ECE 552 / CPS 550
Advanced Computer Architecture I

Lecture 4
Reduced Instruction Set Computers

Benjamin Lee
Electrical and Computer Engineering
Duke University

www.duke.edu/~bcl15
www.duke.edu/~bcl15/class/class_ece252fall11.html

Microprogramming in 1970's

Technology

- ROMs were faster than DRAMs

Instructions

- For complex instruction sets (CISC), datapath and controller were cheaper and simpler
- New instructions (e.g., floating point) supported without datapath modifications

Compatibility

- ISA compatibility across machine models were cheaper and simpler
- Fixing bugs in the controller was easier

In the 1970s , except for cheapest and fastest machines, all computers were microprogrammed

Microprogramming in 1980's

Increasing Complexity

- CISC ISAs led to subroutine and call stacks in microcode
- Fixing bugs in control conflicts with read-only nature of ROMs

Technology

- Advent of VLSI technology
- Assumptions about ROM vs RAM speed became invalid

Instructions

- Better compilers made complex instructions less important
- Compilers had difficulty using complex instructions

Microarchitecture

- Microarchitectural innovations: pipelining, caches and buffers, etc.
- Make multiple-cycle execution of reg-reg instructions unattractive

Modern Microprogramming

Microprogramming is far from extinct

- Played crucial role in microprocessors of 1980's (e.g., Intel 386, 486)
- Plays assisting role in modern microprocessors

Assisting role in modern microprocessors

- Example: AMD Athlon, Intel Core 2 Duo, IBM Power PC
- Most instructions executed directly (hardwired control)
- Infrequently-used, complicated instructions invoke microcode engine

Assisting role in modern microprocessors

- Patchable microcode common for post-fabrication bug fixes
- Example: Intel Pentiums load microcode patches at bootup

CISC to RISC

Instruction Management

- Shift away from fixed hardware microcode, microroutines
- Exploit fast RAM to build instruction cache of user-visible instructions
- Adapt contents of fast instruction memory to fit what application needs at the moment.

Simple Instruction Set

- Shift away from complex CISC instructions, which are rarely used
- Enable hardwired, pipelined implementation

Greater Integration

- In early 1980s, able to fit 32-bit datapath and small cache on die
- Allow faster operation by avoiding chip crossings in common case

CDC 6600

Seymore Cray, 1964

- Fast, pipelined machine with 60-bit words
- Ten functional units (floating-point, integer, etc.)

Control

- Hardwired control, no microcoding
- Dynamic instruction scheduling with a scoreboard

System Organization

- Ten peripheral processors for input/output
- Fast time-shared 12-bit integer ALU
- Very fast clock, 10MHz
- Novel Freon-based technology for cooling

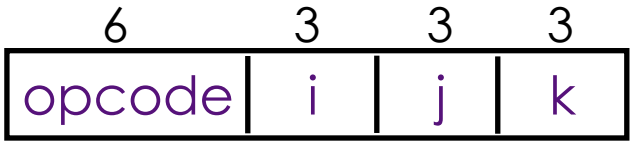
CDC 6600: Load/Store Architecture

Separate instructions manipulate three register types

- 8, 60-bit data registers
- 8, 18-bit address registers
- 8, 18-bit index registers

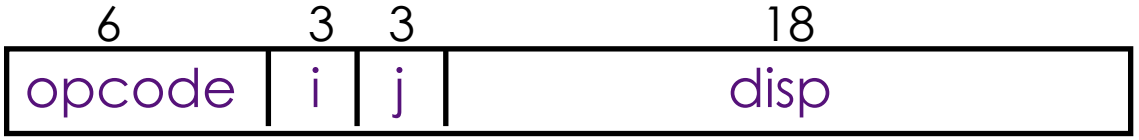
Arithmetic and logic instructions are reg-to-reg

- Hardwired control, no microcoding
- Dynamic instruction scheduling with a scoreboard



Ri ← (Rj) op (Rk)

Only load and store instructions refer to memory



Ri ← M[(Rj) + disp]

Performance Factors

Latency = (Instructions / Program) x (Cycles / Instruction) x (Seconds / Cycle)

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

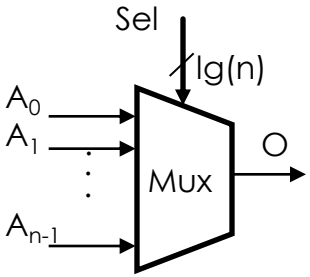
Microarchitecture	Cycles/Instruction	Seconds/Cycle
Microcoded	>1	short
Single-cycle unpipelined	1	long
Pipelined	1	short

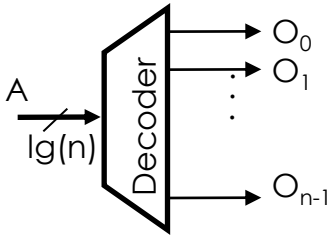
- This lecture presents single-cycle unpipelined microarchitecture

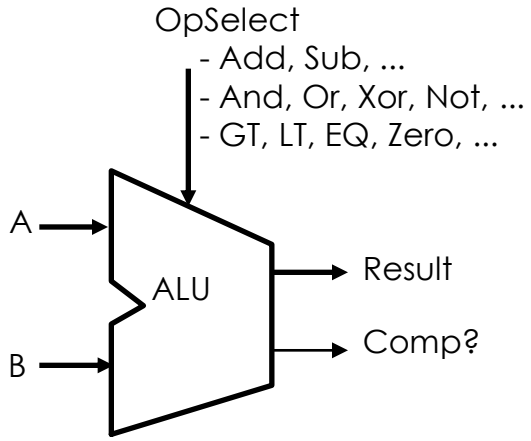
Hardware Elements

Combinational Circuits

- Mux, Decoder, ALU, ...

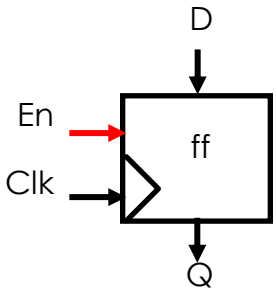


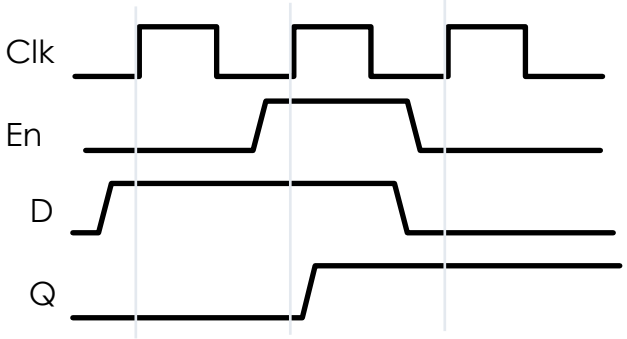




Synchronous State Elements

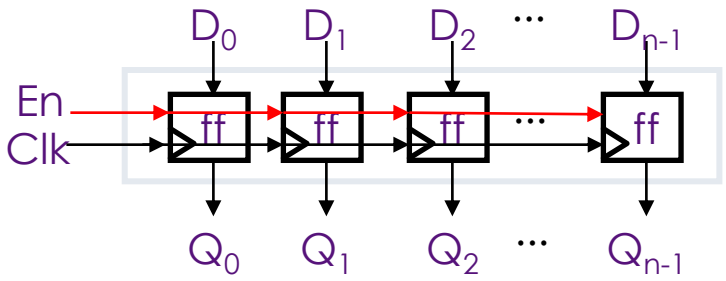
- Flipflop, Register, Register file, SRAM, DRAM
- Edge-triggered elements where data is sampled on rising edge



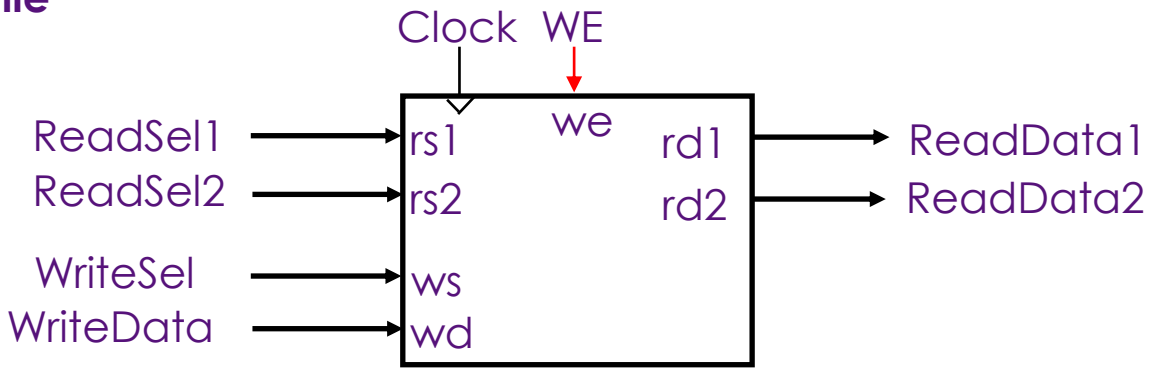


Register Files

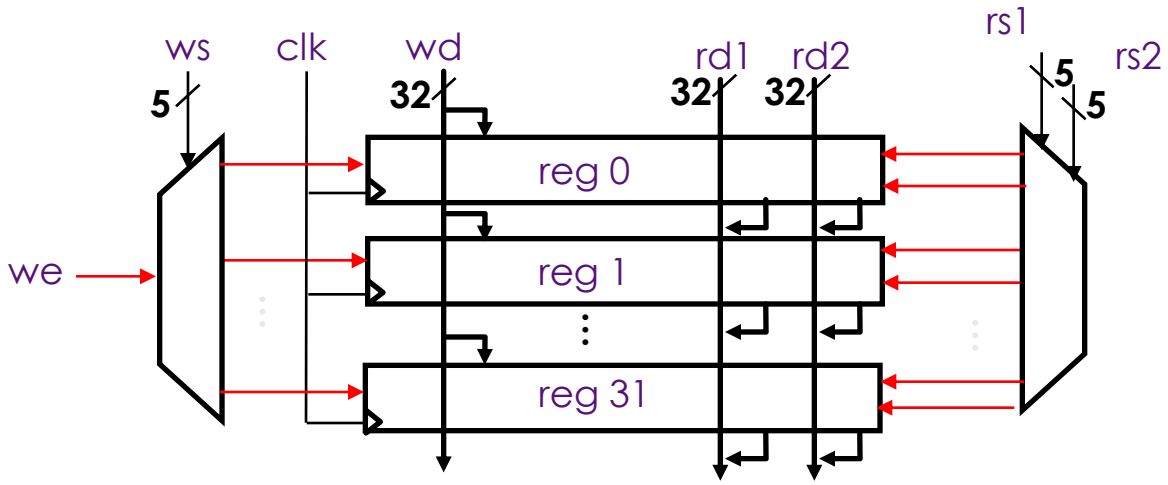
Register



Register File
2R+1W



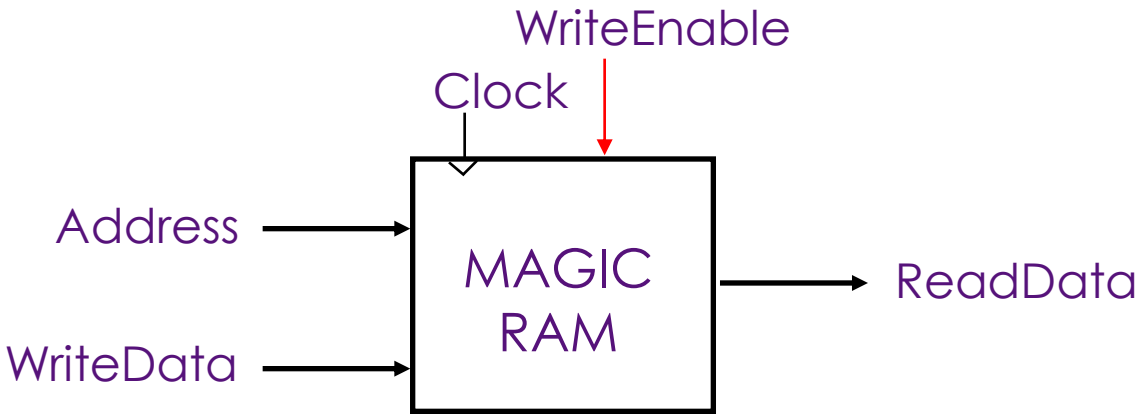
Register Files



Highly ported register files difficult to design

- Almost all MIPS instructions have exactly 2 register source operands
- Intel's Itanium, GPR File has 128 registers with 8 read ports, 4 write ports!!!

Memory – Simple Model



Reads, Writes complete in one cycle

- Read can be done any time (i.e. combinational)
- Write is performed at the rising clock edge if it is enabled
- Write address and data must be stable at the clock edge

MIPS Instruction Set Architecture

Processor State

- 32 32-bit GPRs, R0 always contains a 0
- 32 single precision FPRs, may also be viewed as16 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

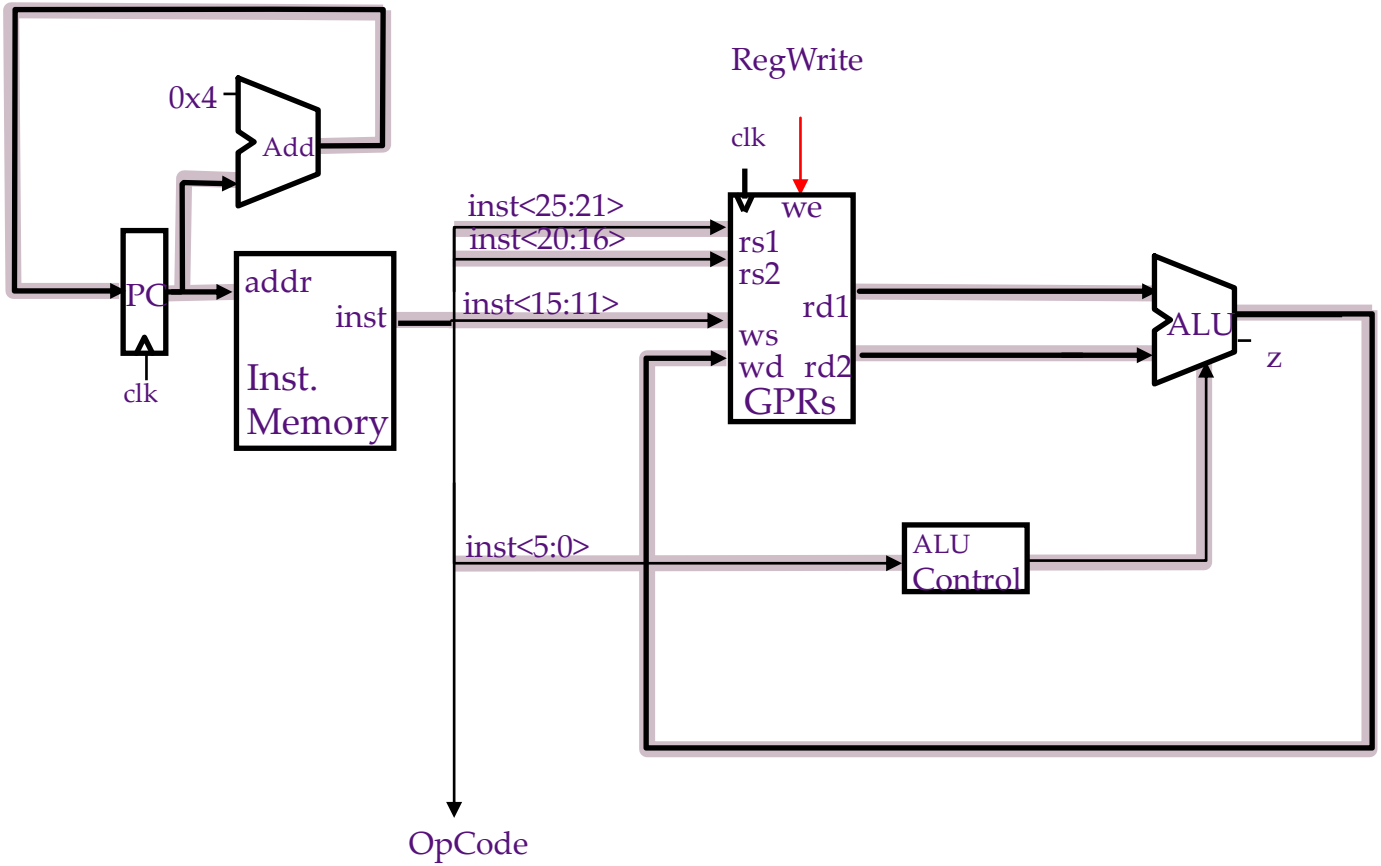
All instructions are 32 bits

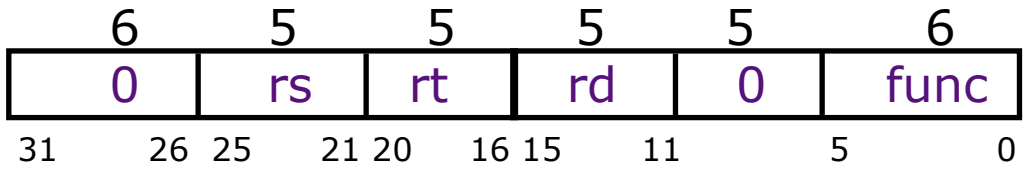
Instruction Execution

1. Instruction Fetch
2. Decode and register access
3. ALU operation
4. Memory operation (optional)
5. Write back

And the computation of the address of the next instruction

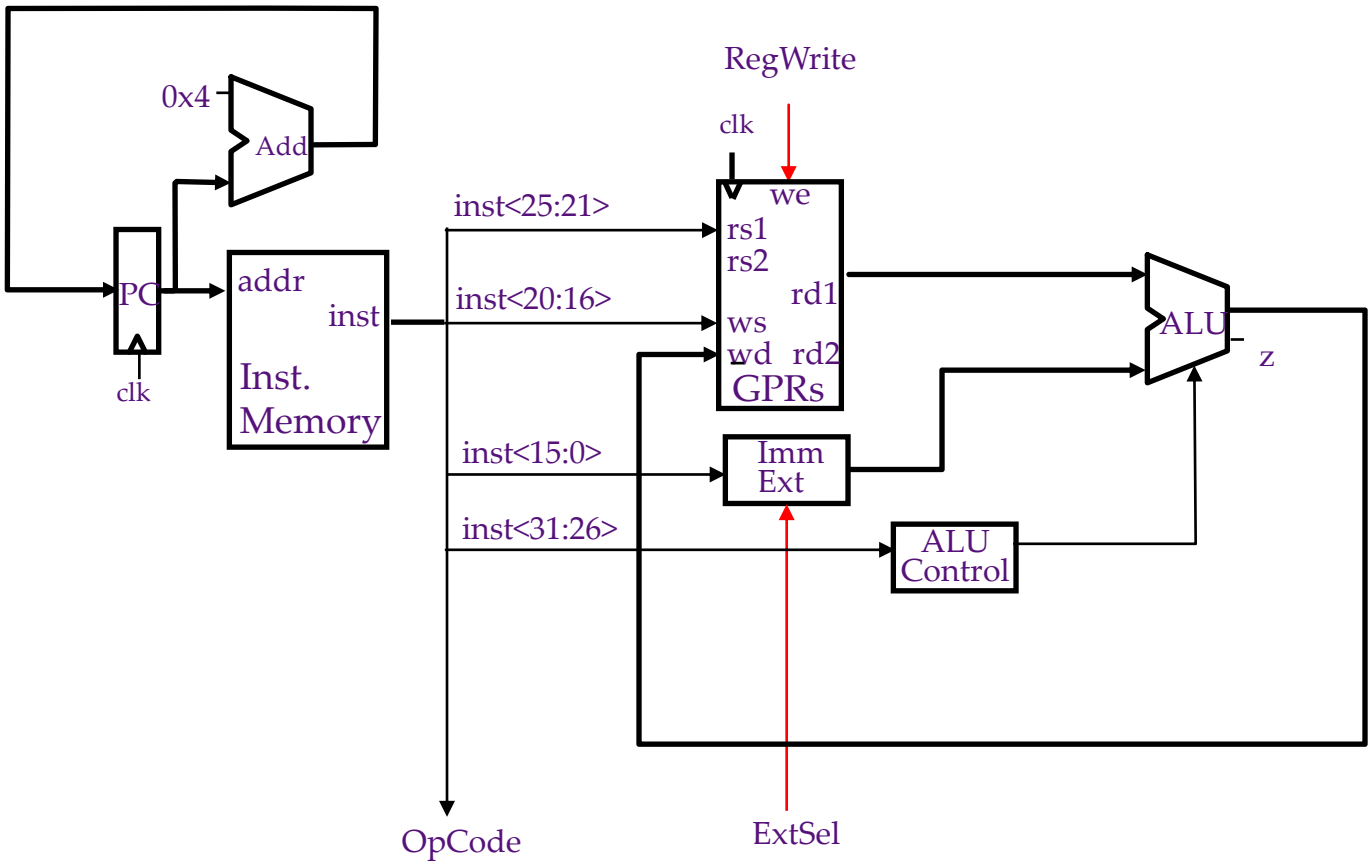
Reg-Reg ALU Instructions

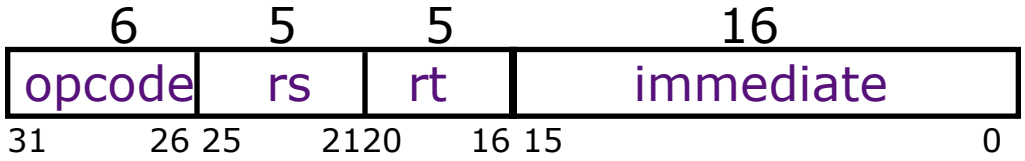




rd ← (rs) func (rt)

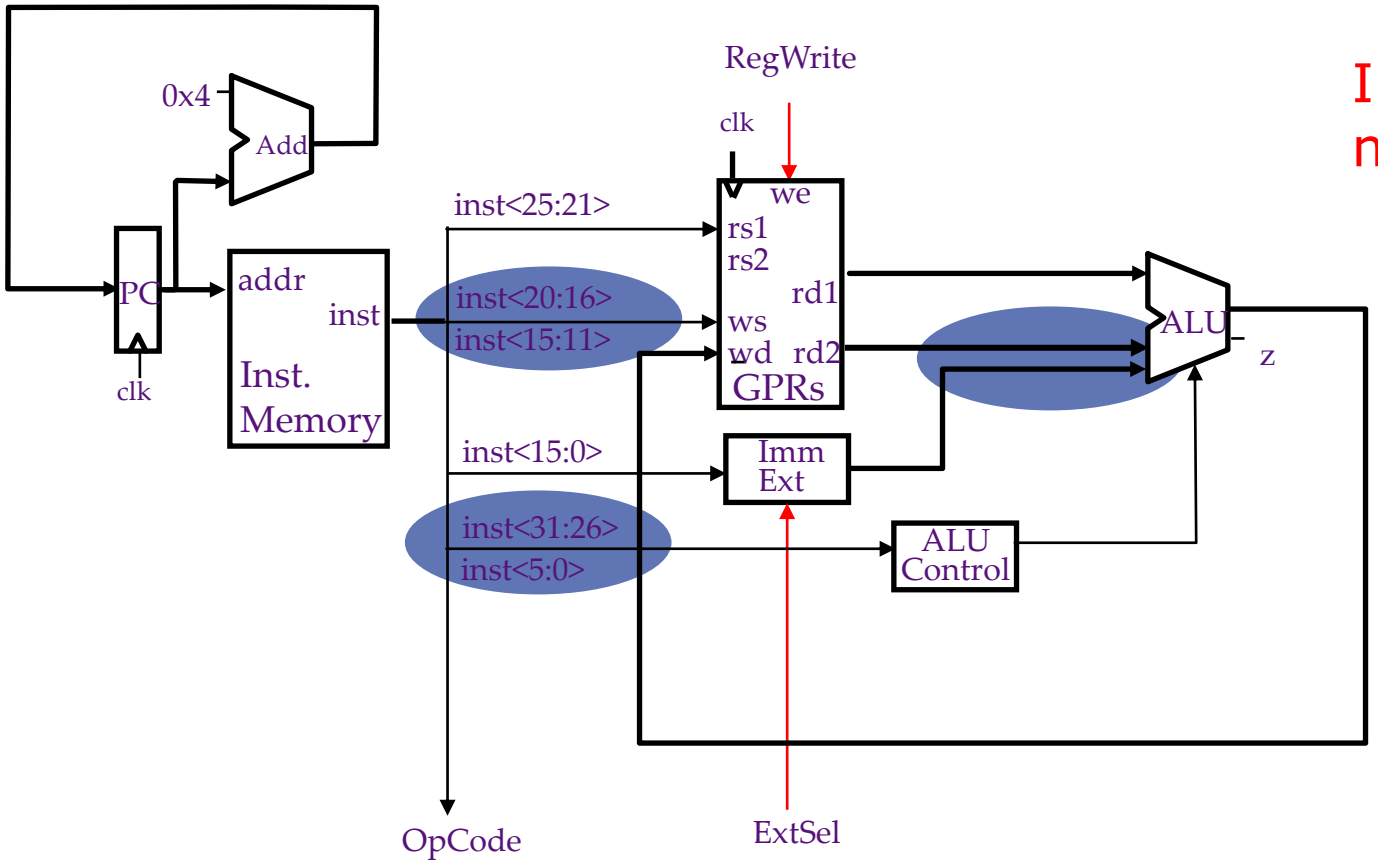
Reg-Imm ALU Instructions



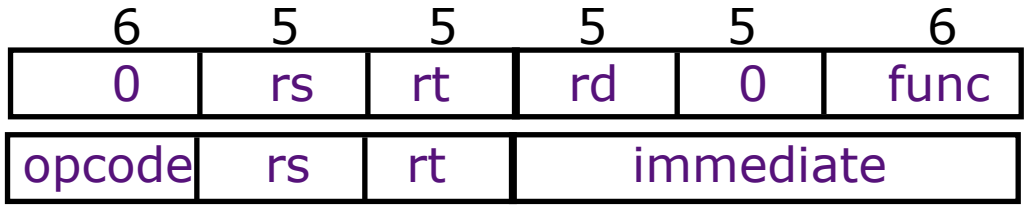


rt ← (rs) op immediate

Conflicts in Merging Datapath



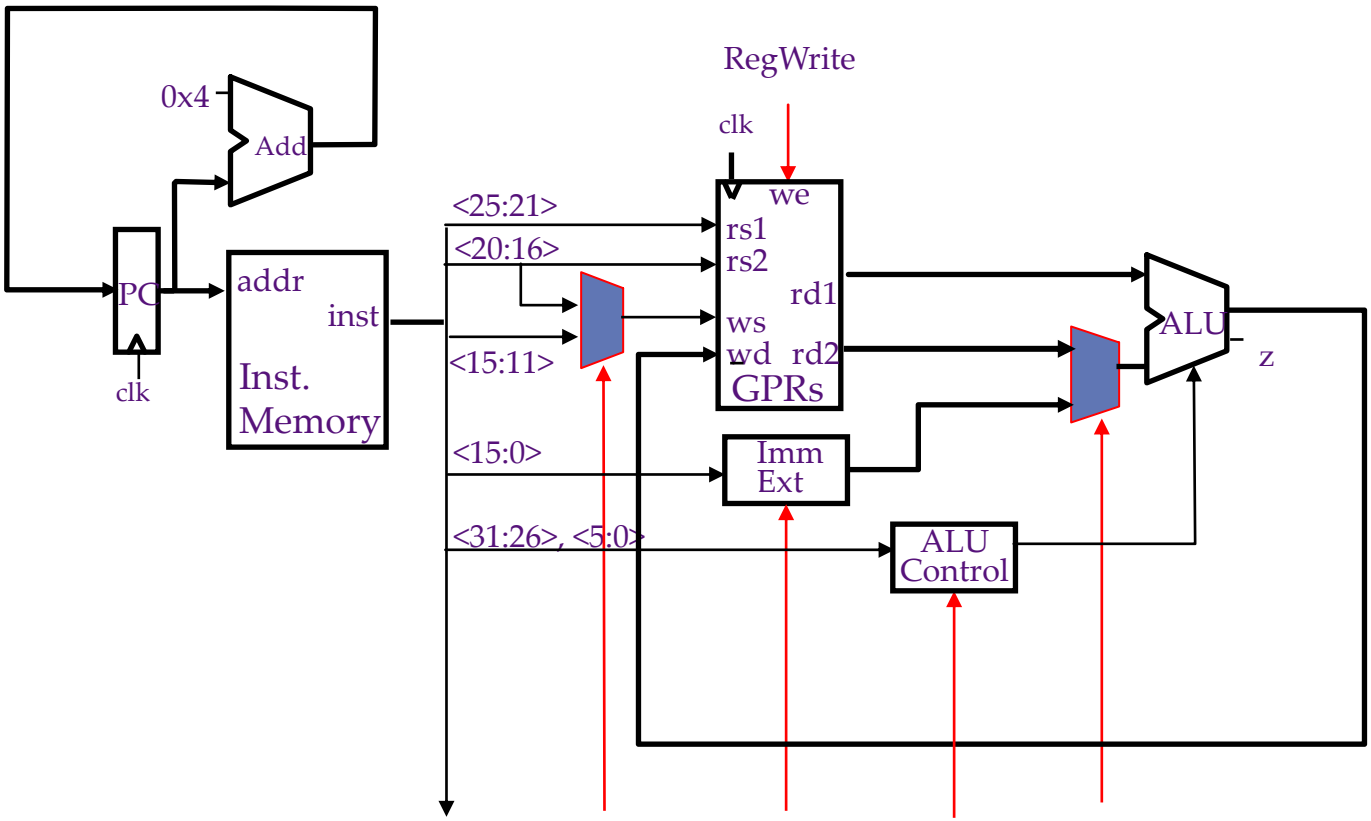
Introduce muxes

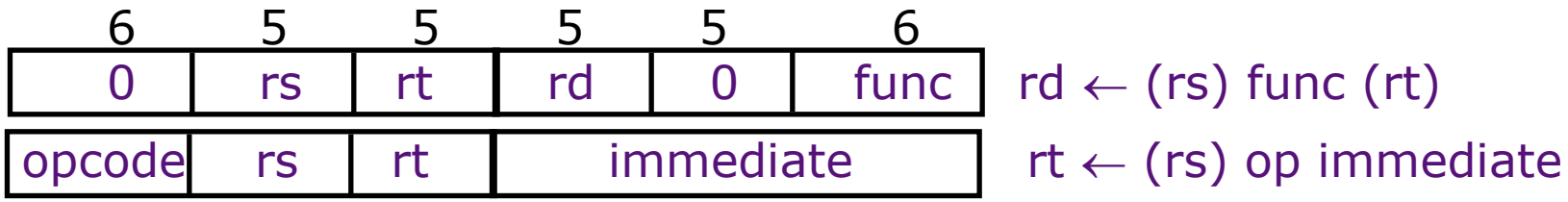


rd ← (rs) func (rt)

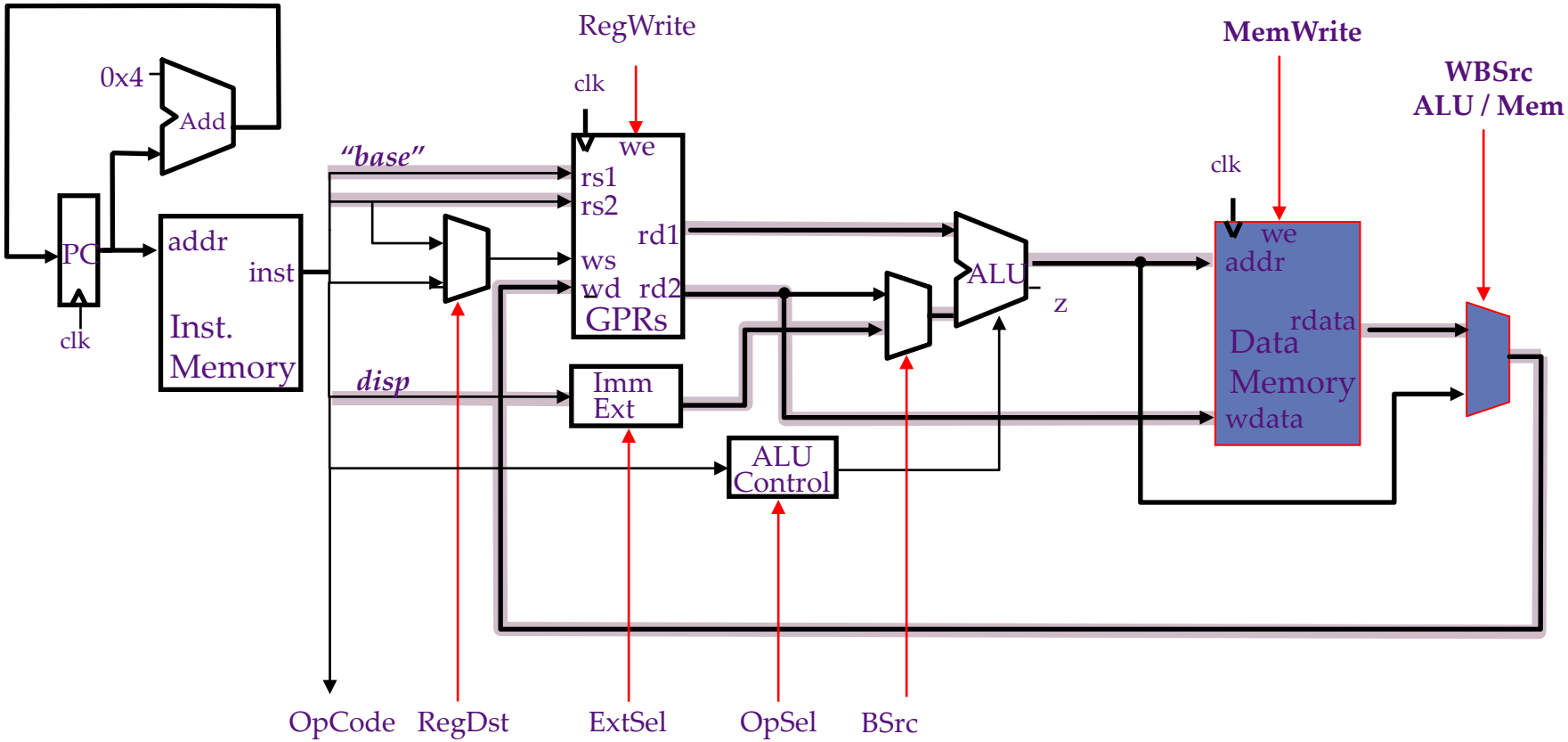
rt ← (rs) op immediate

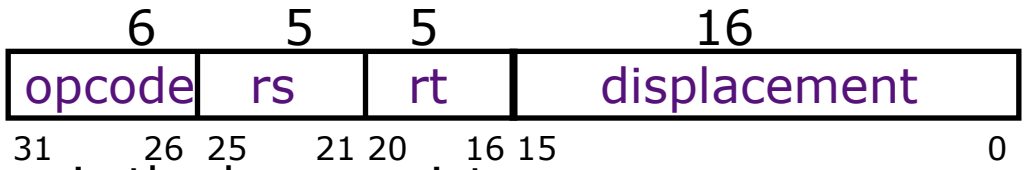
ALU Instructions





Load/Store Instructions (Harvard)





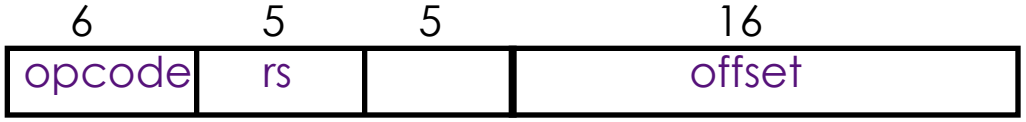
addressing mode
(rs) + displacement

rs is the base register

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

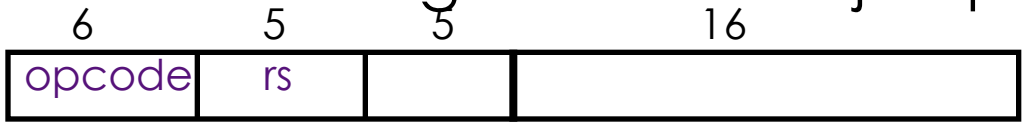
MIPS Control Instructions

Conditional PC-relative branch



BEQZ, BNEZ

Unconditional register-indirect jumps



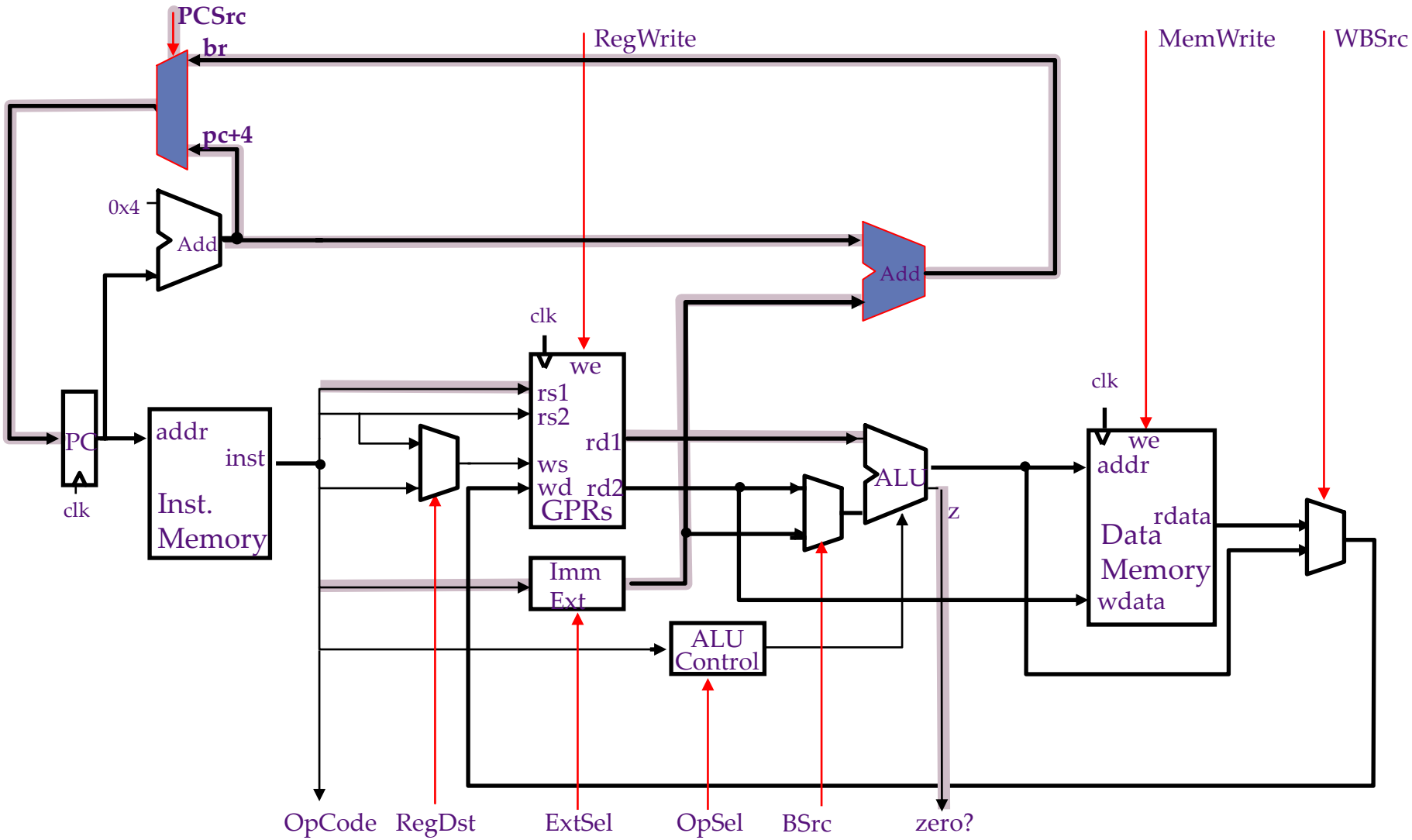
JR, JALR

Unconditional absolute jumps

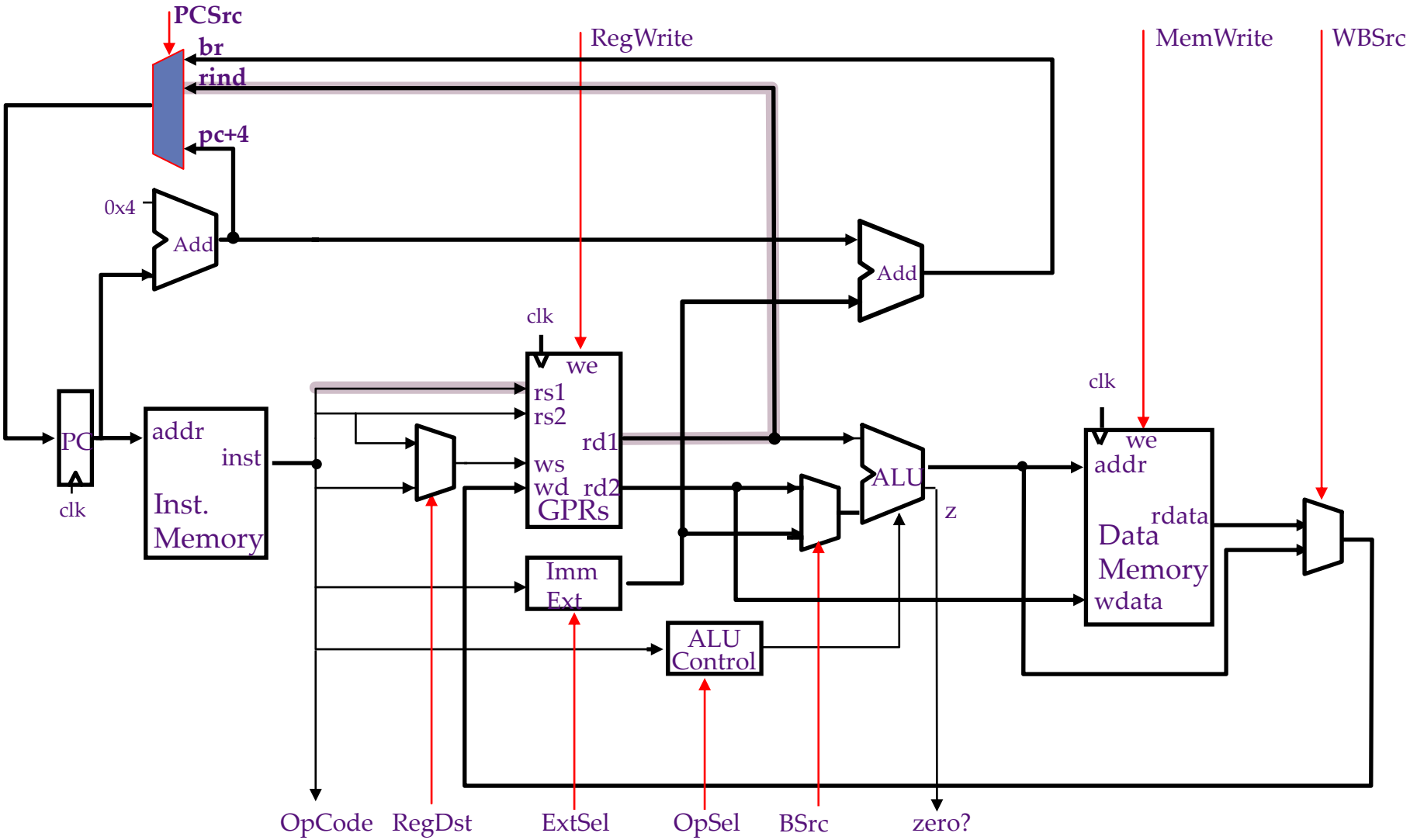
J, JAL

- PC-relative Branches (BEQZ, BNEZ)
- PC ← [offset×4] + [PC+4]; Range is ±128 KB
- Absolute Jumps (J)
- PC ← concat(PC<31:28>, [targetx4]); Range is ± 256MB
- Jump-&-link (JAL)
- R31 ← PC+4; PC ← concat(PC<31:28>, [targetx4]); Range is ± 256MB

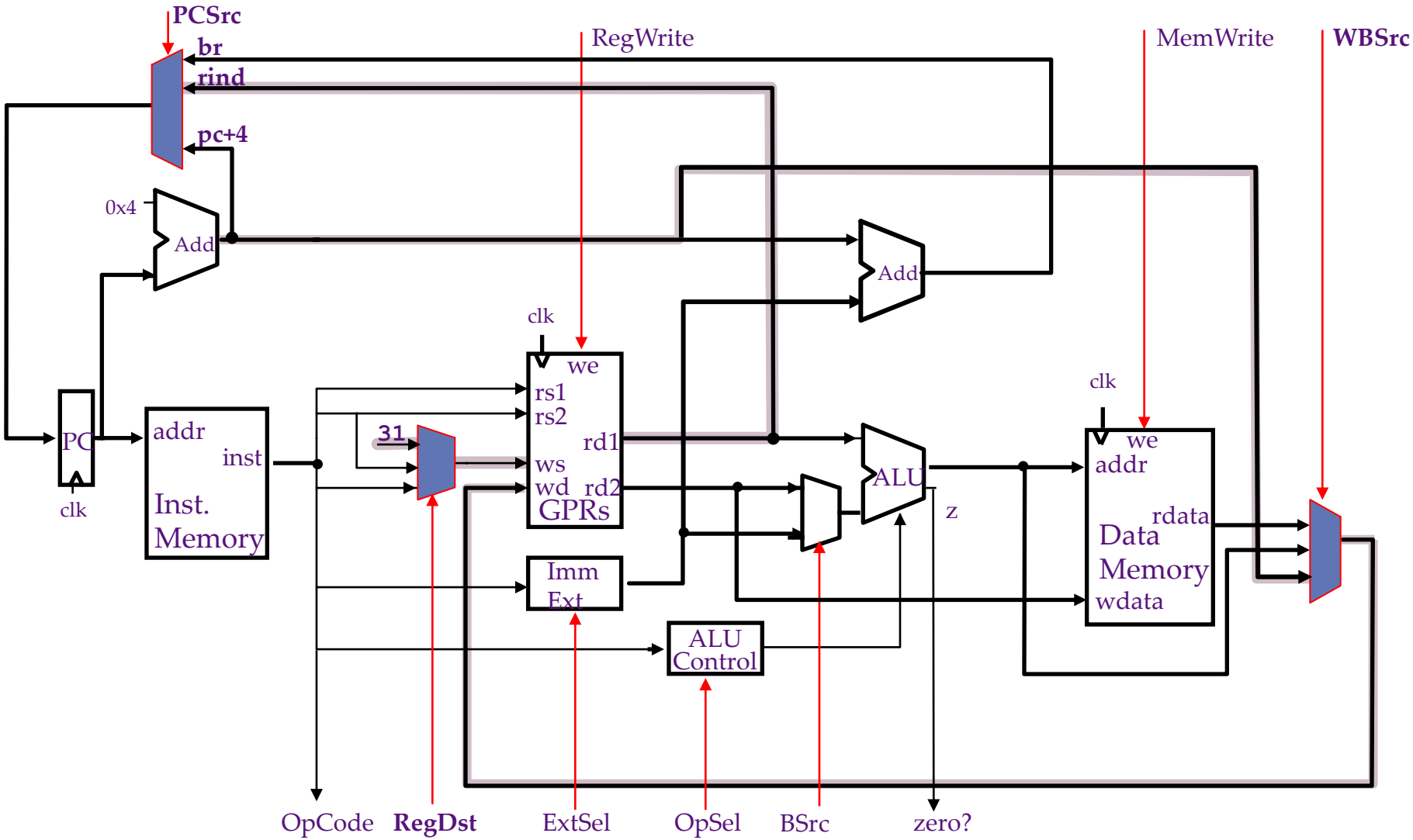
Conditional Branches (BEQZ, BNEZ)



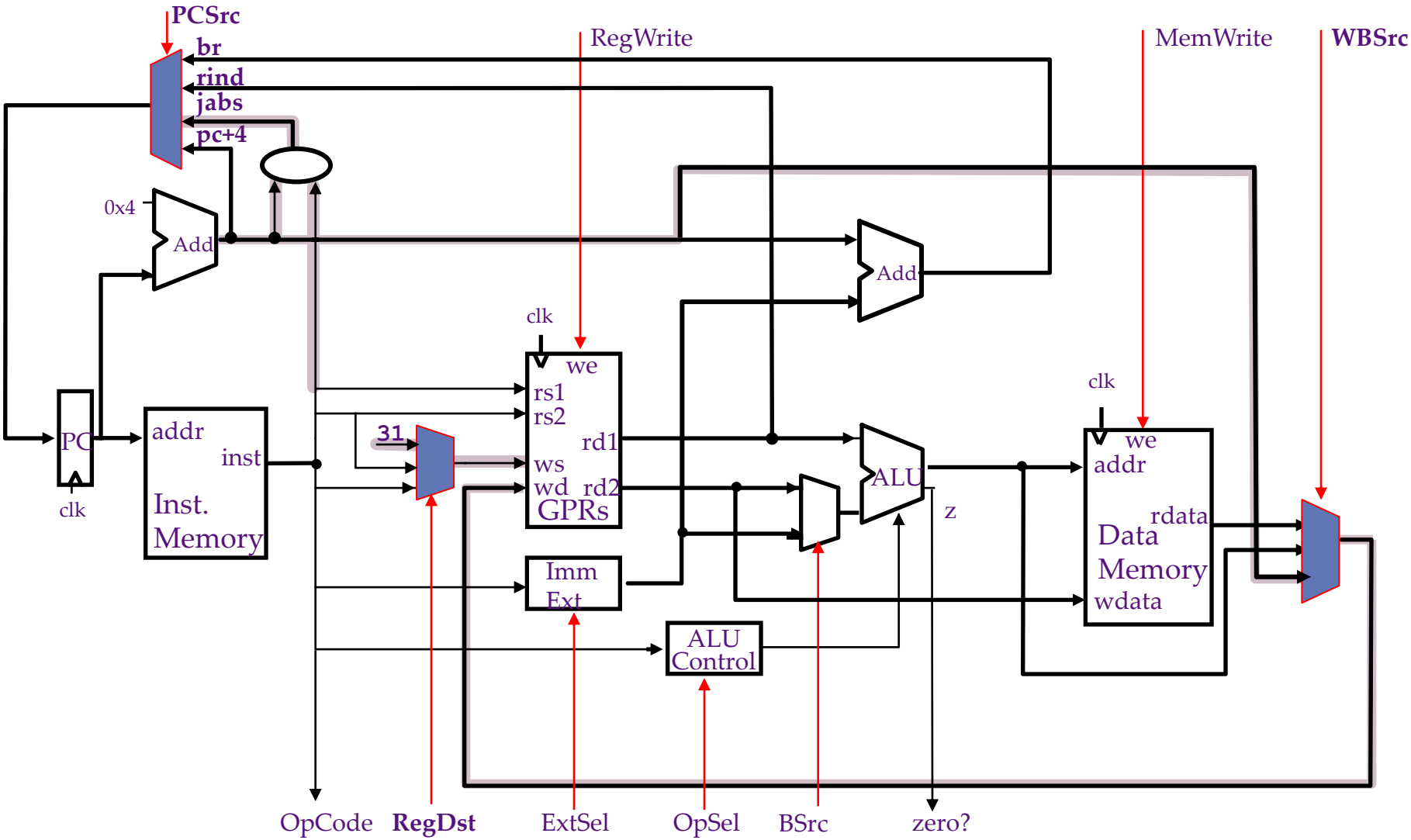
Register-Indirect Jumps (JR)



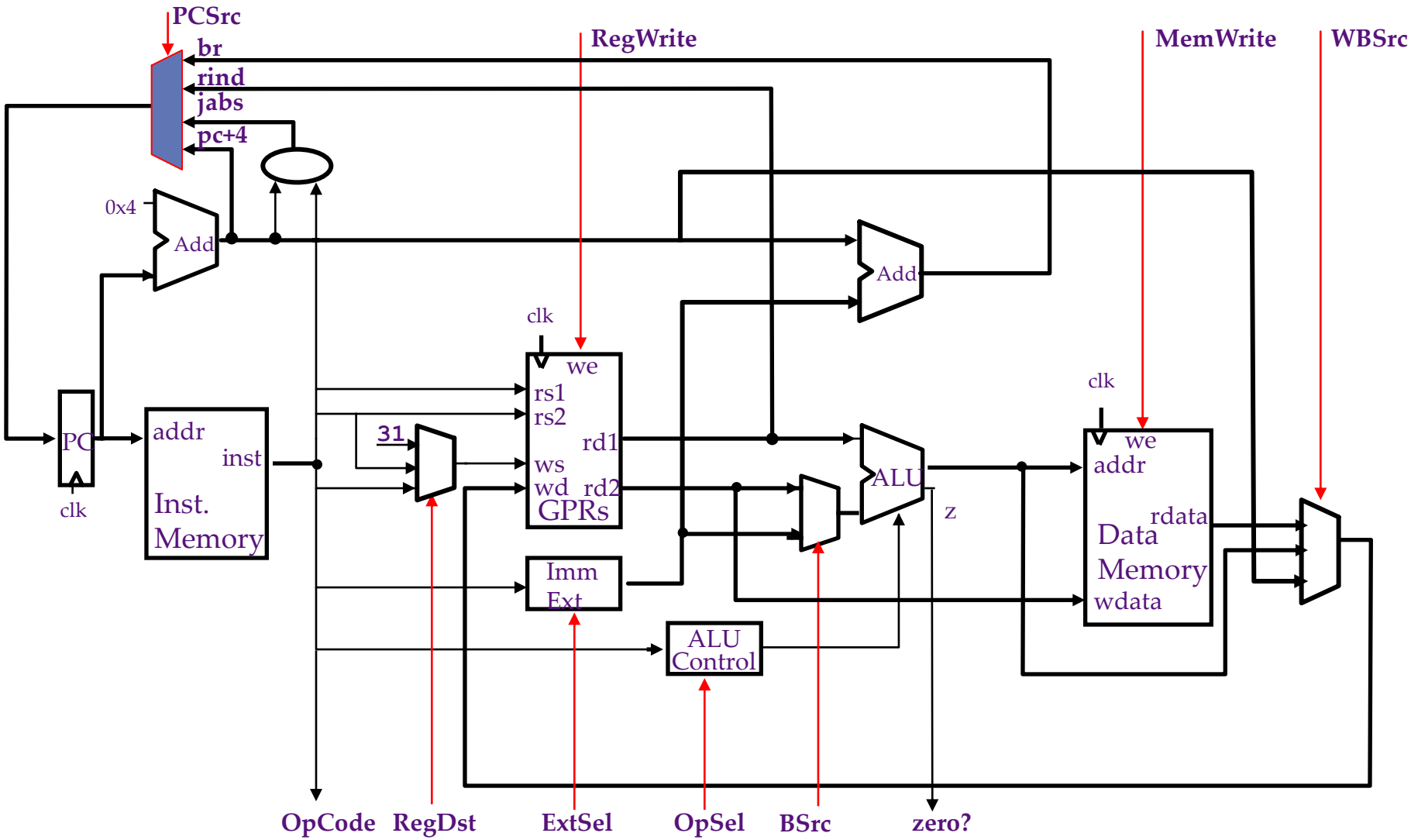
Jump & Link (JALR)



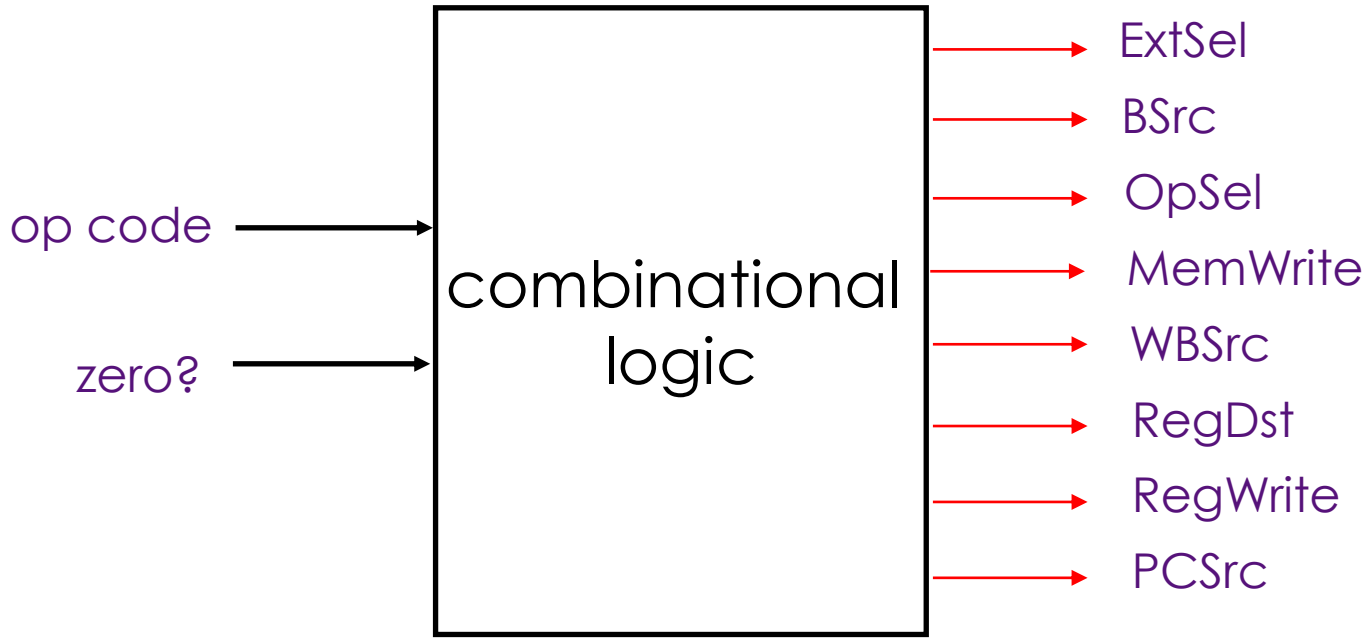
Absolute Jumps (J, JAL)



Harvard Datapath for MIPS



Hardwired Control



Hardware control is pure combinational logic

Hardwired Control Table

Opcode	ExtSel	BSrc	OpSel	MemW	RegW	WBSrc	RegDst	PCSrc
ALU	*	Reg	Func	no	yes	ALU	rd	pc+4
ALUi	$sExt_{16}$	Imm	Op	no	yes	ALU	rt	pc+4
ALUiu	$uExt_{16}$	Imm	Op	no	yes	ALU	rt	pc+4
LW	$sExt_{16}$	Imm	+	no	yes	Mem	rt	pc+4
SW	$sExt_{16}$	Imm	+	yes	no	*	*	pc+4
$BEQZ_{z=0}$	$sExt_{16}$	*	0?	no	no	*	*	br
$BEQZ_{z=1}$	$sExt_{16}$	*	0?	no	no	*	*	pc+4
J	*	*	*	no	no	*	*	jabs
JAL	*	*	*	no	yes	PC	R31	jabs
JR	*	*	*	no	no	*	*	rind
JALR	*	*	*	no	yes	PC	R31	rind

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

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

Harvard Control for MIPS

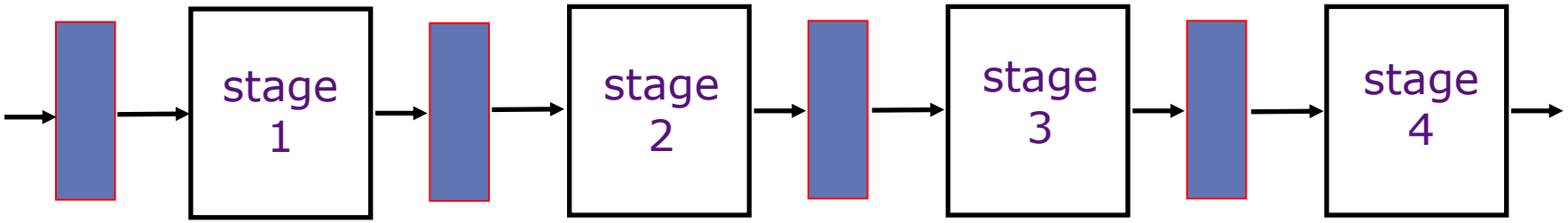
Assumptions

- Clock period is sufficiently long to complete:
  1. instruction fetch
  2. decode and register access
  3. ALU operation
  4. data fetch if required
  5. register write-back setup time
- $t_C > t_{IFetch} + t_{RFetch} + t_{ALU} + t_{DMem} + t_{RWB}$

Update at end of Cycle

- Updates occur on rising edge of following clock
- Update architectural state
- Program counter (PC), register file, memory

An Ideal Pipeline



All objects go through the same stages

No sharing of resources between any two stages

Propagation delay through all pipeline stages is equal

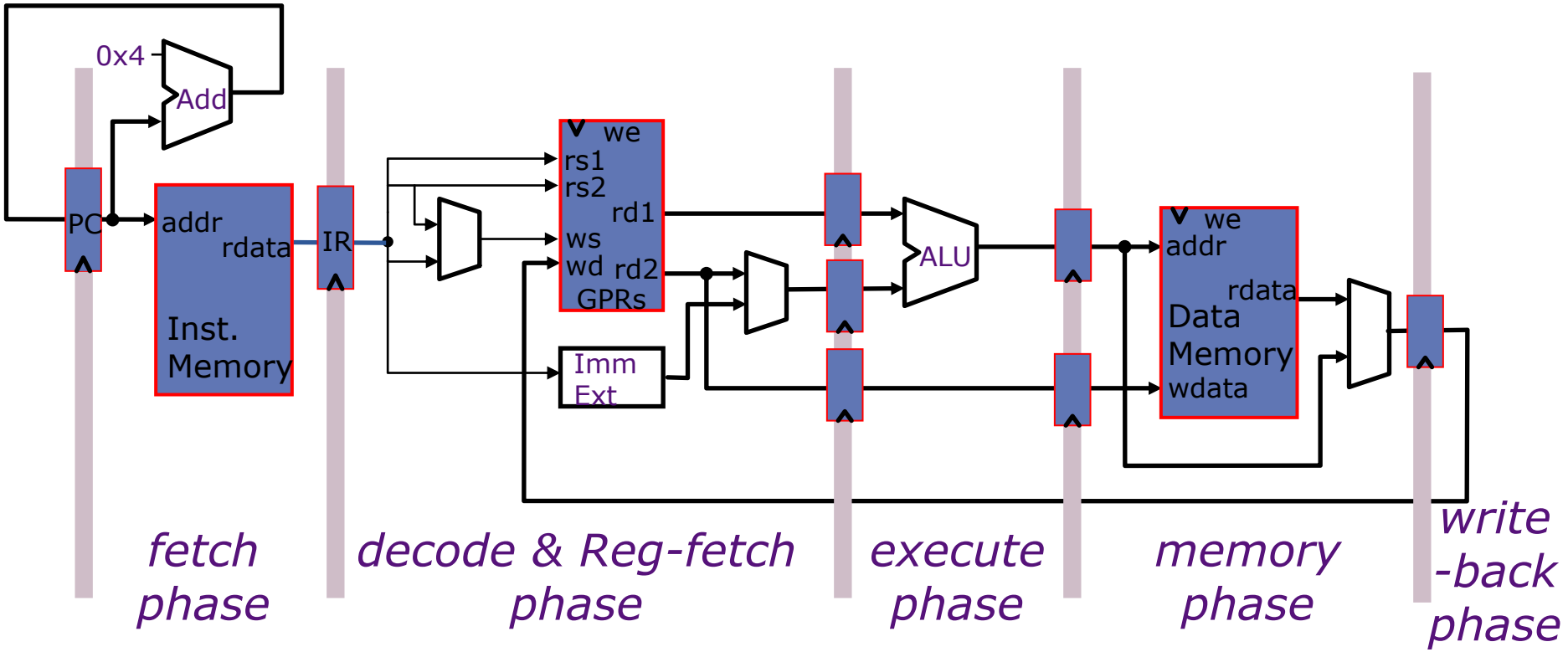
Scheduling of an object entering the pipeline is not affected by objects in other stages

These conditions hold for industrial assembly lines but do they hold for an instruction pipeline?

Pipelining for MIPS

Strategy

- First, build MIPS without pipelining, CPI = 1
- Then, add pipeline registers to reduce cycle time, maintaining CPI=1
- Clock period reduced by dividing the execution of an instruction into multiple cycles, t_C > max {t_{IM}, t_{RF}, t_{ALU}, t_{DM}, t_{RW}} (= t_{DM} *probably*)
- However, CPI will increase unless instructions are pipelined



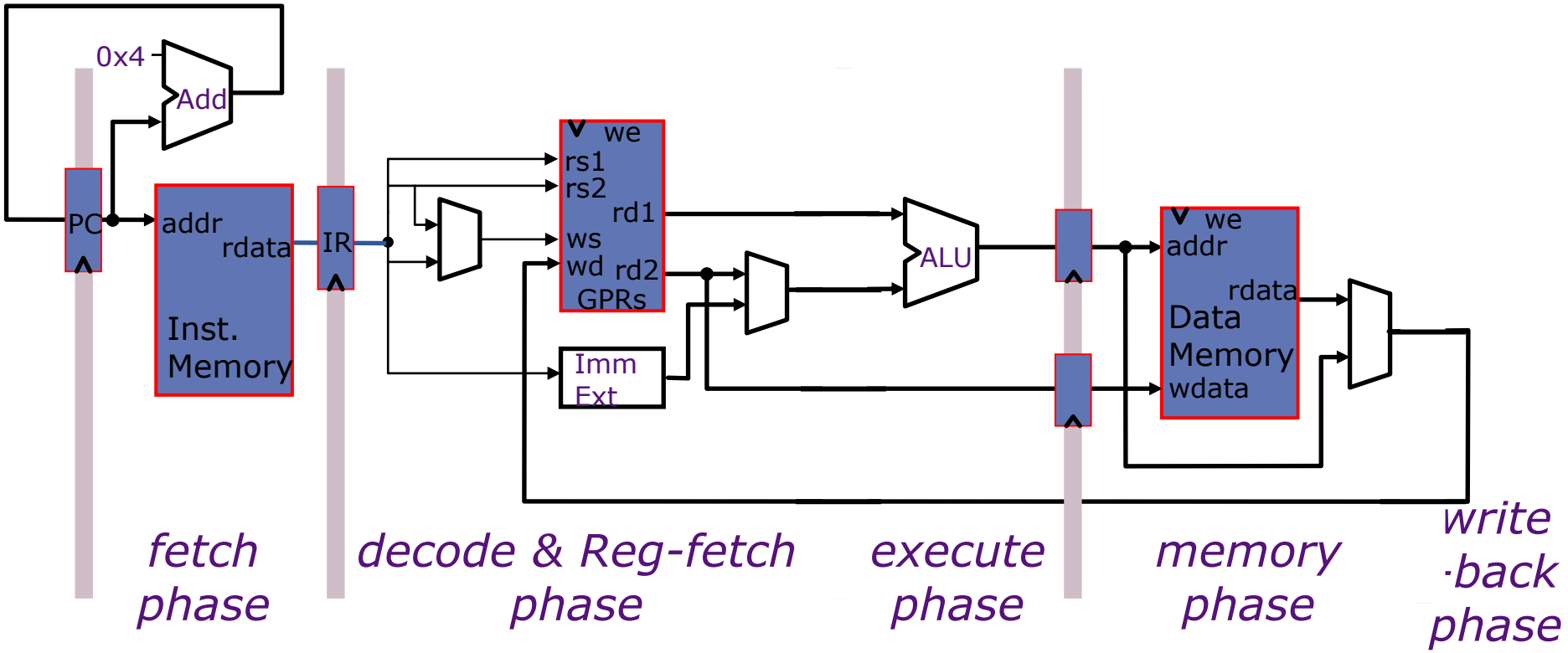
Dividing Datapath into Stages

Suppose memory is slower than other stages.

Since slowest stage determines the clock, it may be possible to combine stages without loss of performance

t_{IM} = 10 units
t_{DM} = 10 units
t_{ALU} = 5 units
t_{RF} = 1 unit
t_{RW} = 1 unit

Pipelining Example



- Write-back requires much less time, combine with memory access
- $t_C > \max\{t_{IM}, t_{RF} + t_{ALU}, t_{DM} + t_{RW}\} = t_{DM} + t_{RW}$

Pipelining Speedup

Assumptions	Unpipelined t_C	Unpipelined t_C	Speedup
$t_{IM} = t_{DM}$ = 10; t_{ALU} = 5, $t_{RF} = t_{RW}$ = 1 4-stage pipeline	27	10	2.7
$t_{IM} = t_{DM} = t_{ALU} = t_{RF} = t_{RW}$ = 5 4-stage pipeline	25	10	2.5
$t_{IM} = t_{DM} = t_{ALU} = t_{RF} = t_{RW}$ = 5 5-stage pipeline	25	5	5.0

Higher speedup possible w/ more pipeline stages

Summary

Microcoding became less attractive as gap between RAM and ROM speeds

Complex instruction sets difficult to pipeline, so it was difficult to increase performance as gate count grew

Processor performance depends on (a) instructions per program, (b) cycles per instruction and (c) time per cycle

Load/Store RISC instruction sets designed for efficient, pipelined implementations

Acknowledgements

These slides contain material developed and copyright by

- Arvind (MIT)
- Krste Asanovic (MIT/UCB)
- Joel Emer (Intel/MIT)
- James Hoe (CMU)
- John Kubiatowicz (UCB)
- Alvin Lebeck (Duke)
- David Patterson (UCB)
- Daniel Sorin (Duke)