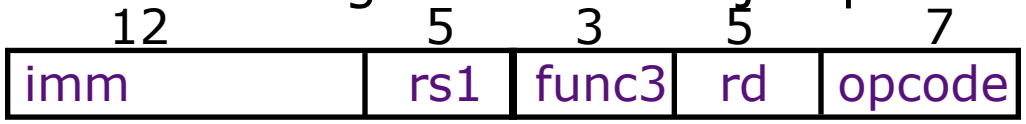


JAL

- Jump-&-link stores PC+4 into the destination register (rd)

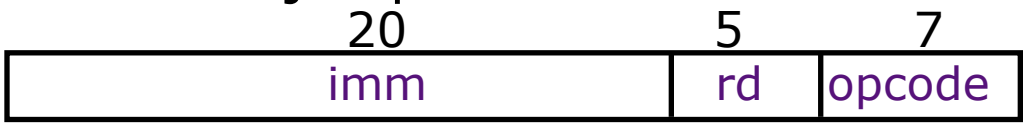
RISC-V Control Instructions

Unconditional register-indirect jumps



JALR

Unconditional jumps

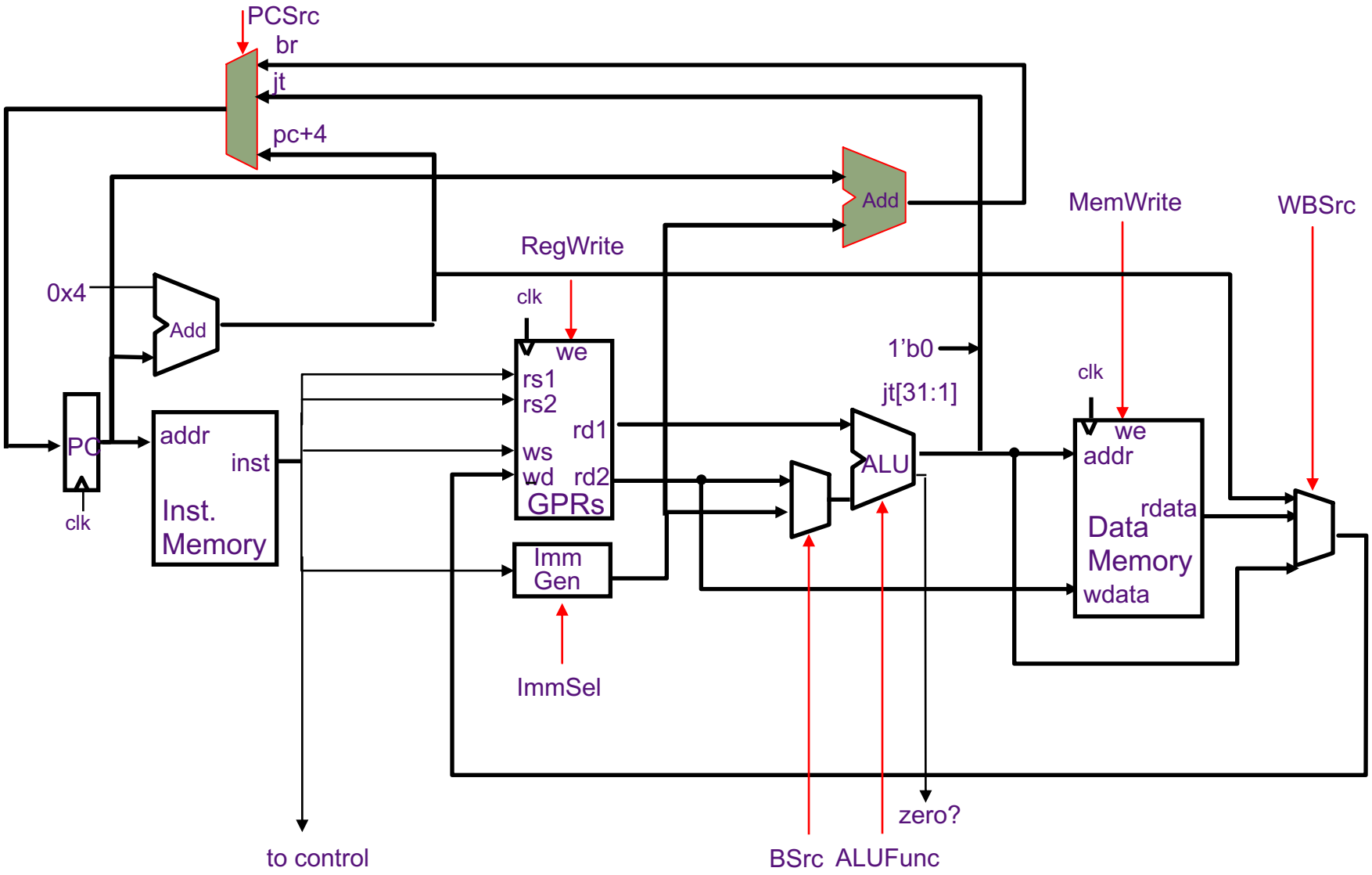


JAL

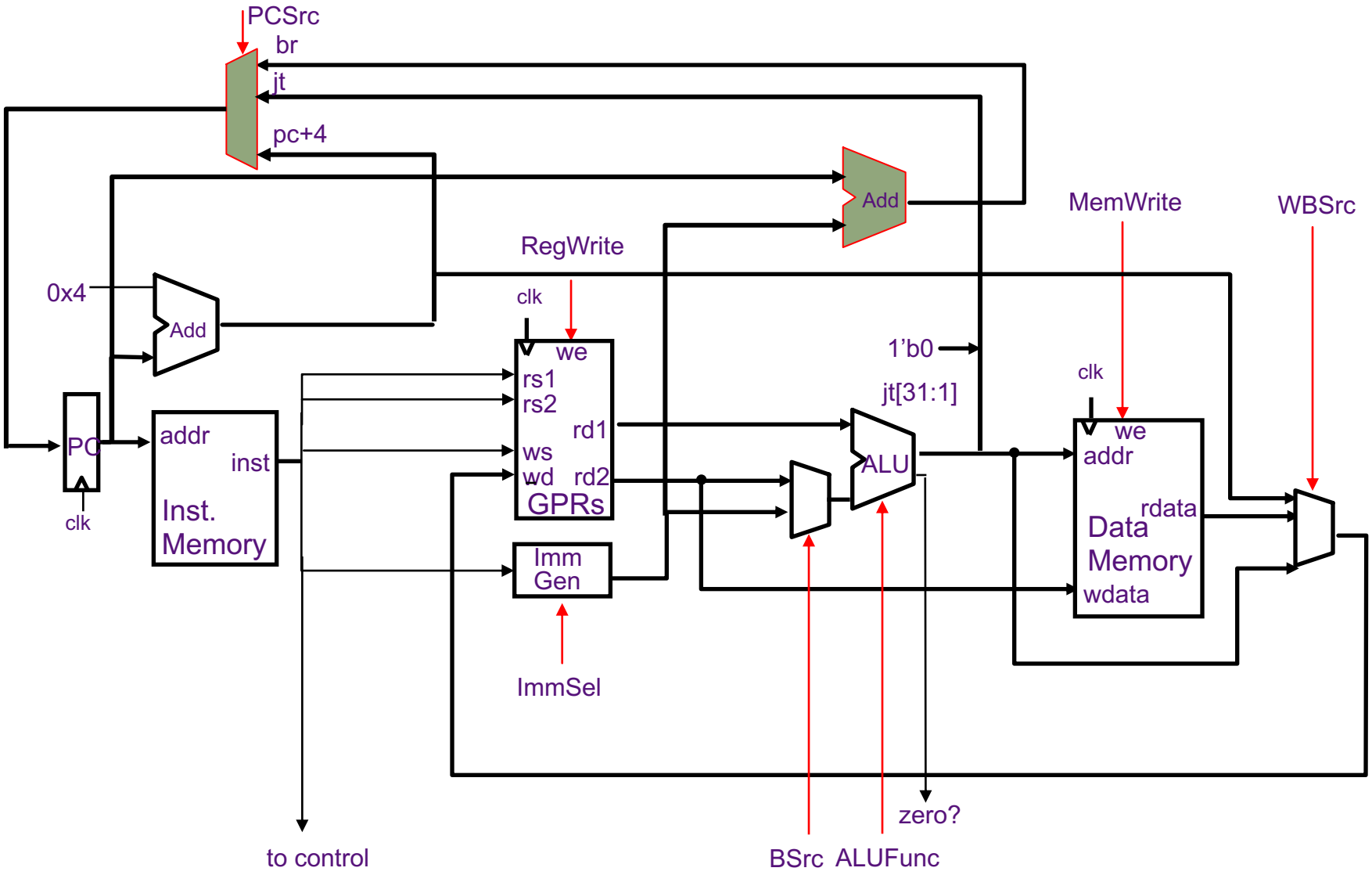
Instruction	Target PC	Condition
JALR	immI = SXT(imm[11:0]) {((rs)+immI)[31:1],1'b0}	Always Taken
JAL	immU=SXT({imm[20:1],1'b0}) PC+immU32	Always Taken

- Jump-&-link stores PC+4 into the destination register (rd)

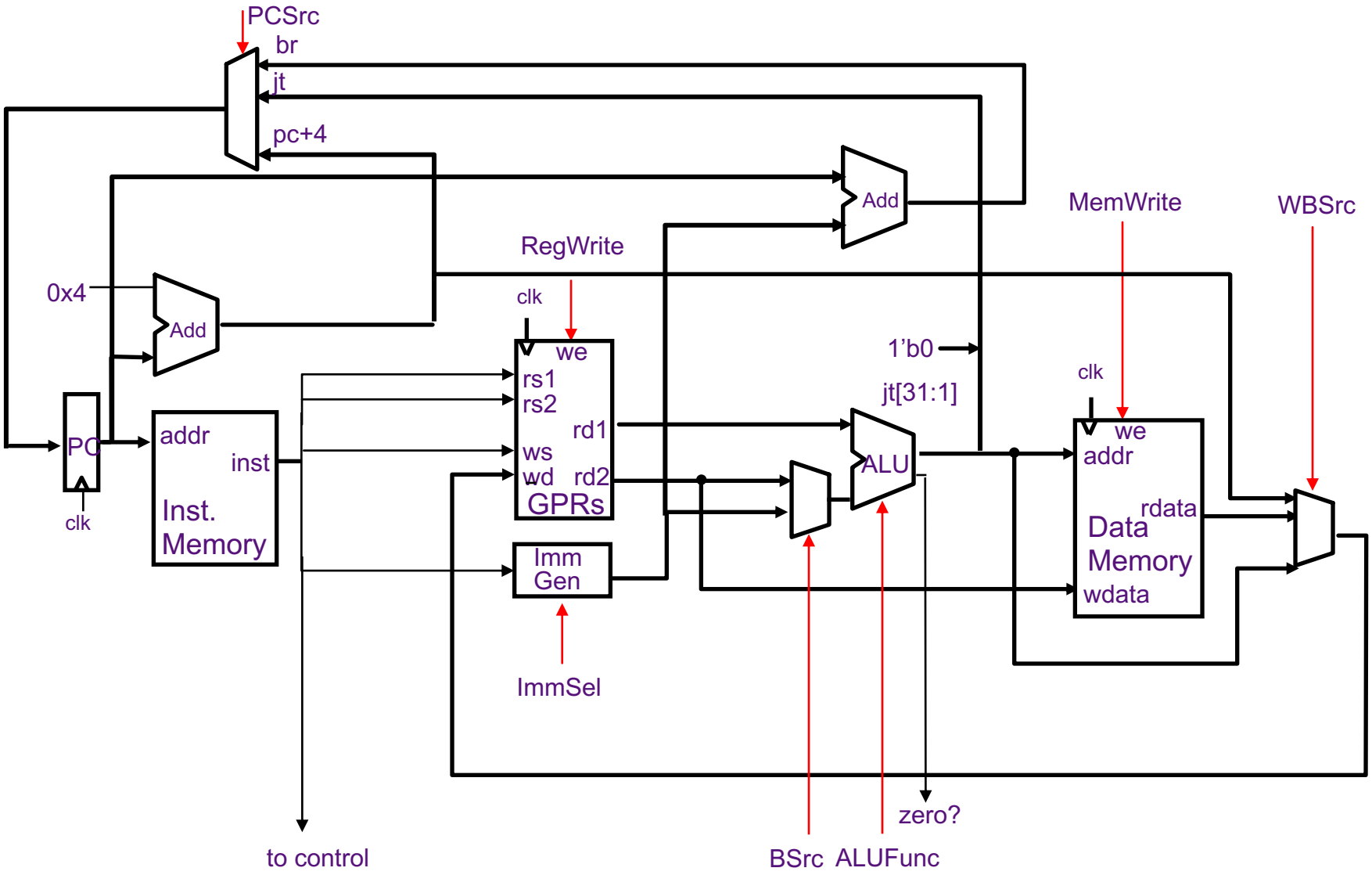
Register-Indirect Jump-&-Link (JALR)



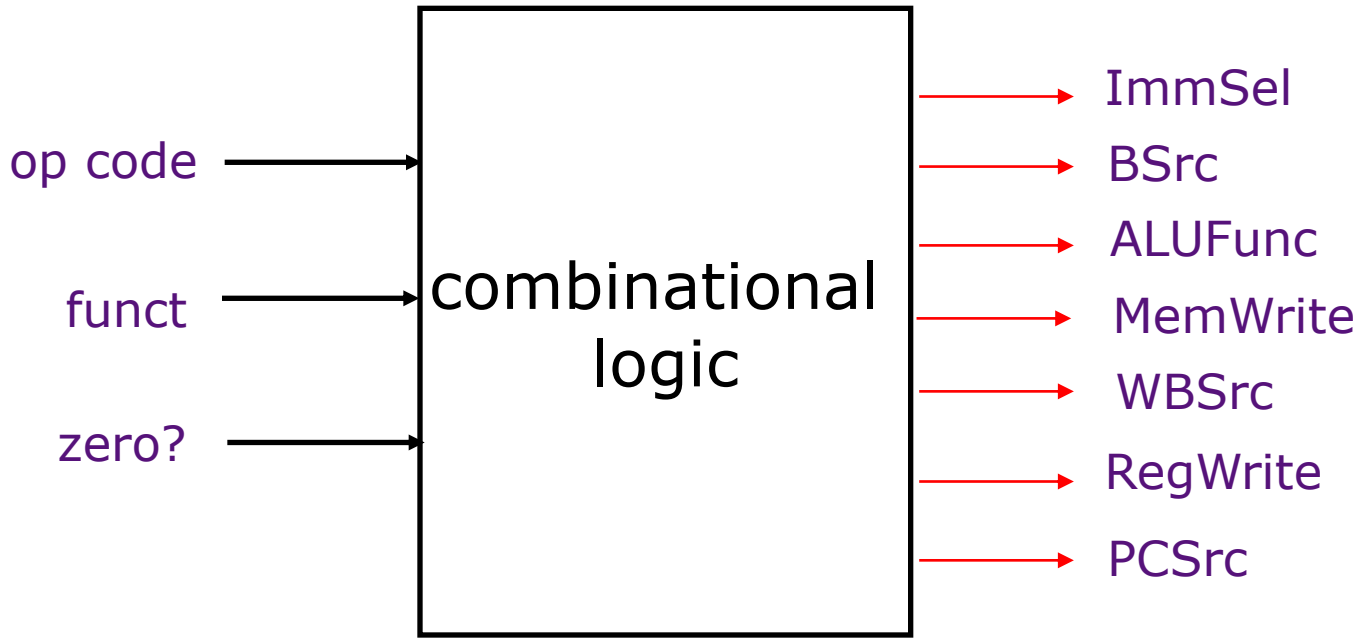
Absolute Jumps (JAL)



Harvard-Style Datapath for RISC-V



Hardwired Control is pure Combinational Logic



Hardwired Control Table

Opcode	ImmSel	BSrc	ALU	MemW	WBSrc	RegWr	PCSrc
ALU							
ALUi							
LW							
SW							
BRtaken							
BRnotTaken							
JALR							
JAL							

BSrc = Reg / Imm WBSrc = ALU / Mem / pc+4
PCSrc = pc+4 / br / rind

Hardwired Control Table

Opcode	ImmSel	BSrc	ALU	MemW	WBSrc	RegWr	PCSrc
ALU	*	Reg	funct	no	ALU	yes	pc+4
ALUi	SXT(imm[11:0])	Imm	funct	no	ALU	yes	pc+4
LW	SXT(imm[11:0])	Imm	+	no	Mem	yes	pc+4
SW	SXT(imm[11:0])	Imm	+	yes	*	no	pc+4
BRtaken	ImmB	Reg	funct	no	*	no	br
BRnotTaken	ImmB	Reg	funct	no	*	no	pc+4
JALR	ImmI	Imm	+	no	pc+4	yes	jt
JAL	ImmU	Imm	*	no	pc+4	yes	br

BSrc = Reg / Imm WBSrc = ALU / Mem / pc+4
PCSrc = pc+4 / br / rind

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

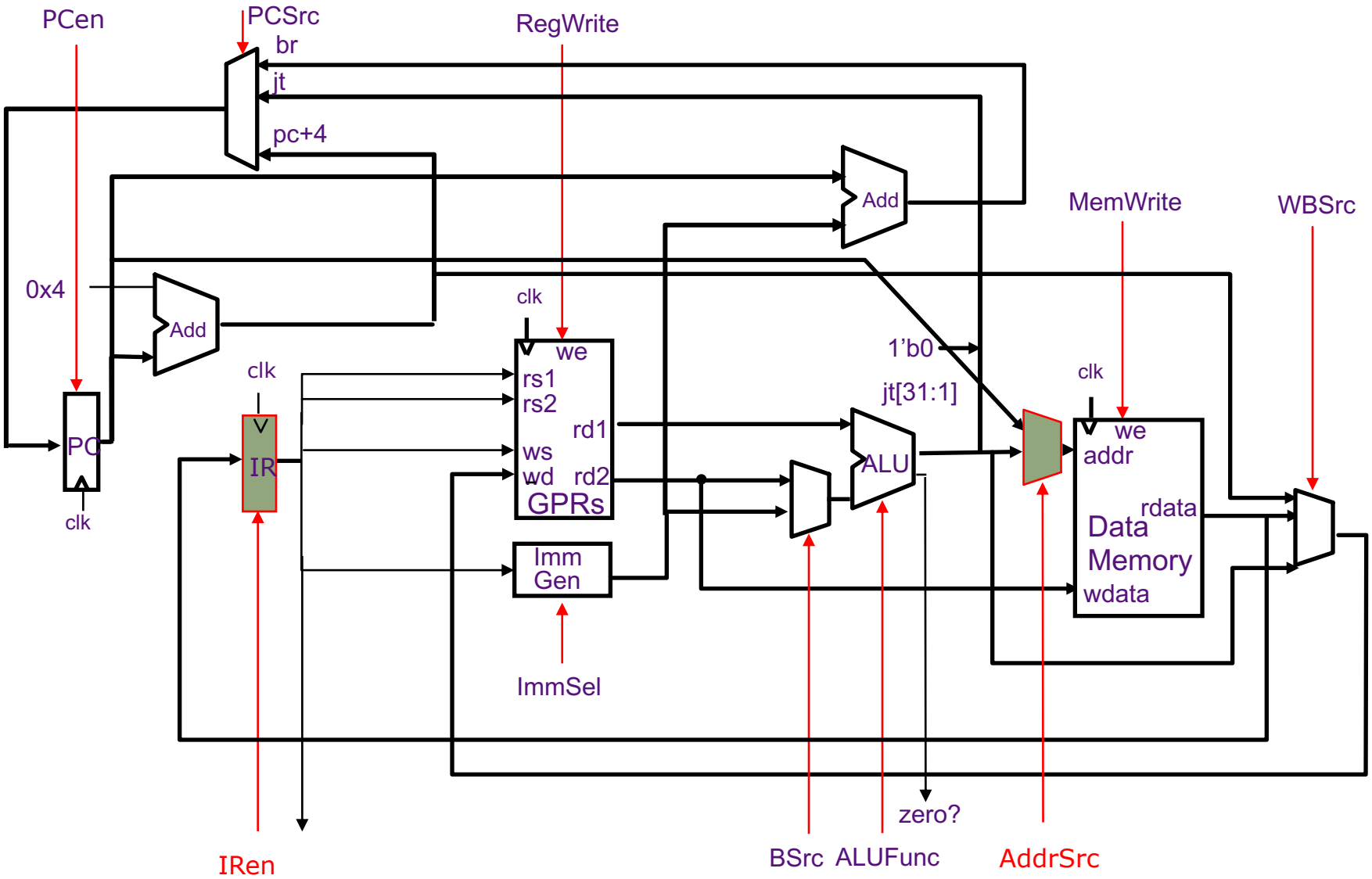
$$\Rightarrow\ t_C > t_{IFetch} + t_{RFetch} + t_{ALU} + t_{DMem} + t_{RWB}$$

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

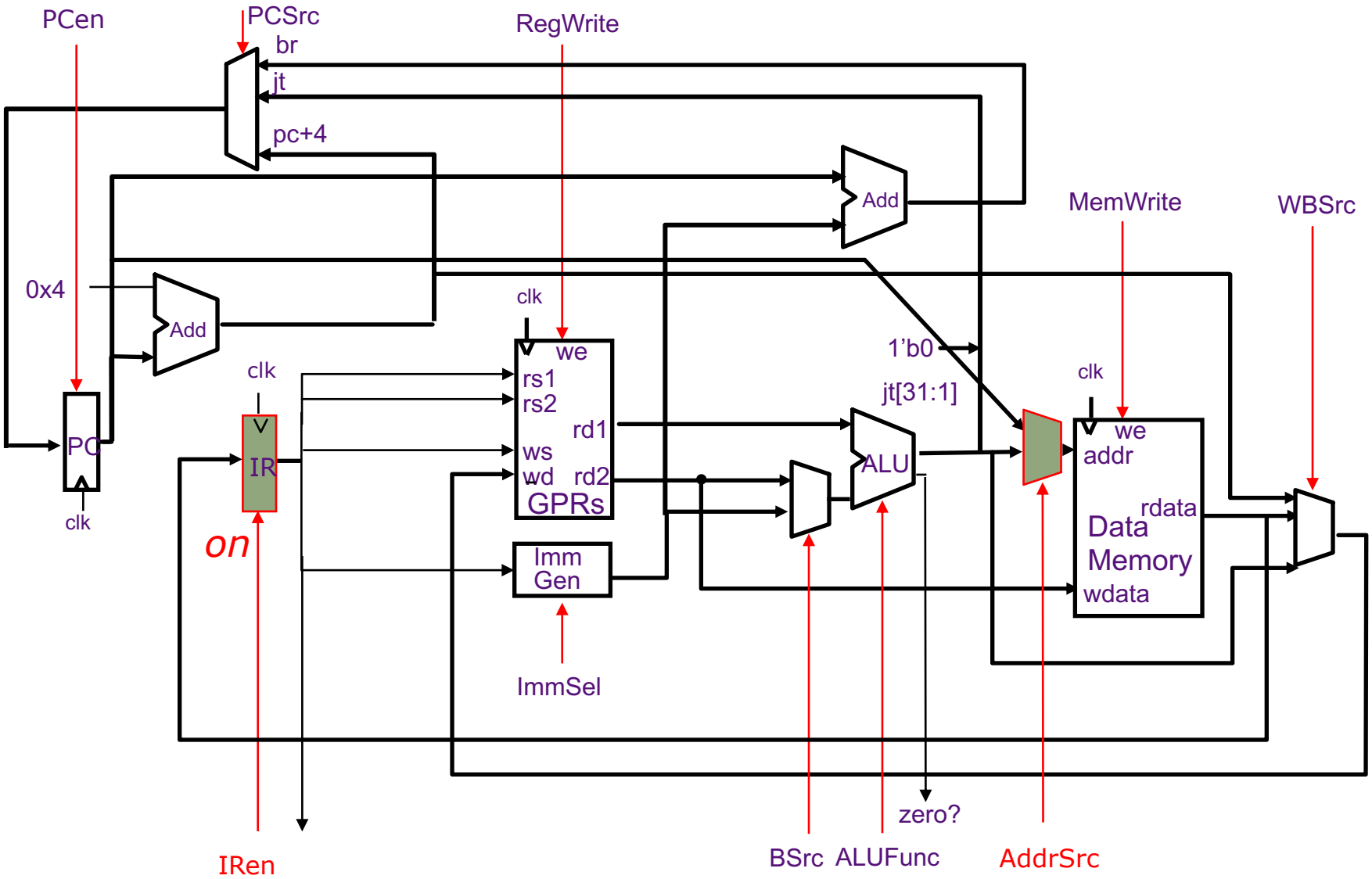
Princeton challenge

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

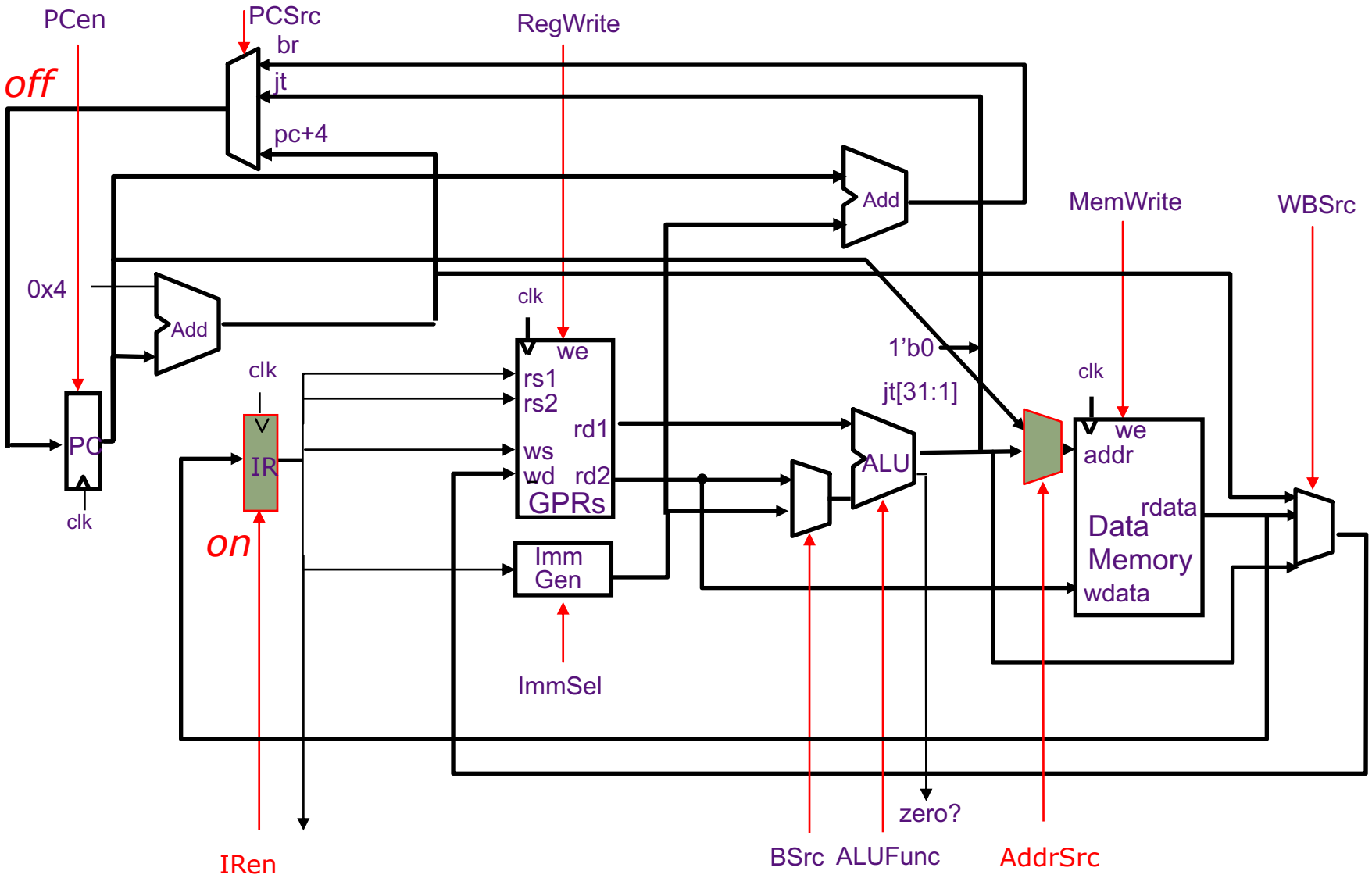
Princeton Microarchitecture
Datapath & Control



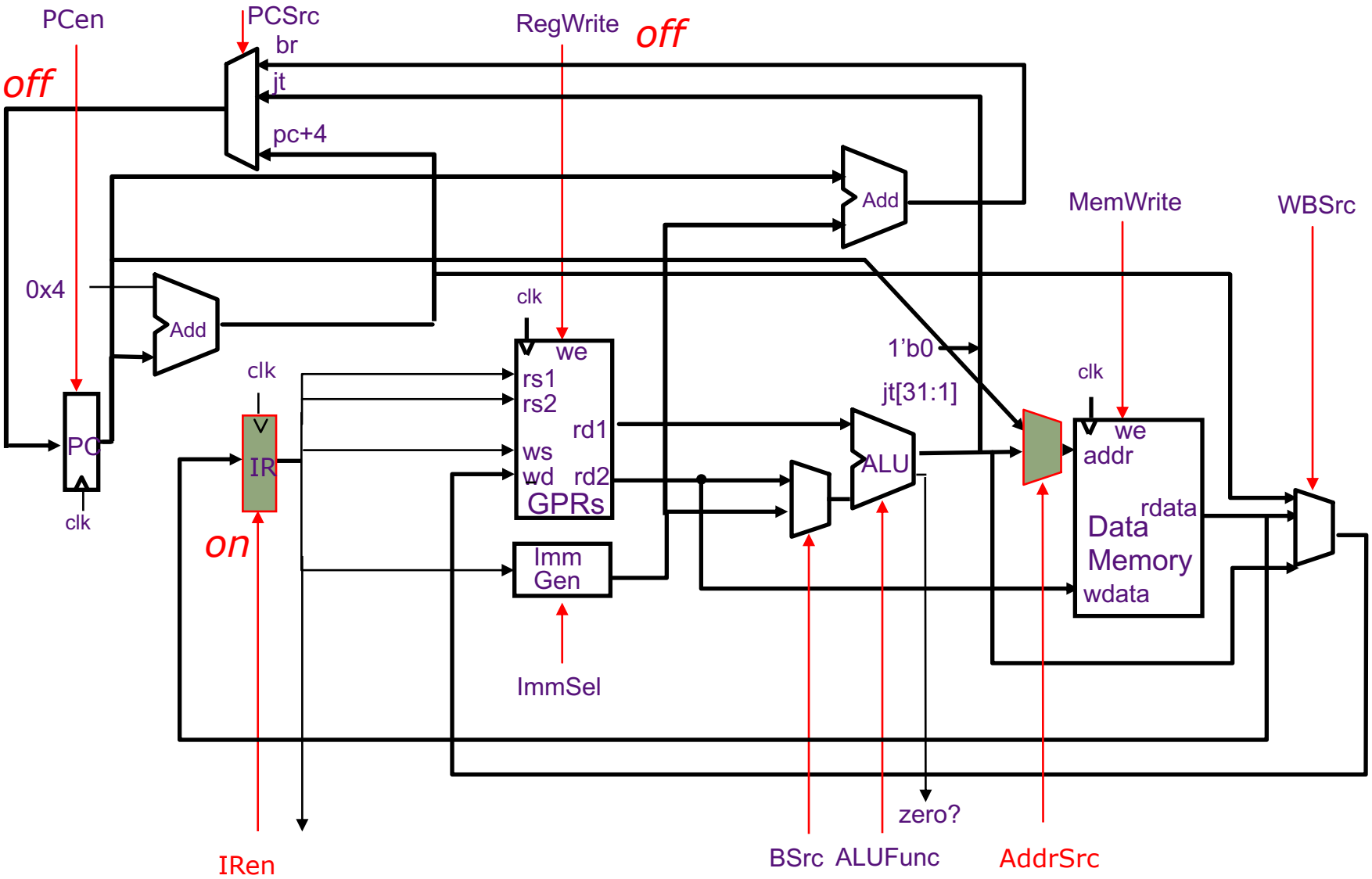
Princeton Microarchitecture
Datapath & Control



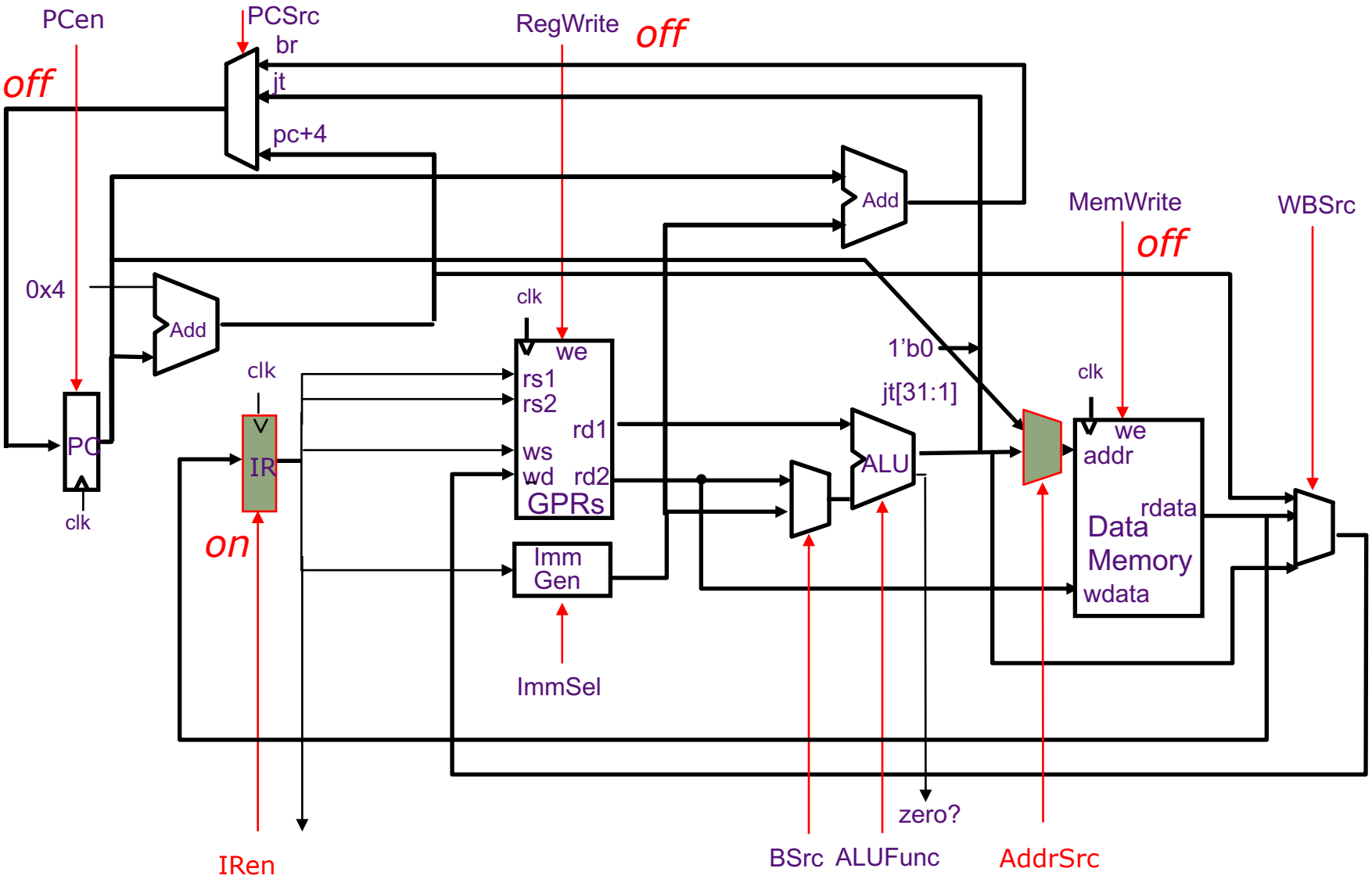
Princeton Microarchitecture
Datapath & Control



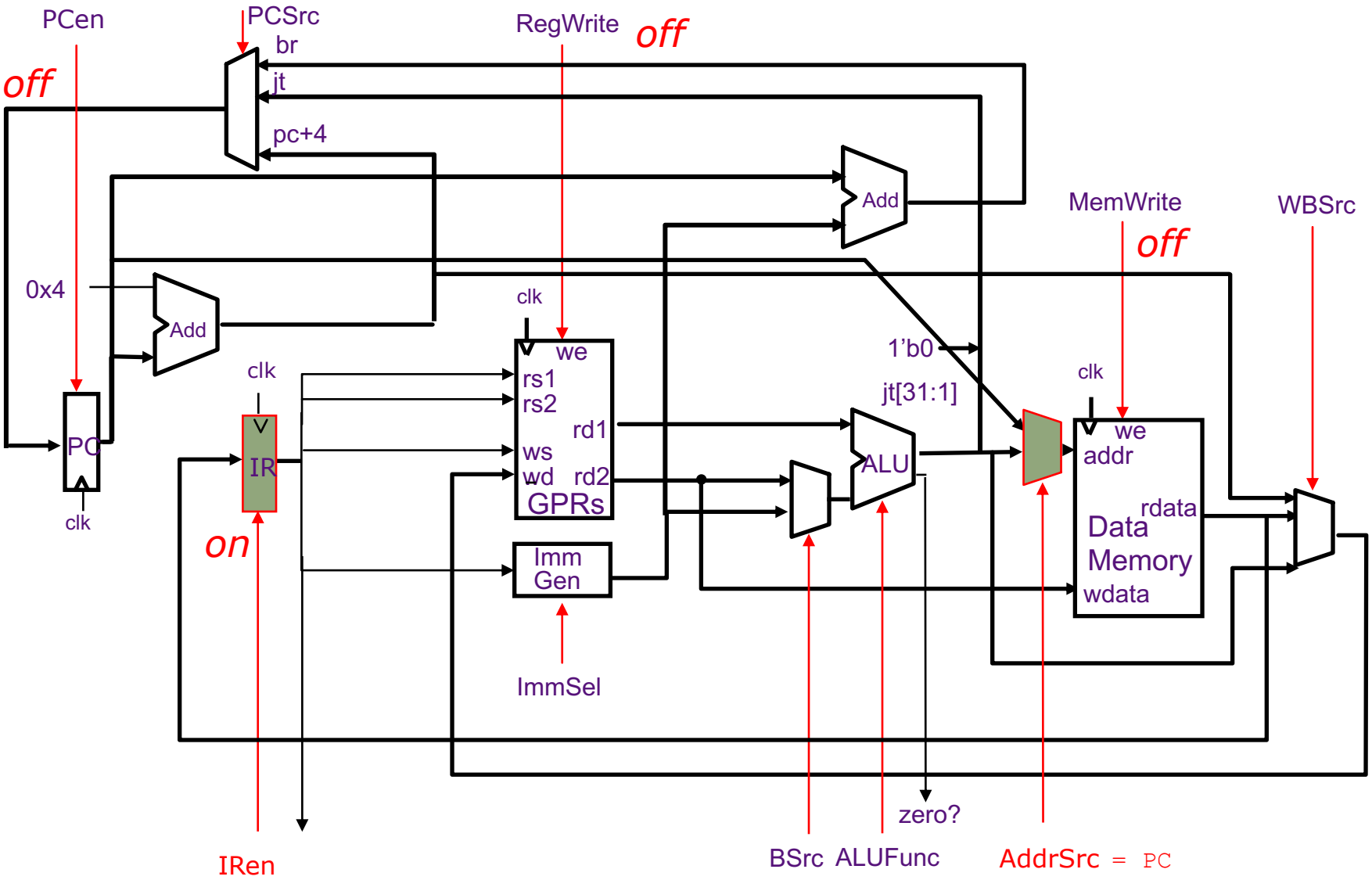
Princeton Microarchitecture
Datapath & Control



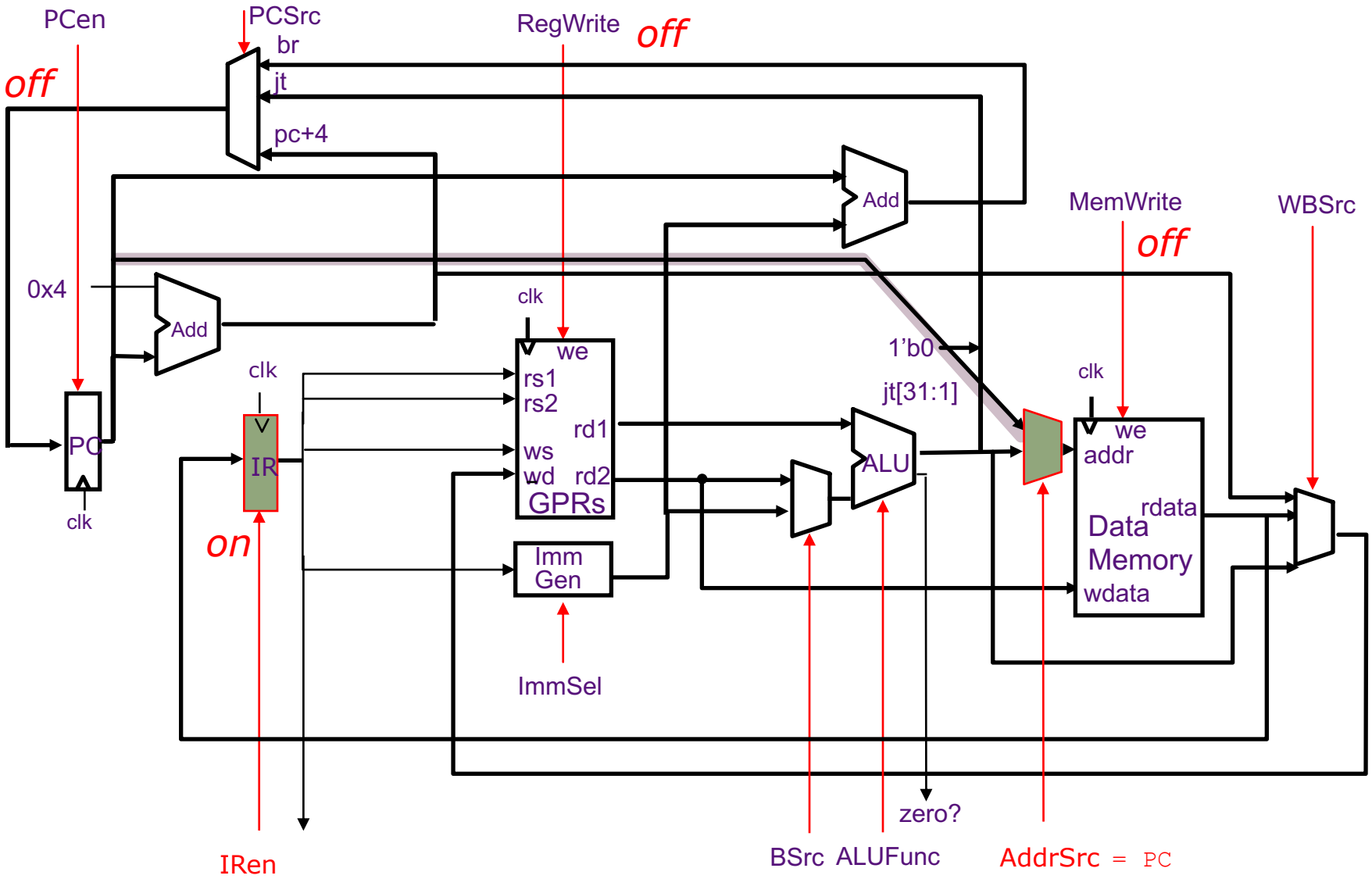
Princeton Microarchitecture
Datapath & Control



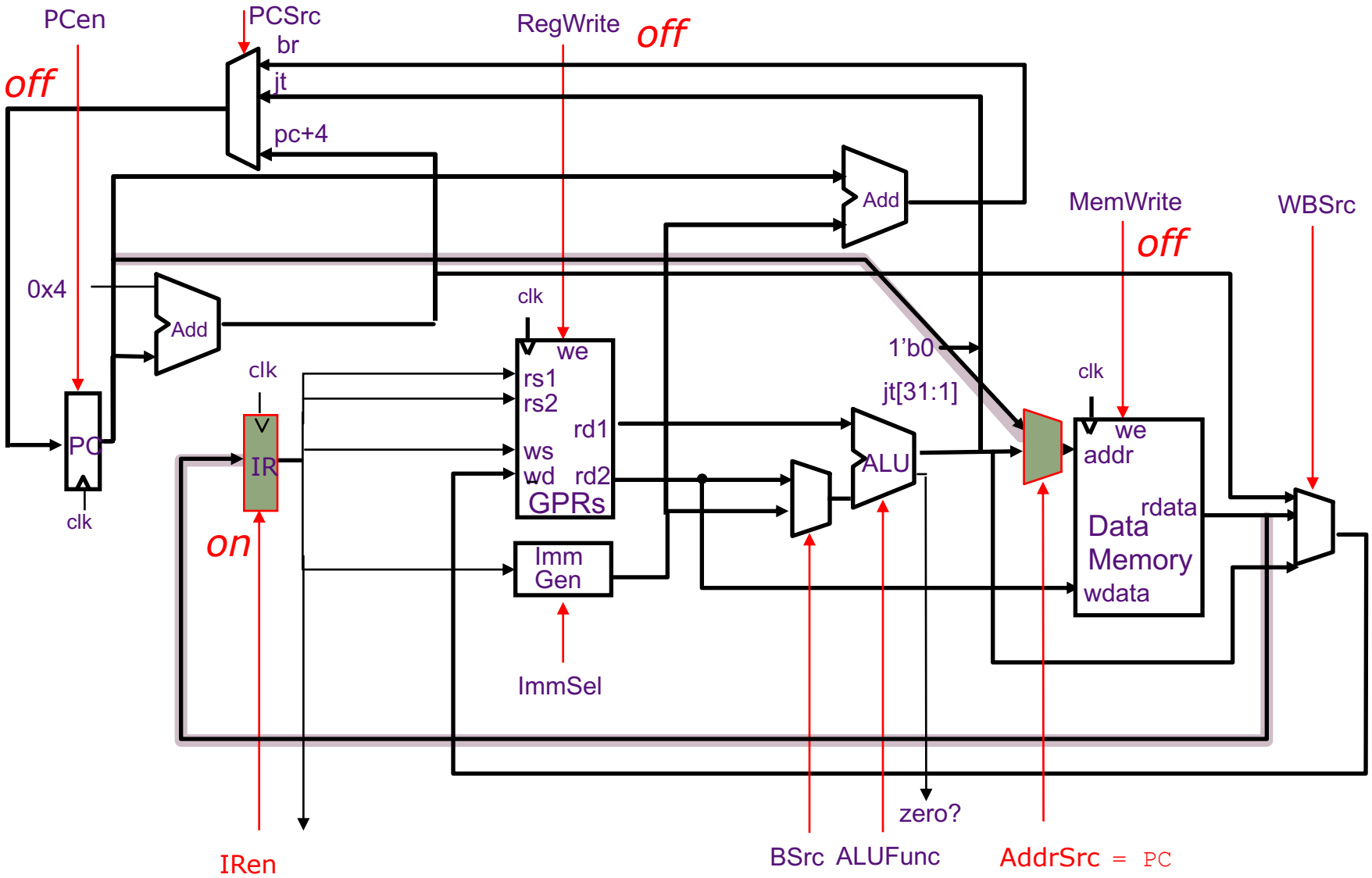
Princeton Microarchitecture
Datapath & Control



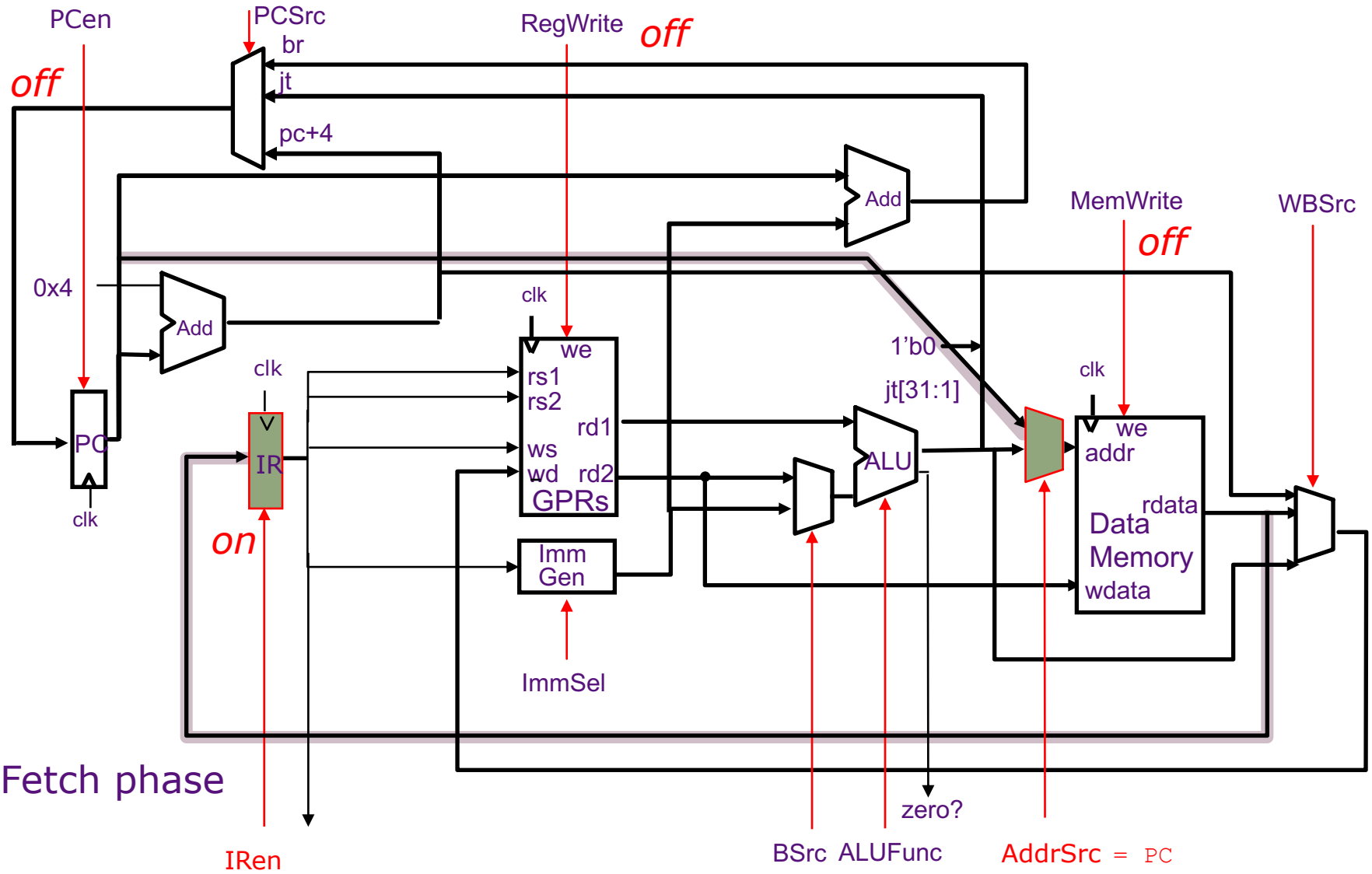
Princeton Microarchitecture
Datapath & Control



Princeton Microarchitecture
Datapath & Control



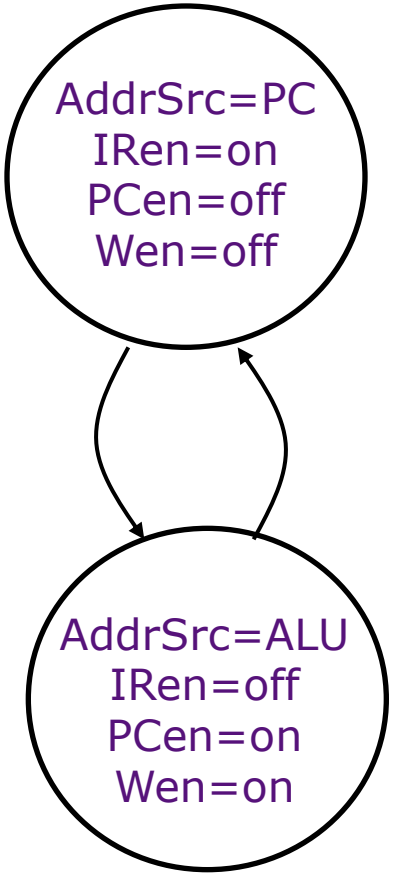
Princeton Microarchitecture
Datapath & Control



Two-State Controller:
Princeton Architecture

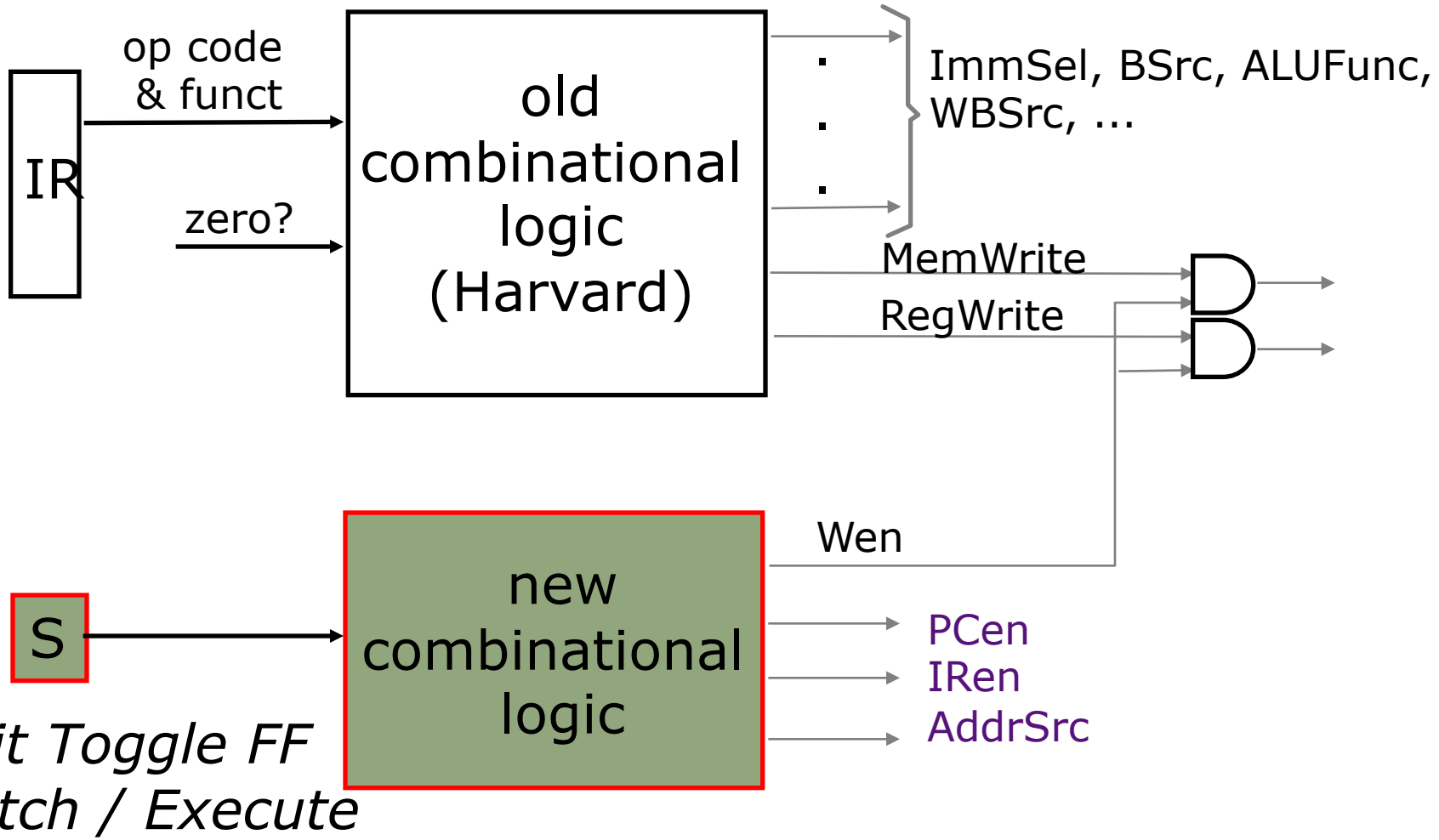
fetch phase

execute phase



A flipflop can be used to remember the phase

Hardwired Controller:
Princeton Architecture



Clock Rate vs CPI

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

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

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

Suppose $t_M >> t_{RF} + t_{ALU} + t_{WB}$

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

$CPI_{Princeton} = 2$

$CPI_{Harvard} = 1$

No difference in performance!

Is it possible to design a controller for the Princeton architecture with CPI < 2 ?

CPI = Clock cycles Per Instruction

Stay tuned!