دانشگاه کردستان
University of Kurdistan
زانکۆی کوردستان

Department of Computer Engineering
University of Kurdistan

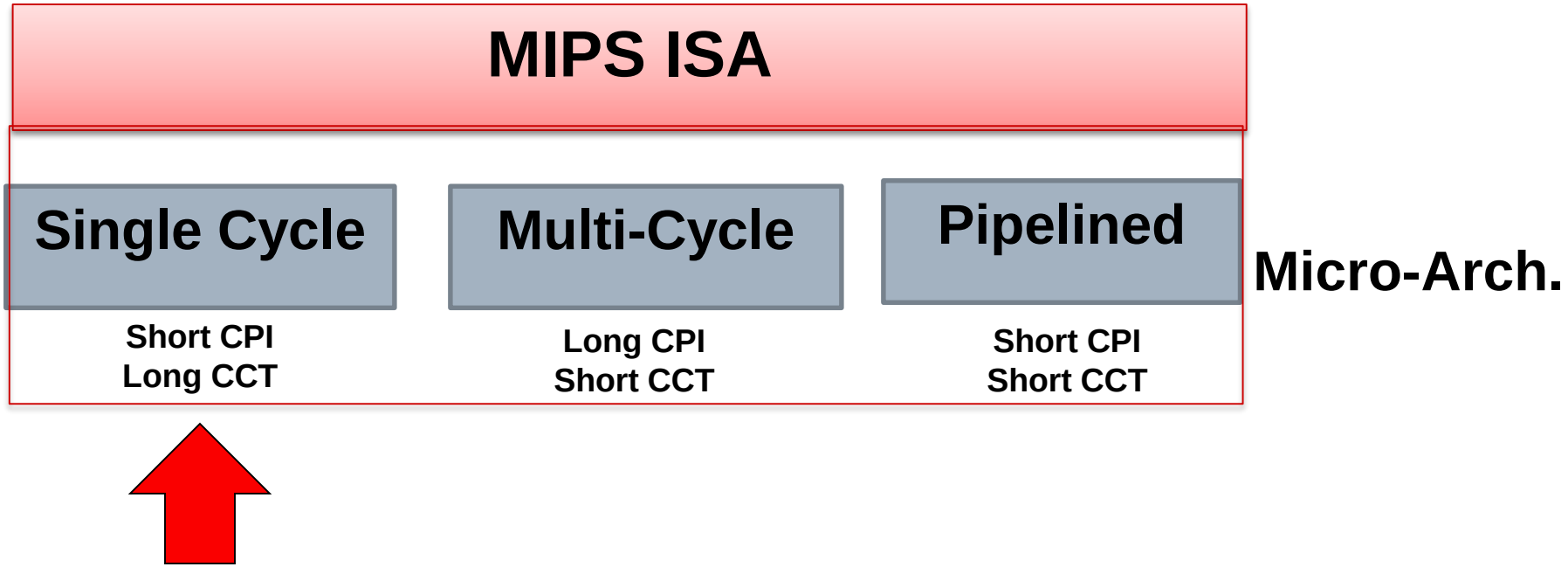
Computer Architecture

MIPS Datapath and Control (Single Cycle)

By: Dr. Alireza Abdollahpouri

A single-cycle MIPS processor

Any instruction set can be implemented in many different ways

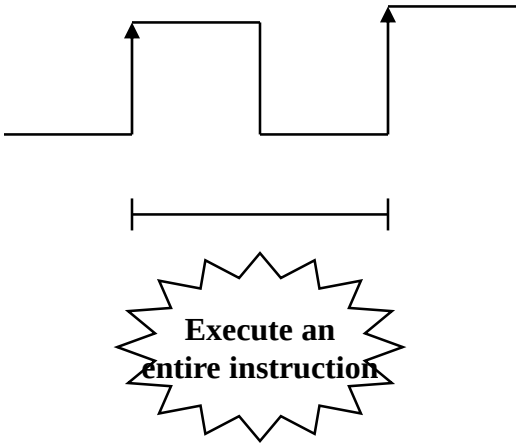


All instructions will execute in the same amount of time; this will determine the clock cycle time for our performance equations.

در این روش پیاده سازی، تمامی دستورالعملها برای اجرا به زمان یکسانی احتیاج دارند (یک پالس ساعت). پس طول کلاک باید برابر با طولانیترین دستورالعمل باشد.

The Performance Big Picture

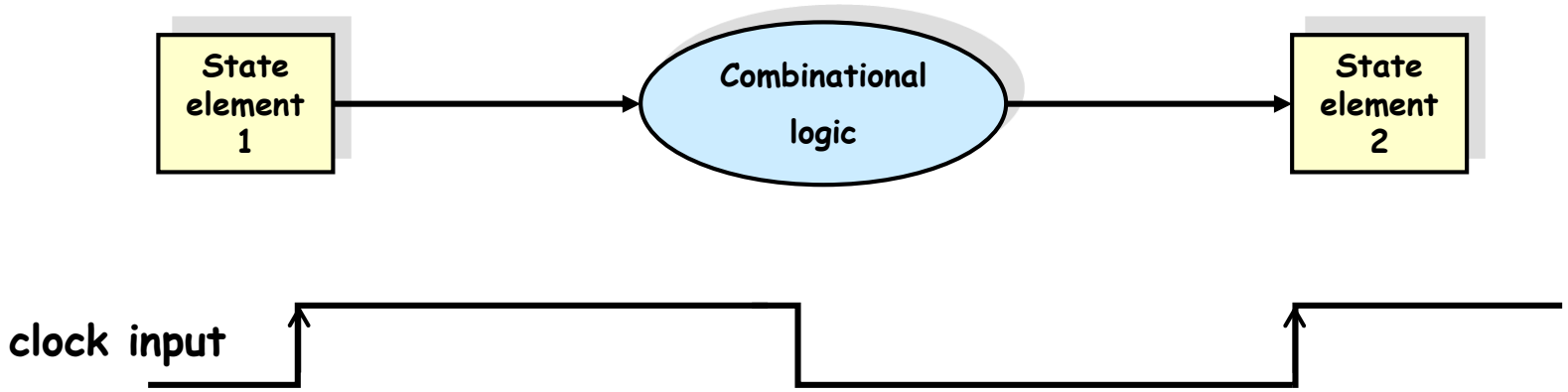
- Execution Time = IC * CPI * Cycle Time
- Instruction count is controlled by the ISA and the compiler design
- Processor design (Datapath and control) will determine:
  - Clock cycle time
  - Clock cycles per instruction
- Starting today:
  - Single cycle processor:
    - Advantage: CPI = 1
    - Disadvantage: long cycle time



روش اعمال کلاک

- Needed to prevent simultaneous read/write to state elements
- Edge-triggered methodology:

  state elements updated at rising clock edge



Data Path & Control Unit

- **Data path** مسیری است که مشخص میکند داده ها چگونه بین پردازنده و سایر المانهای اصلی رد و بدل میشود. اجزای آن عبارتند از:
  - combinational elements
  - state (sequential) elements
- **Control Unit** مشخص میکند که سیگنالهای کنترلی و زمانبندی چگونه به المانهای Data path میرسد.

روند اجرای دستورات در MIPS

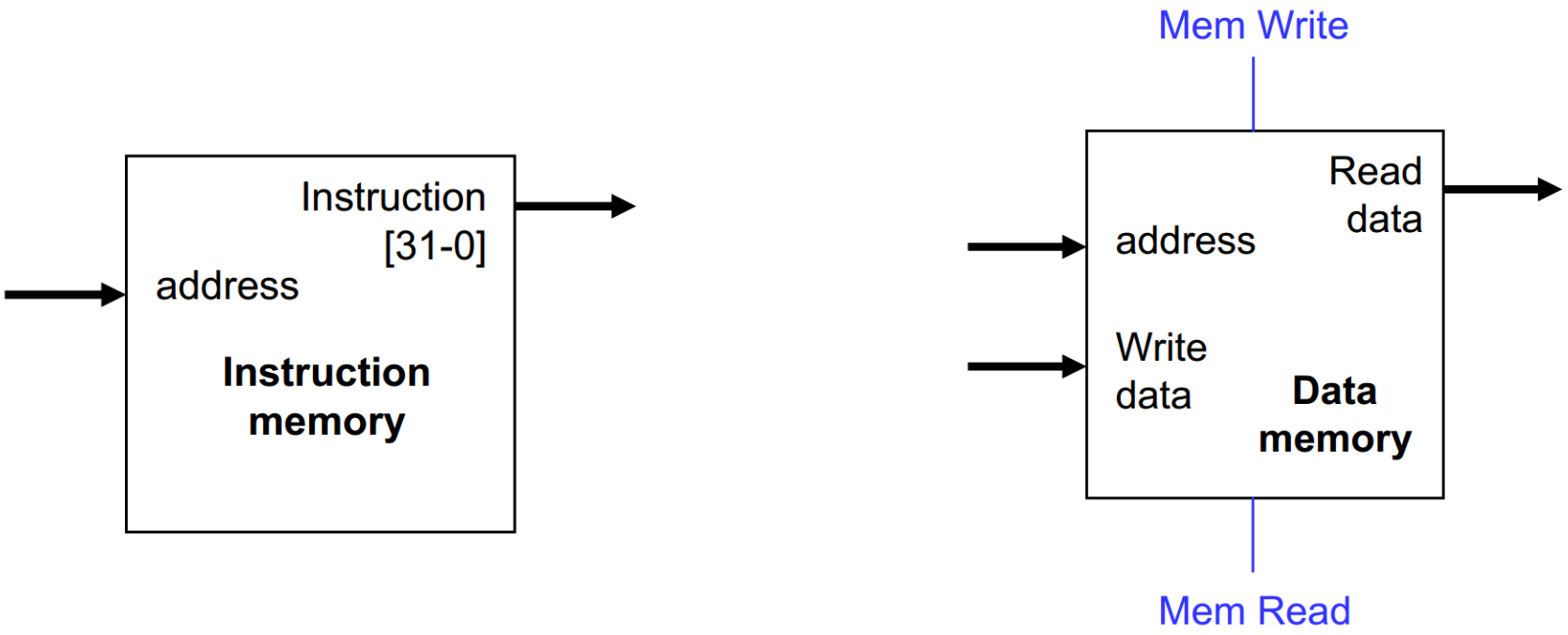
- واکشی دستورالعمل از حافظه
- کدبرداری (دیکد کردن) دستورالعمل
- خواندن عملوندها
- اجرای دستورالعمل (عمل ALU)
- نوشتن (یا خواندن) برای دستورات LW و SW
- ثبت نتیجه در رجیستر مقصد (در صورت لزوم)

در پیاده سازی **Single-cycle** تمامی این مراحل در یک کلاک انجام میشوند

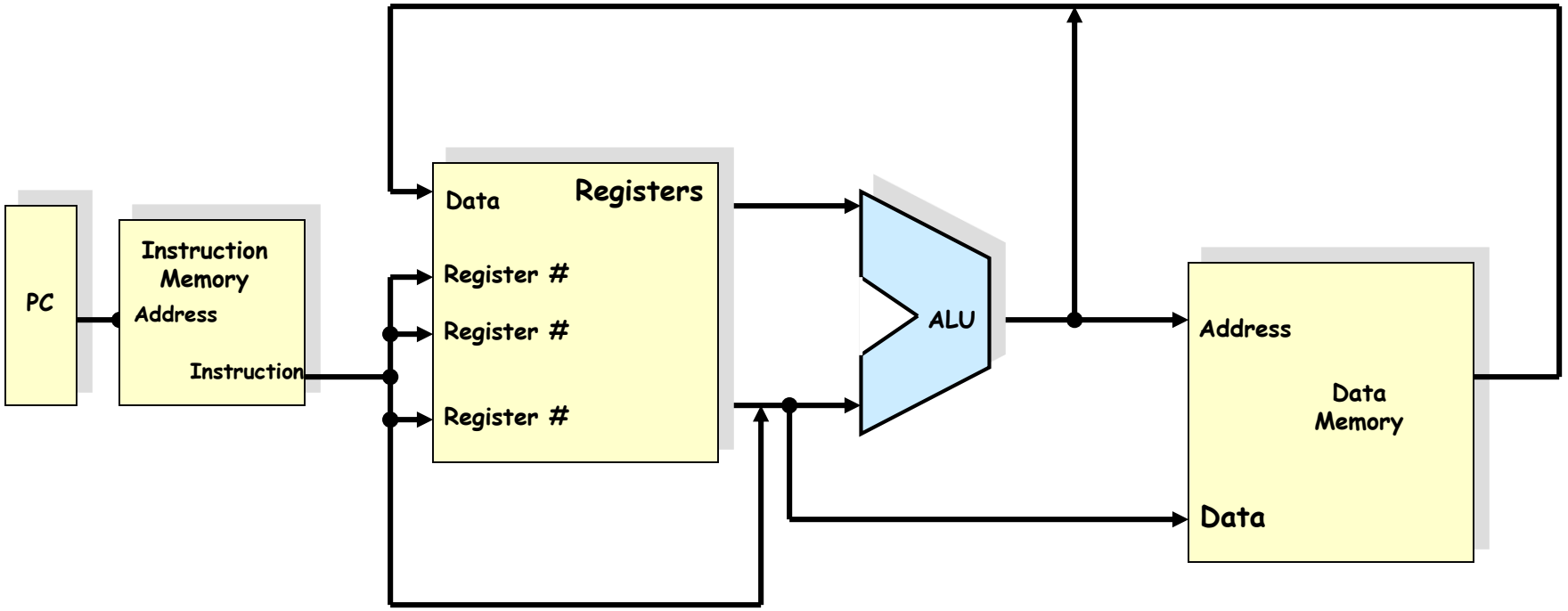
Instruction and Data Memory

برای شروع از معماری **Harward** استفاده میکنیم (دو حافظه جدا)

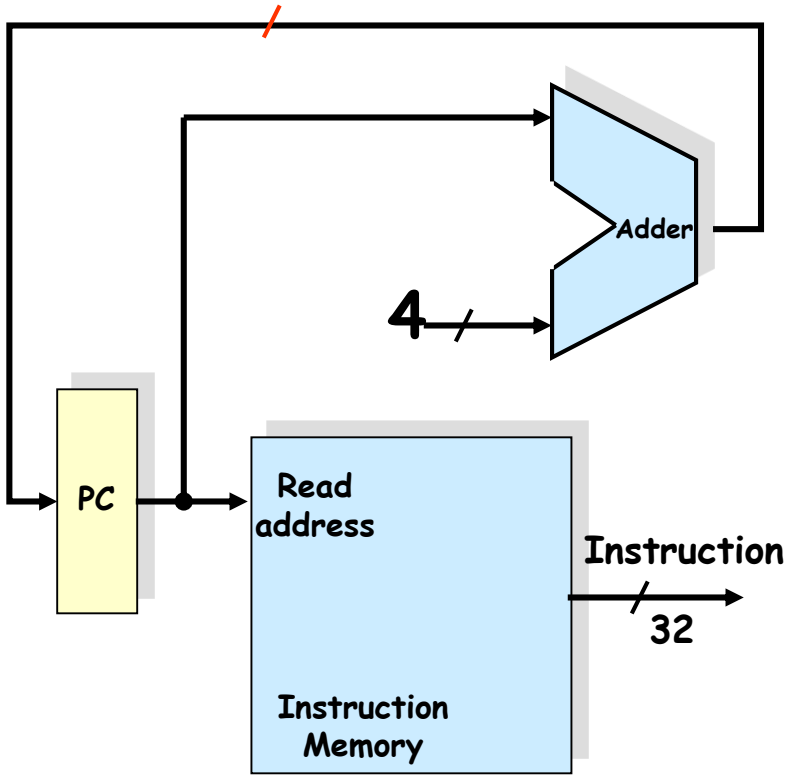
There will be only one DRAM Main memory
But we can have separate SRAM caches (We'll study caches later)



Datapath Schematic (Simplified)



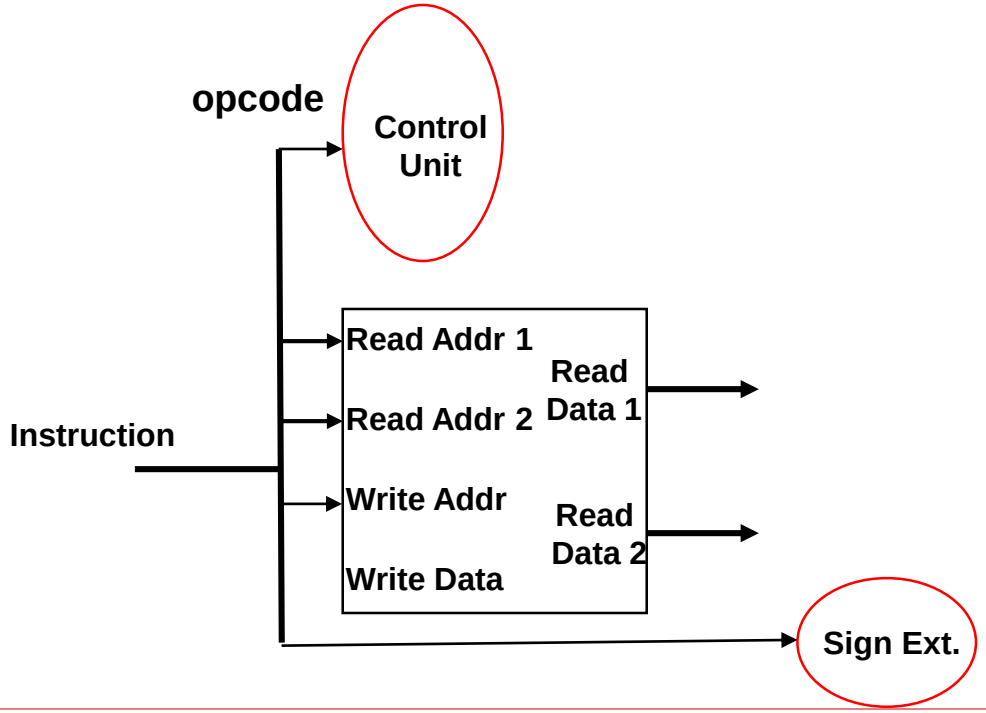
Instruction Fetch



- محتوی PC توسط یک جمع کننده با ۴ جمع میشود تا آدرس دستور بعدی محاسبه شود.
- مقدار PC به حافظه دستورالعمل داده میشود تا دستور واکشی شده و به سایر اجزای Data Path ارسال شود.
- اگر دستور بعدی پرش باشد، مقدار PCمتفاوت خواهد بود (بعدا صحبت خواهد شد)

Instruction Decode

- باید مقدار opcode به واحد کنترل فرستاده شود.
- سایر فیلدهای دستور به قسمتهای مختلف (رجیستر فایل یا توسعه علامت) داده میشوند و رجیسترهای لازم هم از رجیستر فایل خوانده میشوند.



MIPS Subset of Instructions

- Only a subset of the MIPS instructions are considered
  - ALU instructions (R-type): **add, sub, and, or, xor, slt**
  - Immediate instructions (I-type): **addi, slti, andi, ori, xori**
  - Load and Store (I-type): **lw, sw**
  - Branch (I-type): **beq, bne**
  - Jump (J-type): **j**
- This subset does not include all the integer instructions
- But sufficient to illustrate design of datapath and control

Details of the MIPS Subset

Instruction		Meaning	Format					
add	rd, rs, rt	addition	$op^6 = 0$	rs^5	rt^5	rd^5	0	0x20
sub	rd, rs, rt	subtraction	$op^6 = 0$	rs^5	rt^5	rd^5	0	0x22
and	rd, rs, rt	bitwise and	$op^6 = 0$	rs^5	rt^5	rd^5	0	0x24
or	rd, rs, rt	bitwise or	$op^6 = 0$	rs^5	rt^5	rd^5	0	0x25
xor	rd, rs, rt	exclusive or	$op^6 = 0$	rs^5	rt^5	rd^5	0	0x26
slt	rd, rs, rt	set on less than	$op^6 = 0$	rs^5	rt^5	rd^5	0	0x2a
addi	rt, rs, im^{16}	add immediate	0x08	rs^5	rt^5	im^{16}		
slti	rt, rs, im^{16}	slt immediate	0x0a	rs^5	rt^5	im^{16}		
andi	rt, rs, im^{16}	and immediate	0x0c	rs^5	rt^5	im^{16}		
ori	rt, rs, im^{16}	or immediate	0x0d	rs^5	rt^5	im^{16}		
xori	rt, im^{16}	xor immediate	0x0e	rs^5	rt^5	im^{16}		
lw	rt, im^{16}(rs)	load word	0x23	rs^5	rt^5	im^{16}		
sw	rt, im^{16}(rs)	store word	0x2b	rs^5	rt^5	im^{16}		
beq	rs, rt, im^{16}	branch if equal	0x04	rs^5	rt^5	im^{16}		
bne	rs, rt, im^{16}	branch not equal	0x05	rs^5	rt^5	im^{16}		
j	im^{26}	jump	0x02	im^{26}				

RTL Description

- RTL gives a meaning to the instructions
- All instructions are fetched from memory at address PC

Instruction	RTL Description	
ADD	Reg(Rd) ← Reg(Rs) + Reg(Rt);	PC ← PC + 4
SUB	Reg(Rd) ← Reg(Rs) – Reg(Rt);	PC ← PC + 4
ORI	Reg(Rt) ← Reg(Rs) | zero_ext(Im16);	PC ← PC + 4
LW	Reg(Rt) ← MEM[Reg(Rs) + sign_ext(Im16)];	PC ← PC + 4
SW	MEM[Reg(Rs) + sign_ext(Im16)] ← Reg(Rt);	PC ← PC + 4
BEQ	if (Reg(Rs) == Reg(Rt))	
	PC ← PC + 4 + 4 × sign_extend(Im16)	
	else PC ← PC + 4	

Instructions are Executed in Steps

- **R-type**

Fetch instruction:	Instruction ← MEM[PC]
Fetch operands:	data1 ← Reg(Rs), data2 ← Reg(Rt)
Execute operation:	ALU_result ← func(data1, data2)
Write ALU result:	Reg(Rd) ← ALU_result
Next PC address:	PC ← PC + 4

- **I-type**

Fetch instruction:	Instruction ← MEM[PC]
Fetch operands:	data1 ← Reg(Rs), data2 ← Extend(imm16)
Execute operation:	ALU_result ← op(data1, data2)
Write ALU result:	Reg(Rt) ← ALU_result
Next PC address:	PC ← PC + 4

- **BEQ**

Fetch instruction:	Instruction ← MEM[PC]
Fetch operands:	data1 ← Reg(Rs), data2 ← Reg(Rt)
Equality:	zero ← subtract(data1, data2)
Branch:	if (zero) PC ← PC + 4 + 4×sign_ext(imm16)
	else PC ← PC + 4

Instruction Execution – cont'd

- **LW**

Fetch instruction:	Instruction ← MEM[PC]
Fetch base register:	base ← Reg(Rs)
Calculate address:	address ← base + sign_extend(imm16)
Read memory:	data ← MEM[address]
Write register Rt:	Reg(Rt) ← data
Next PC address:	PC ← PC + 4

- **SW**

Fetch instruction:	Instruction ← MEM[PC]
Fetch registers:	base ← Reg(Rs), data ← Reg(Rt)
Calculate address:	address ← base + sign_extend(imm16)
Write memory:	MEM[address] ← data
Next PC address:	PC ← PC + 4

- **Jump**

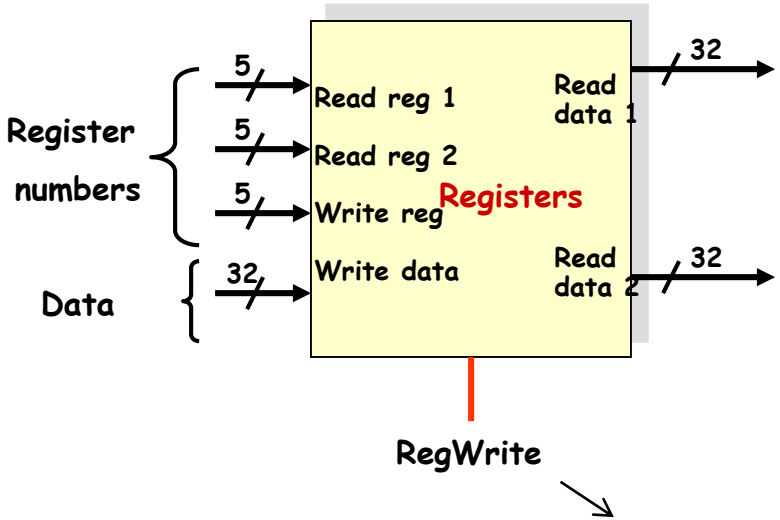
Fetch instruction:	Instruction ← MEM[PC]
Target PC address:	target ← PC[31:28] , Imm26 , '00'
Jump:	PC ← target

concatenation

خواندن عملوندها از رجیستر فایل

- رجیسترهای ۳۲ گانه پردازنده در ساختاری به اسم رجیستر فایل نگهداری میشوند.
- هر یک از رجیسترها را میتوان با مشخص کردن شماره آن خواند و یا نوشت (شماره ۵ بیتی).

- Register File's I/O structure
  - -3 inputs derived from current instruction to specify register operands (2 for read and 1 for write)
  - -1 input to write data into a register
  - -2 outputs carrying contents of the specified registers

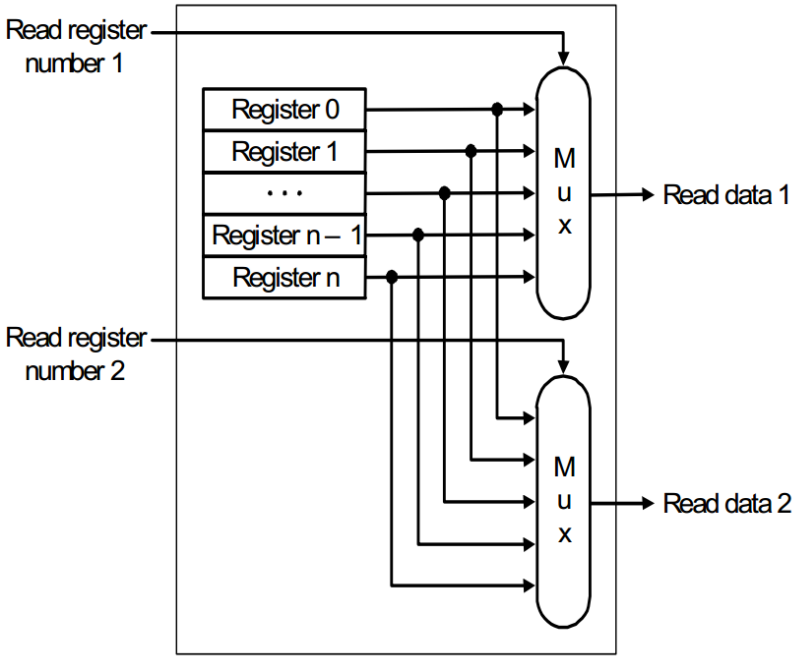


برای کنترل عملیات نوشتن

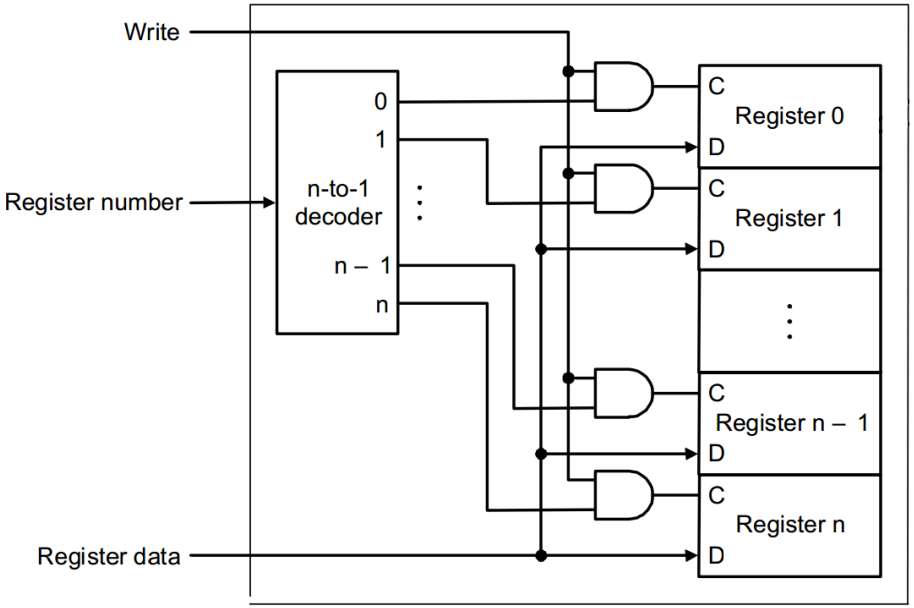
- Register file's outputs are always available on the output lines

Register File

خواندن

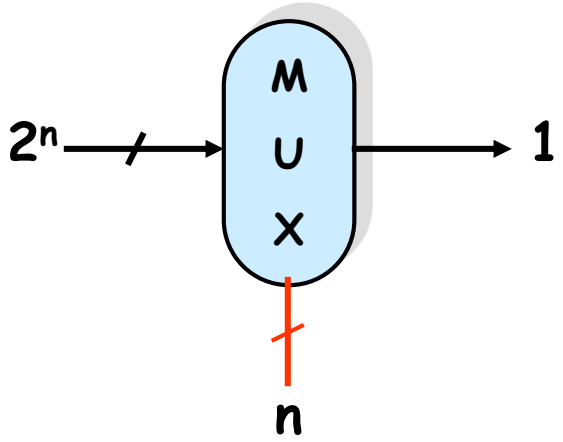


نوشتن

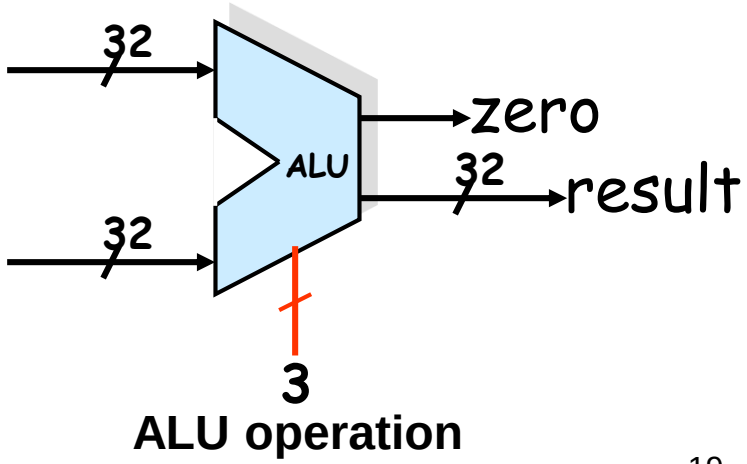


سایر مدارات

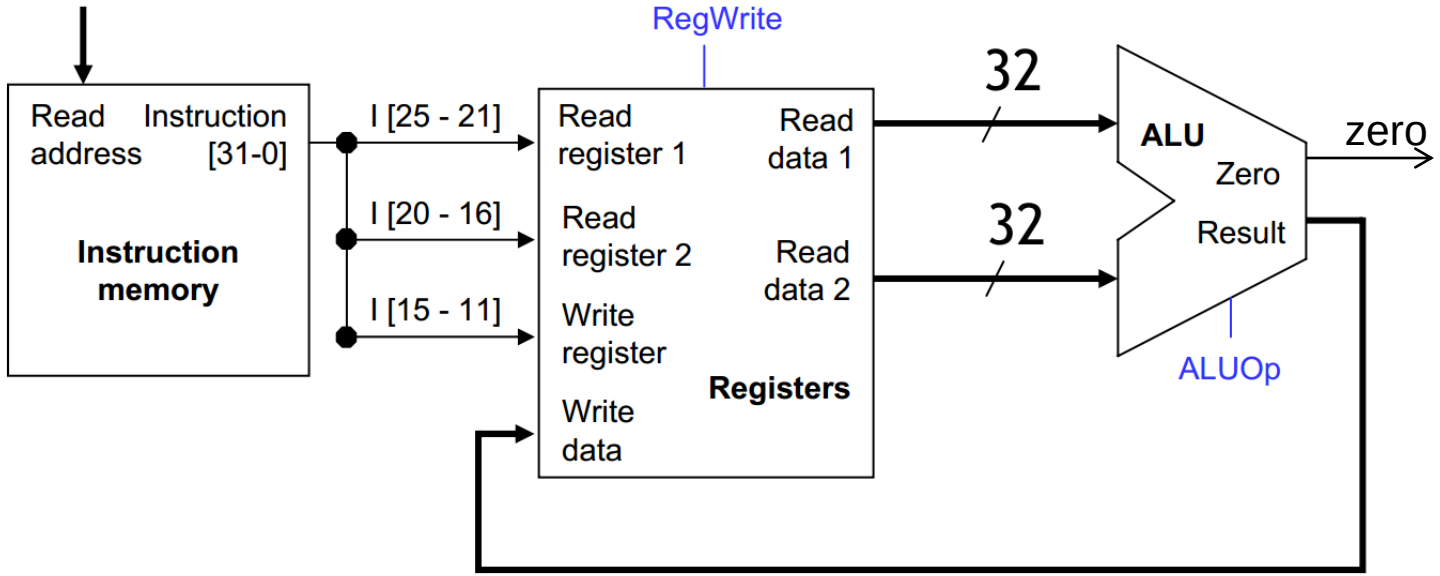
- Multiplexor selects one out of 2^n inputs

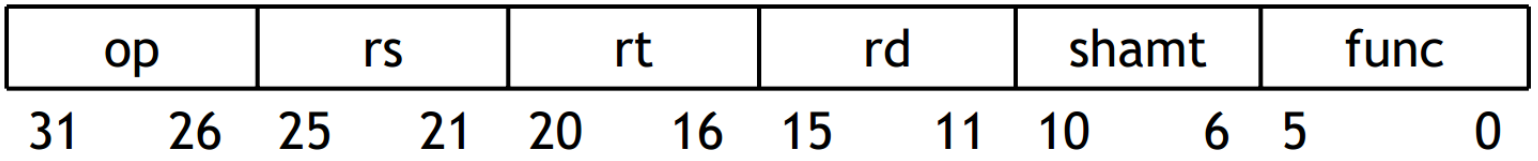


ALU operation	Function
000	and
001	or
010	add
110	sub
111	slt (set less than)
others	don't care



Datapath Building Blocks: R-Type Instruction

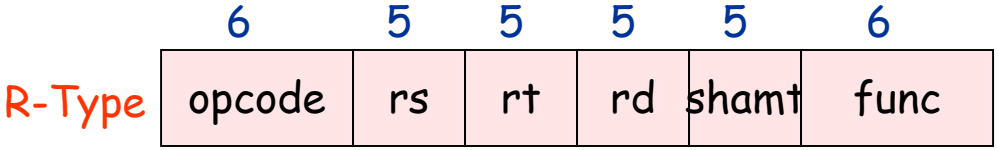




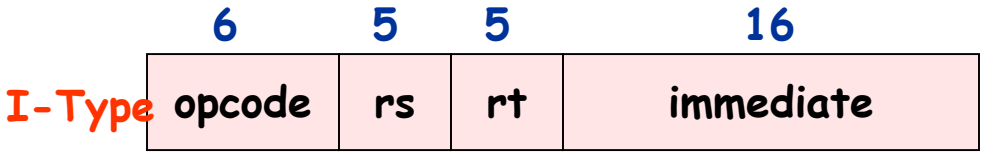
R[rd] ← R[rs] op R[rt]

Datapath Building Blocks: R-Type Instruction

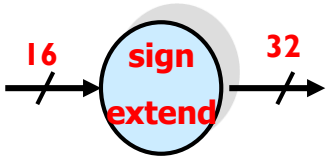
- برای دستورات محاسباتی و منطقی که توسط این فرمت نشان داده میشوند لازم است تا دو رجیستر(rs, rt) از رجیستر فایل خوانده شده و داده آنها به ALU منتقل شود.
- عمل ALU بر اساس نوع دستور تعیین شده و بر روی محتوی رجیسترها انجام میشود.
- نتیجه در رجیستر مقصد (rd) نوشته میشود. (سیگنال RegWrite باید فعال گردد)
- سیگناهای کنترلی باید ایجاد شود تا نتیجه در لبه کلاک در رجیستر مقصد نوشته شود. همچنین سیگنال ALUop باید تولید شود تا عمل ALU را تعیین کند.



I-Type Instruction: load/store




```
LW R2, 232(R1)
SW R5, -88(R4)
```



- برای محاسبه آدرس باید مقدار افست ۱۶ بیتی موجود در دستورالعمل بصورت یک عدد علامت دار ۳۲ بیتی تبدیل شده وبا مقدار پایه موجود در rs جمع شود.

Load Instruction Steps

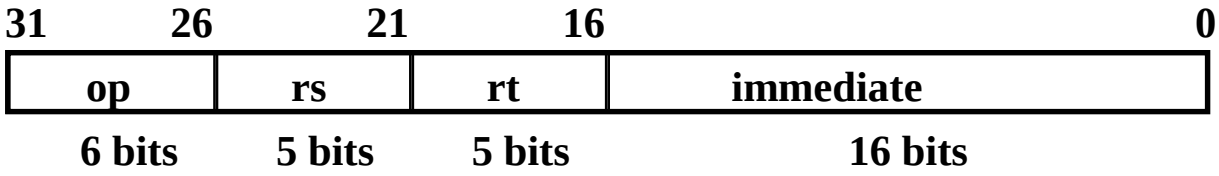
lw $rt, offset($rs)

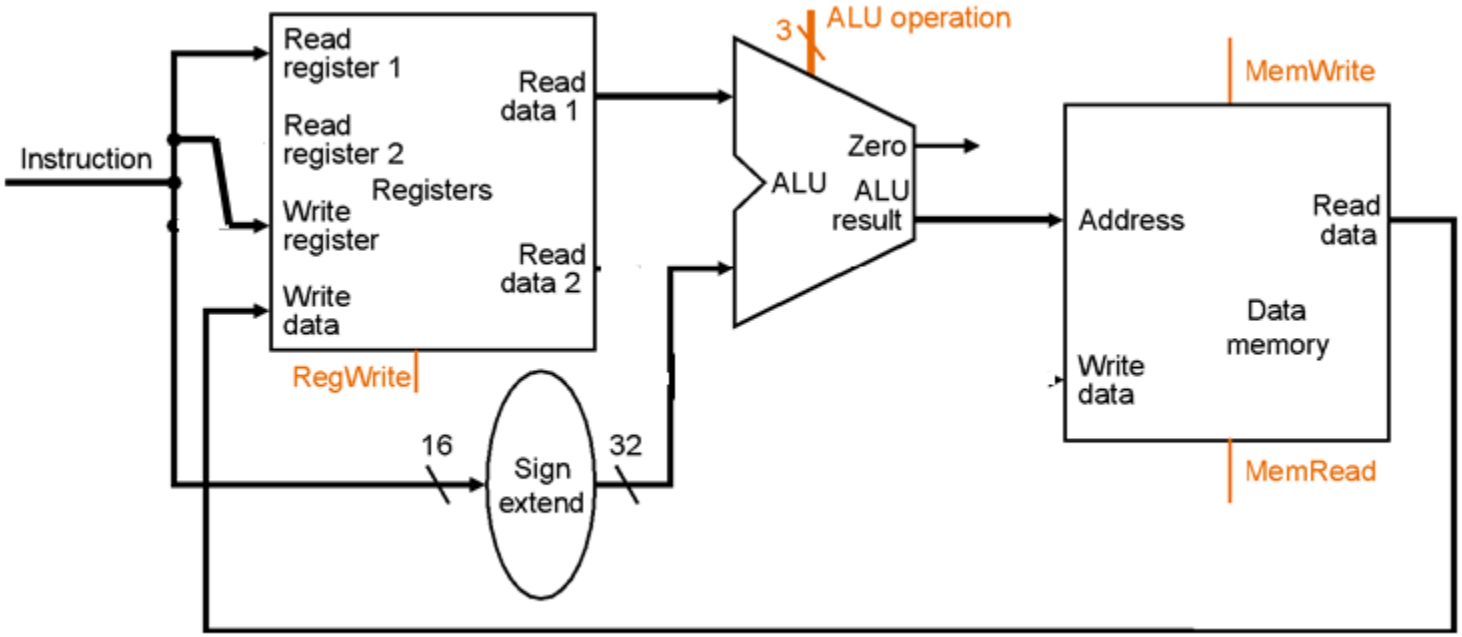
R[rt] ← Mem[R[rs] + SignExt[imm16]]

1. **Fetch instruction and increment PC**
2. **Read base register from the register file: the base register ($rs) is given by bits 25-21 of the instruction**
3. **ALU computes sum of value read from the register file and the sign-extended lower 16 bits (offset) of the instruction**
4. **The sum from the ALU is used as the address for the data memory**
5. **The data from the memory unit is written into the register file: the destination register ($rt) is given by bits 20-16 of the instruction**

Datapath for Load Operation

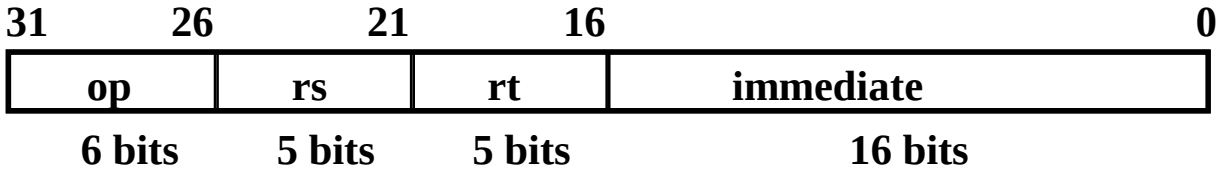
R[rt] ← Mem[R[rs] + SignExt[imm16]]

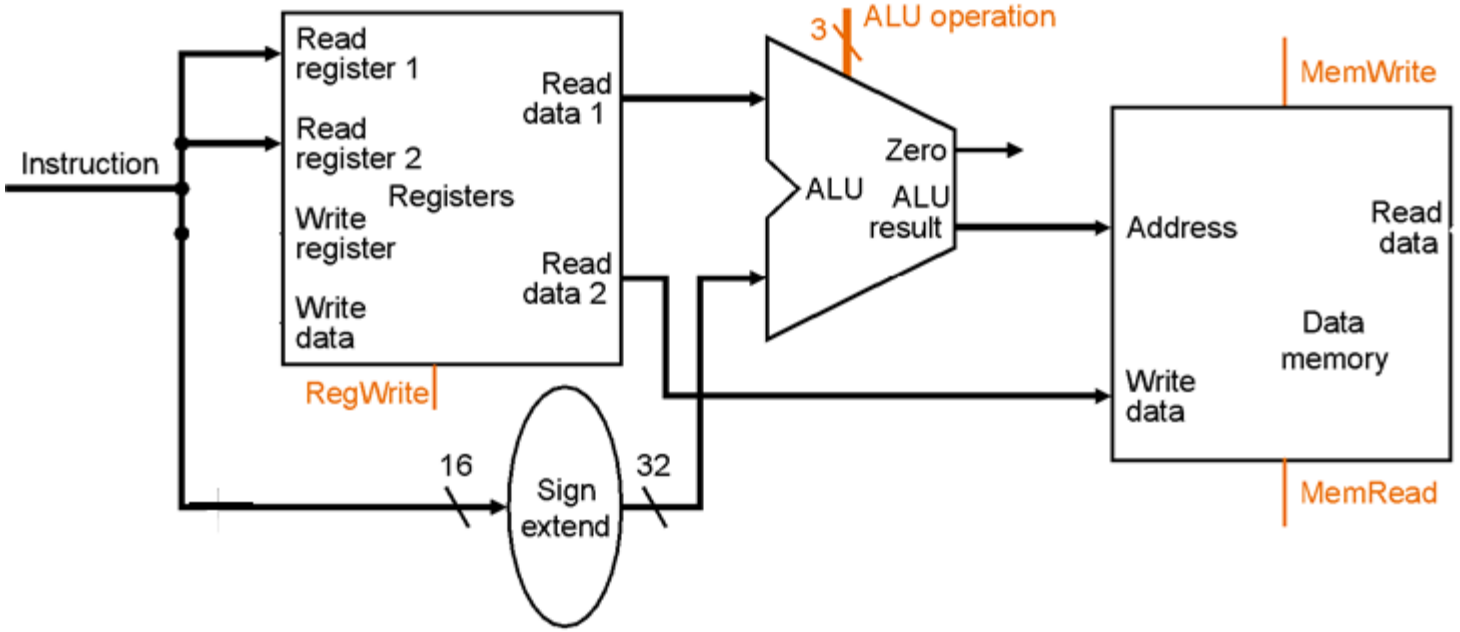




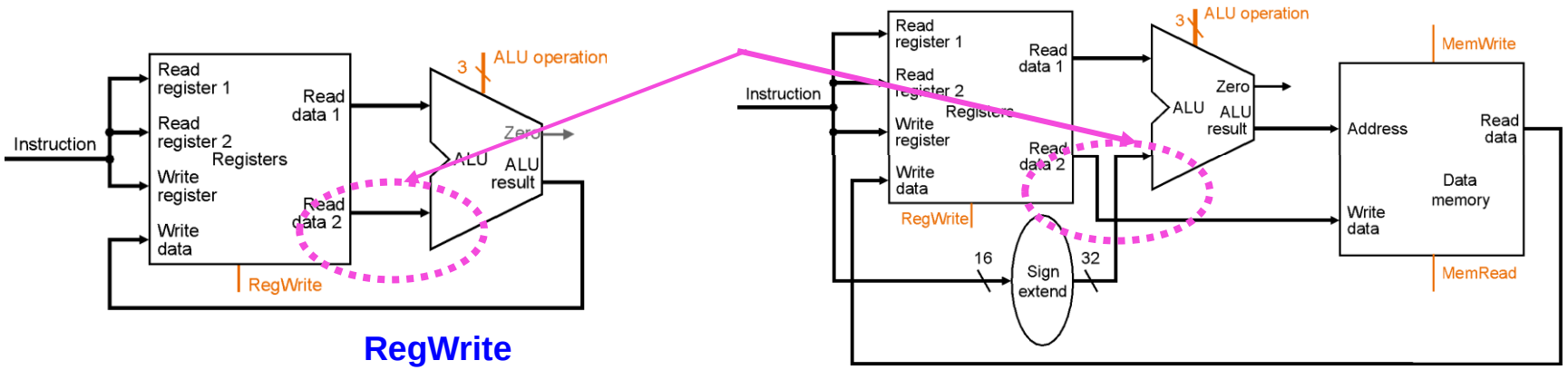
Datapath for Store Operation

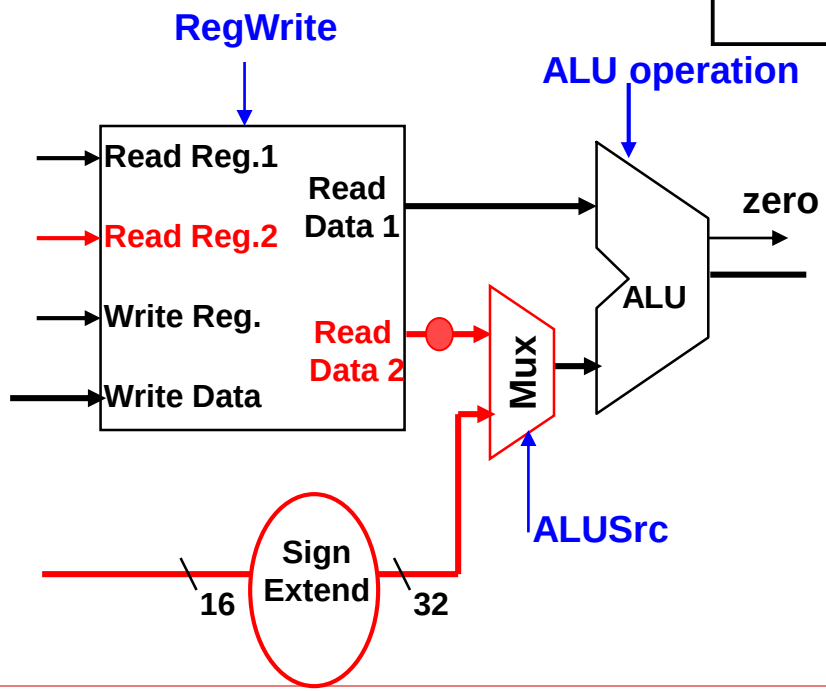
Mem[R[rs] + SignExt[imm16]] ← R[rt]





Combining datapaths



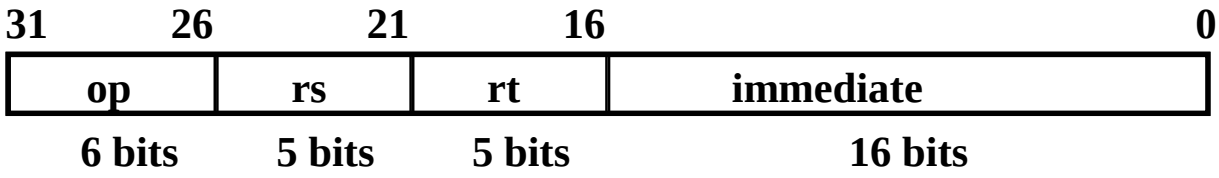


استفاده از مالتی پلکسربا سیگنال کنترلی ALUsrc

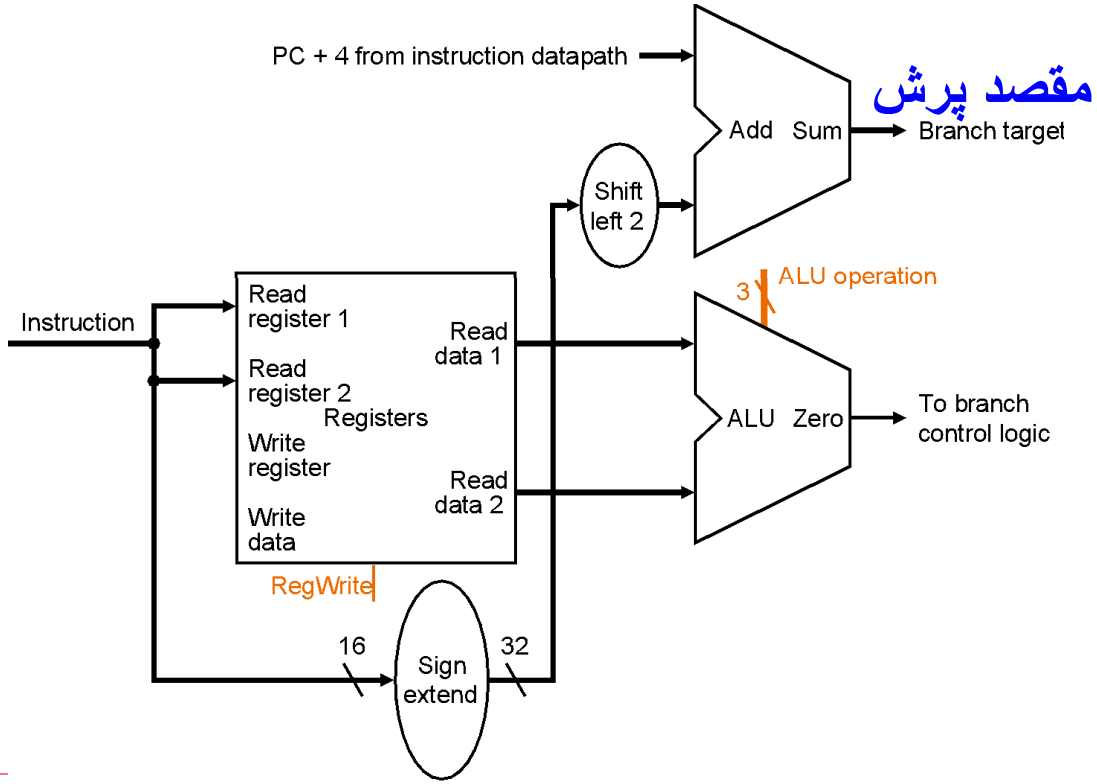
0=register

1=immediate

Datapath for Branch Operations



مقصد دستور انشعاب از جمع مقدار افست با **PC** بدست می آید. از آنجائیکه این مقصد باید مضربی از ٤ باشد، نیاز است تا مقدار افست ۲ بار به سمت چپ شیفت داده شود.



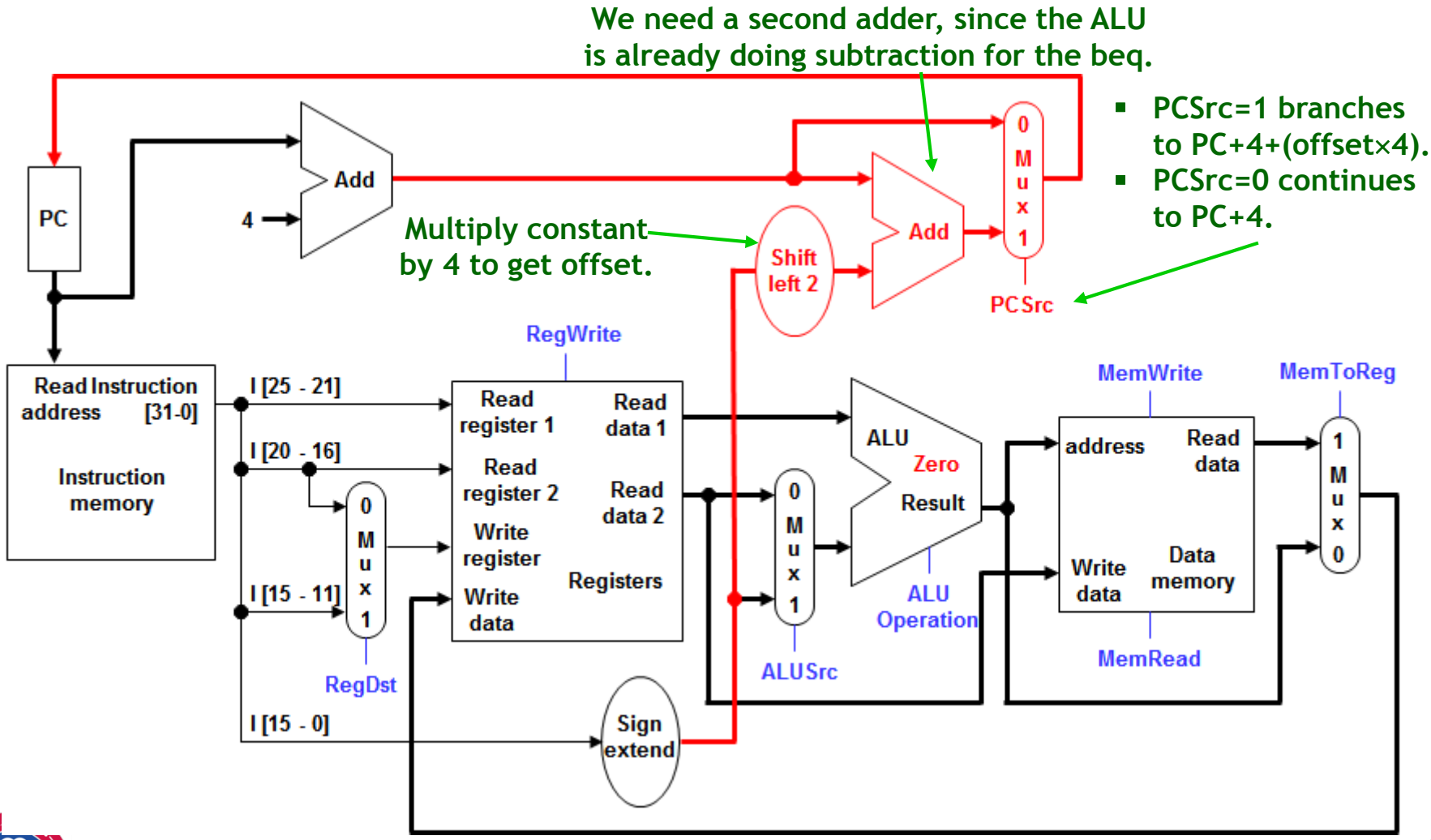
Branch Instruction Steps

beq $rs, $rt, offset

1. **Fetch instruction and increment PC**
2. **Read two register ($rs and $rt) from the register file**
3. **ALU performs a subtract on the data values from the register file; the value of PC+4 is added to the sign-extended lower 16 bits (offset) of the instruction shifted left by two to give the branch target address**
4. **The Zero result from the ALU is used to decide which adder result (from step 1 or 3) to store in the PC**

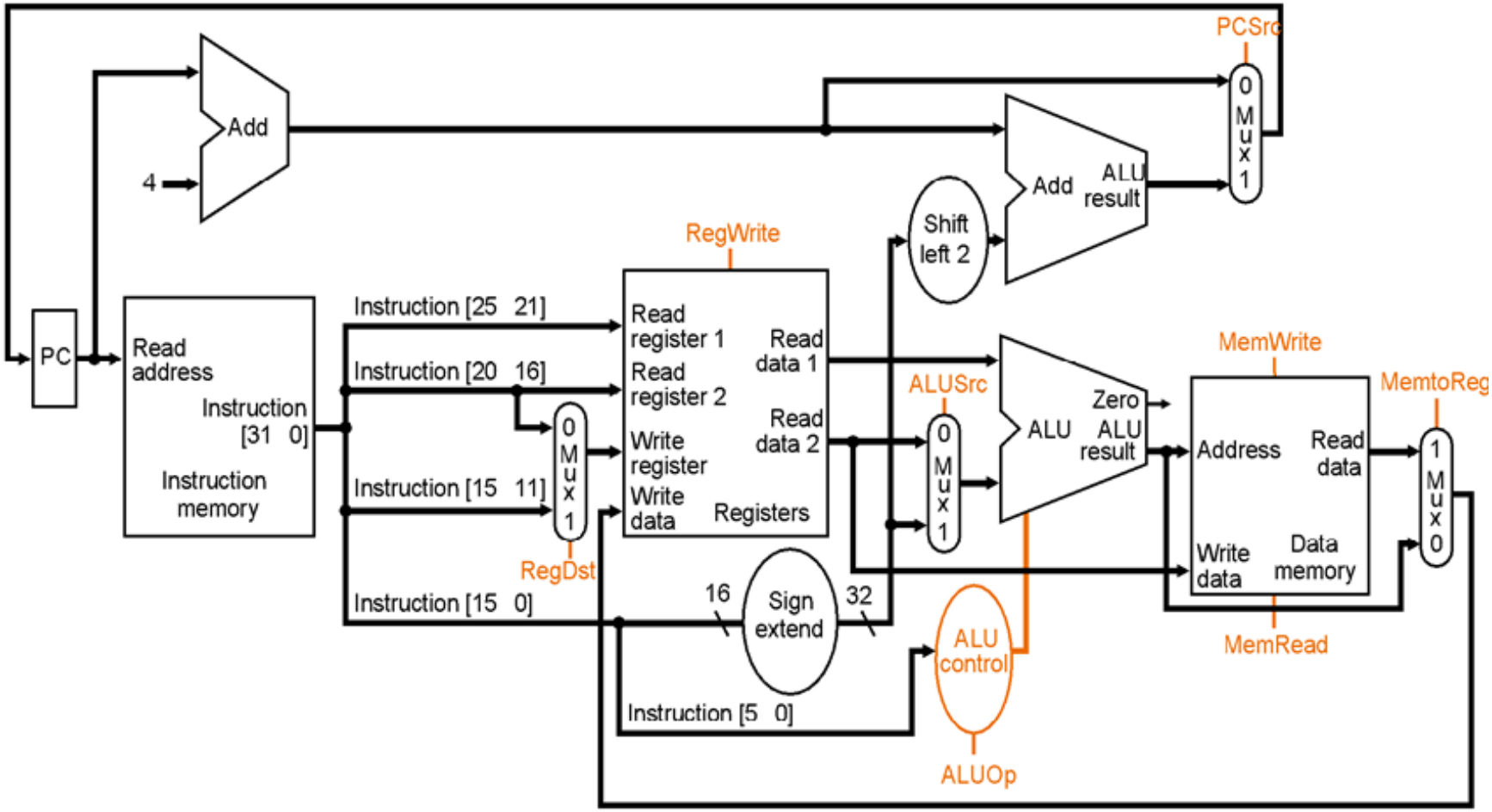
- برای مقایسه رجیسترهای دستور beq از ALU استفاده میشود (عمل تفریق). از اینرو نتیجه این مقایسه با سیگنال Zero که در خروجی ALU تعبیه شده است مشخص میگردد.
- بعلت درگیر بودن ALU، برای محاسبه آدرس مقصد از یک جمع کننده دیگر استفاده میشود.

Branching hardware

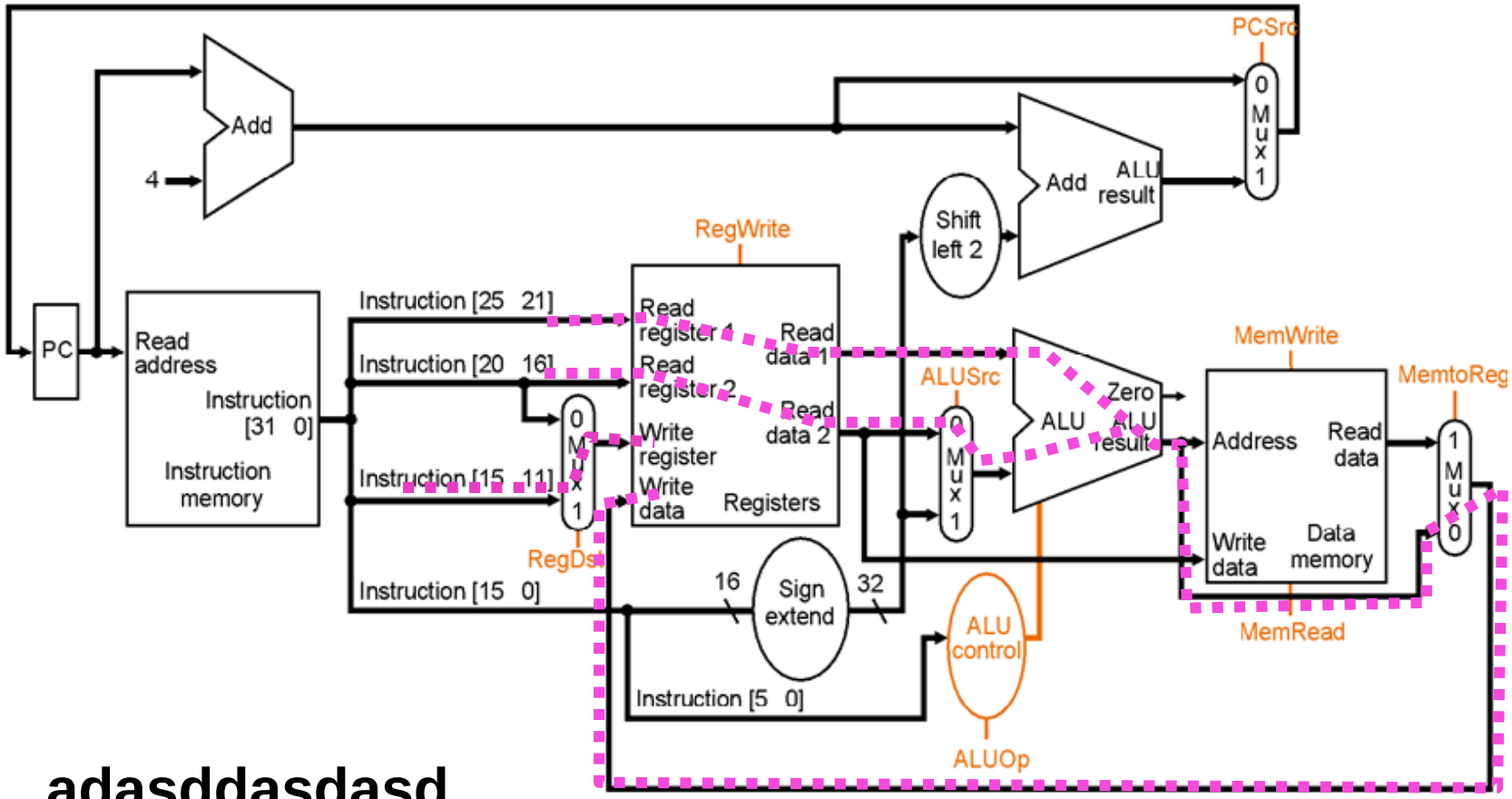


- اجزای مورد نیاز برای قسمت های مختلف در کنار هم قرار داده شده و سیگناهای کنترلی و مالتی پلکسرهای مورد نیاز به آن افزوده میشوند.
- در طراحی Single Cycle همه مراحل واکشی، دیکد و اجرا در یک کلاک انجام میشود!
- زمان این کلاک برابر خواهد بود با زمان لازم برای طی کردن طولانی ترین مسیر که میتواند زمان زیادی باشد.
- علاوه برآن امکان به اشتراک گذاشتن سخت افزار برای عملیات یکسان وجود ندارد.

All together: the single cycle datapath



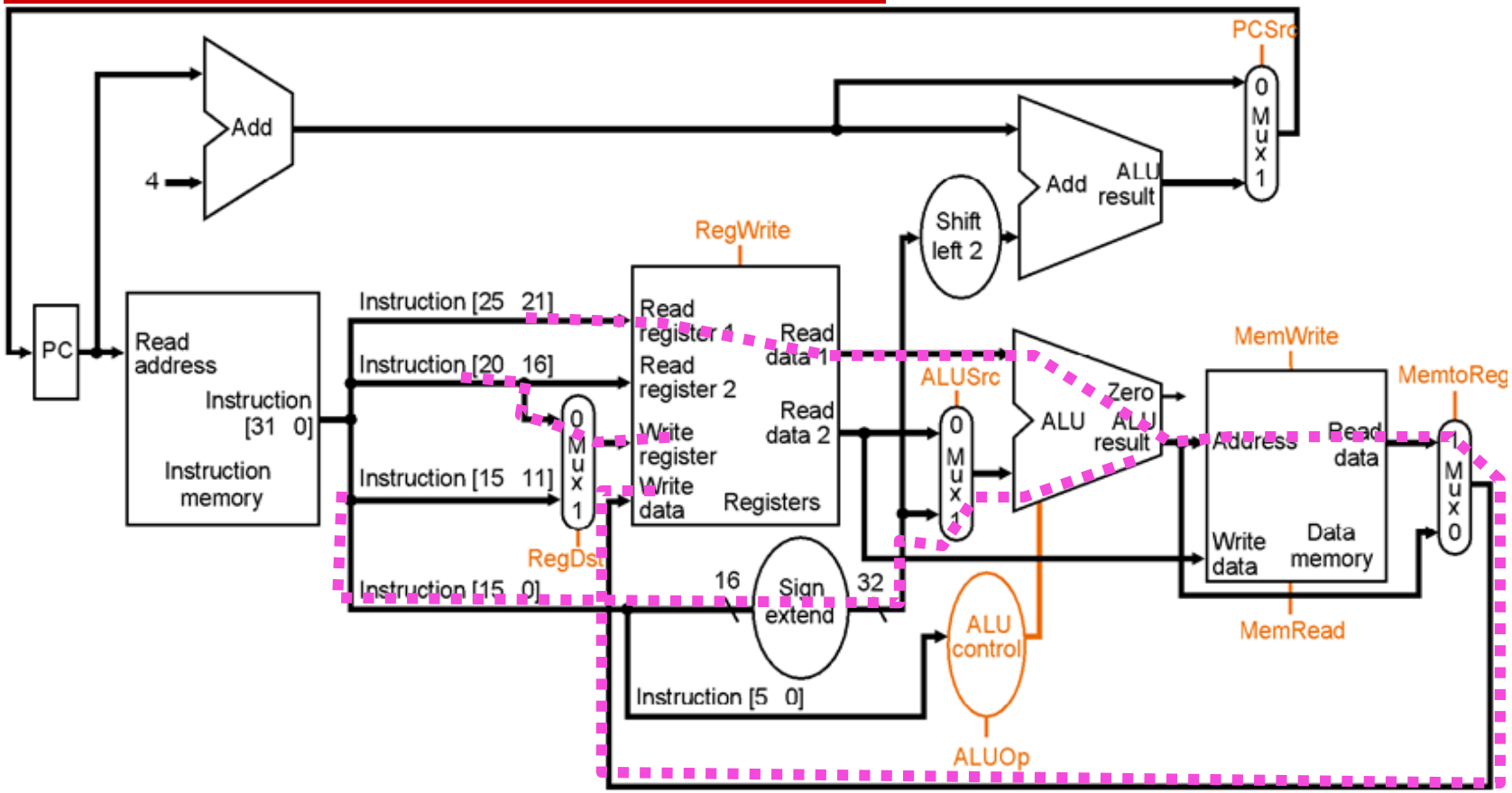
The R-Format (e.g. *add*) Datapath



adasddasdasd

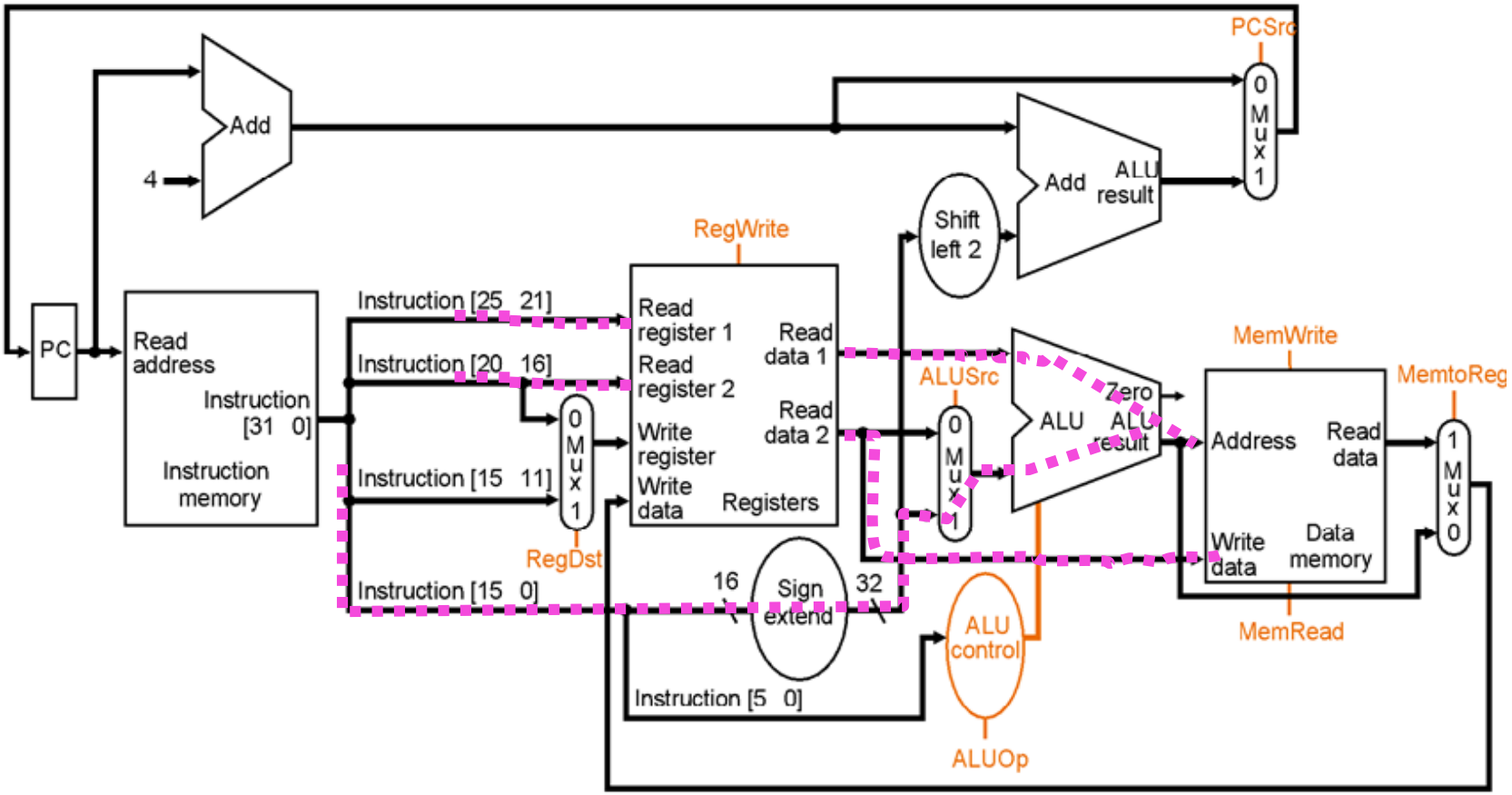
ALUsrc=0, ALUop="add", MemWrite=0, MemToReg=0, RegDst = 1, RegWrite=1 and PCsrc=0.

The Load Datapath

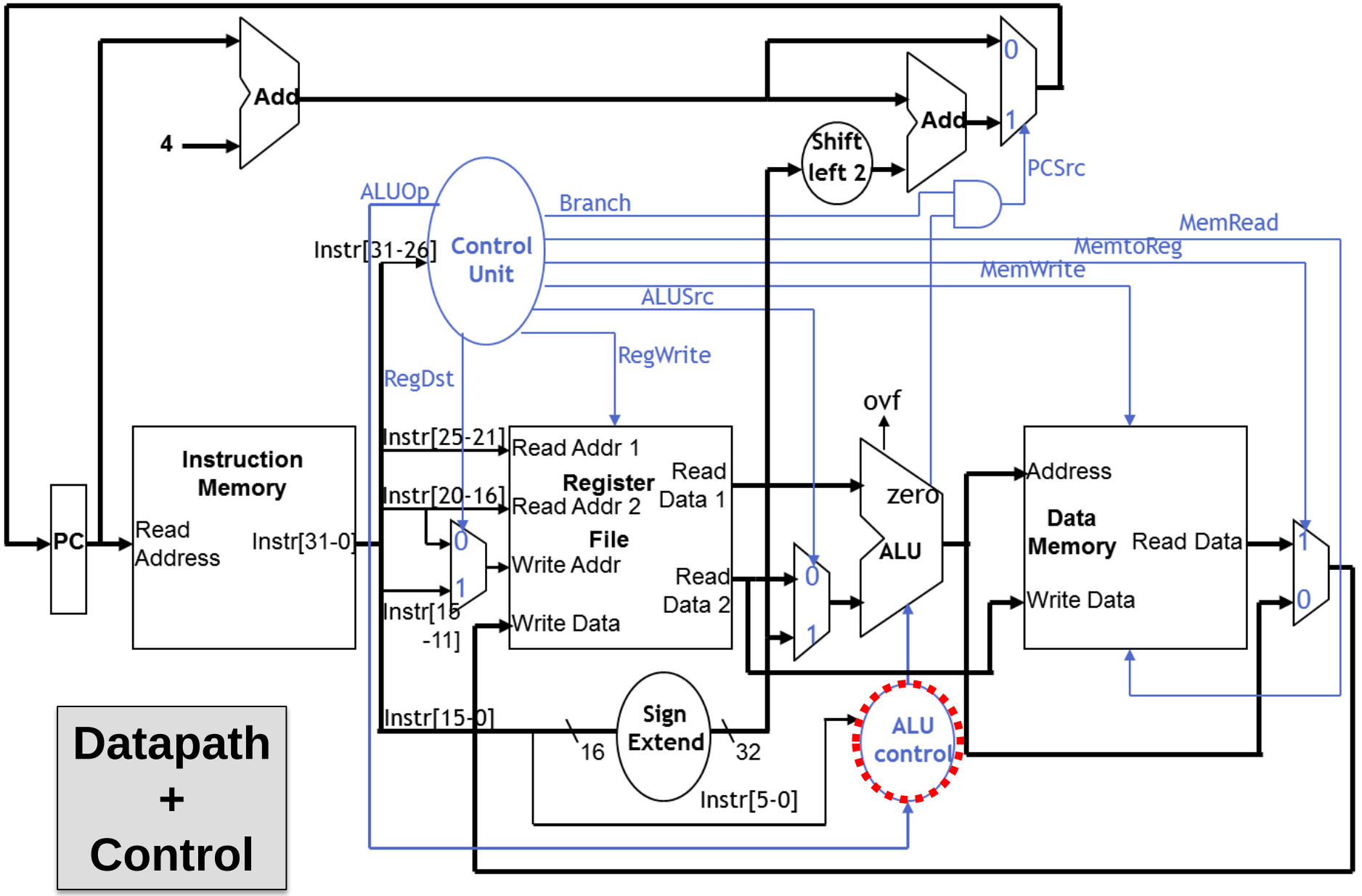


What control signals do we need for load??

The Store Datapath



Mem[R[rs] + SignExt[imm16]] ← R[rt]

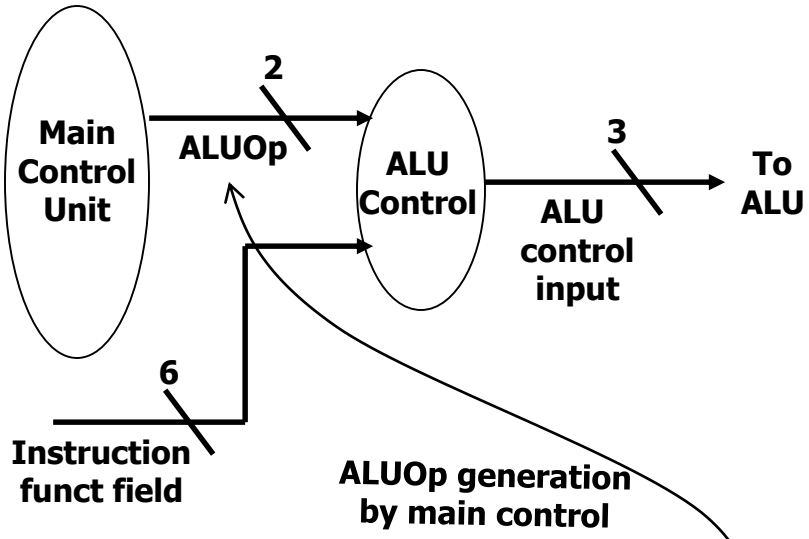


Datapath + Control

ALU Control

- Plan to control ALU: main control sends a 2-bit ALUOp control field to the ALU control. Based on ALUOp and funct field of instruction the ALU control generates the 3-bit ALU control field

ALU operation	Function
000	and
001	or
010	add
110	sub
111	slt (set less than)



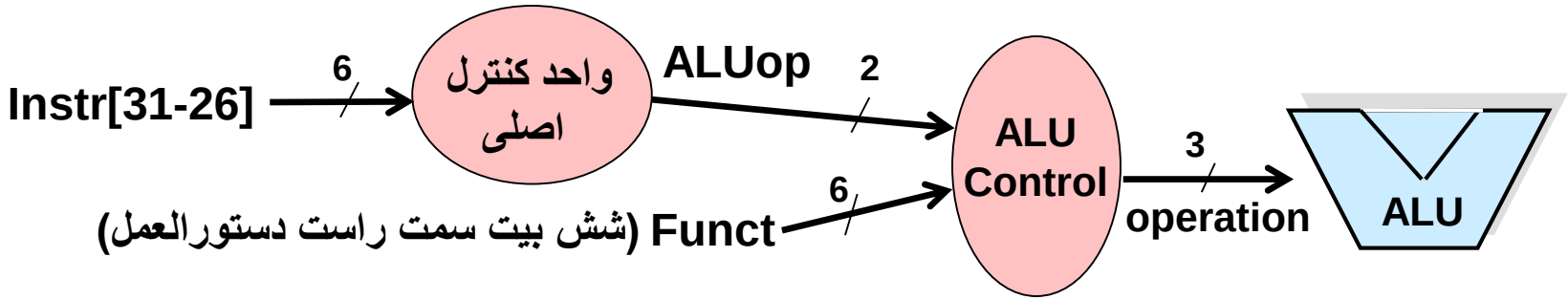
- ALU must perform
  - *add* for load/stores (ALUOp 00)
  - *sub* for branches (ALUOp 01)
  - one of *and*, *or*, *add*, *sub*, *slt* for R-type instructions, depending on the instruction's 6-bit funct field (ALUOp 10)

طراحی واحد کنترل ALU

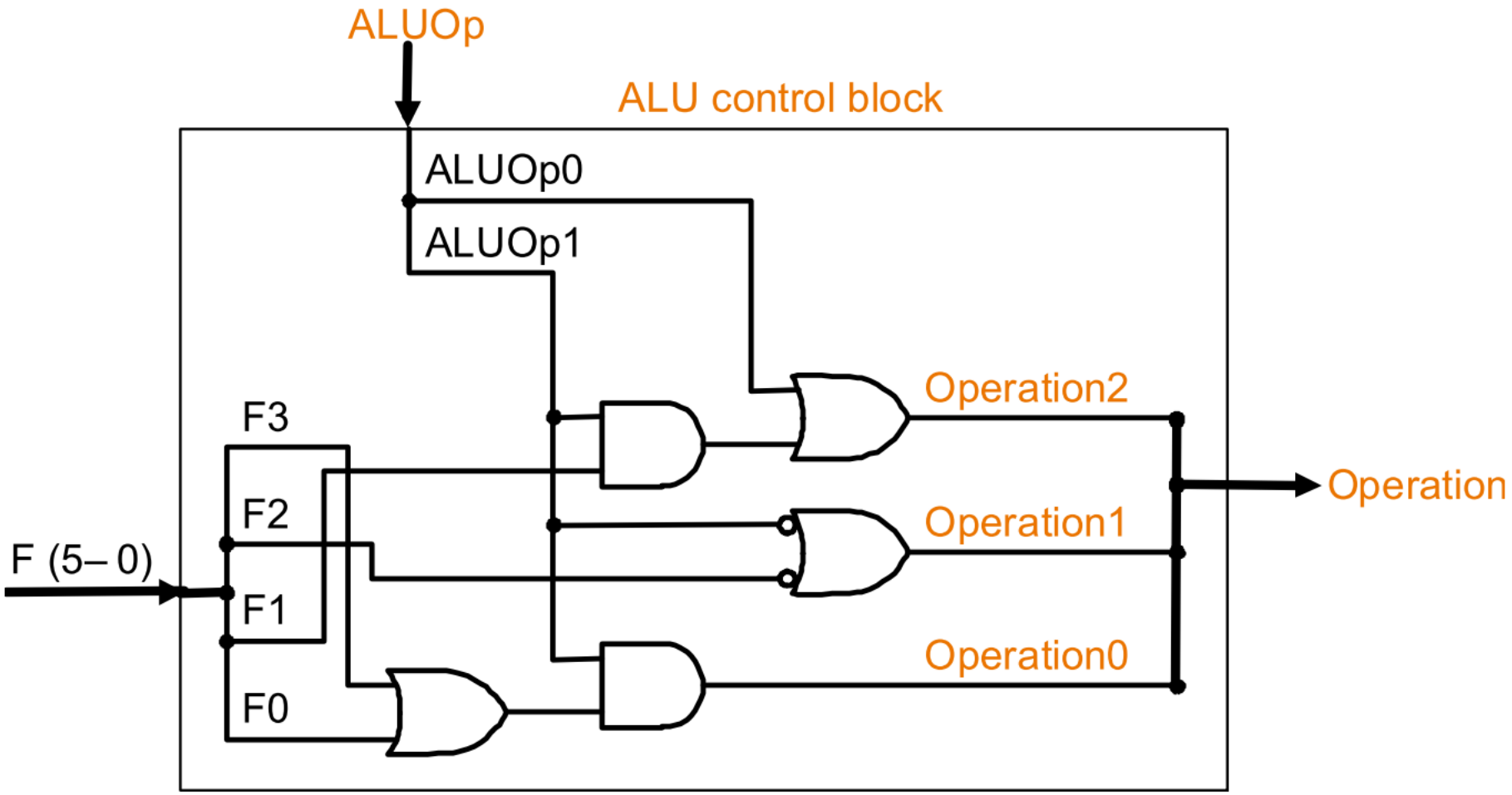
opcode دستور	ALUOp	عملیاتی که دستور انجام می‌دهد	فیلد Funct	عملیاتی که ALU انجام می‌دهد	خطوط کنترلی ALU
LW	00	Load word	xxxxxx	add	010
SW	00	Store word	xxxxxx	add	010
branch equal	01	Branch equal	xxxxxx	subtract	110
R-type	10	add	100000	add	010
R-type	10	subtract	100010	subtract	110
R-type	10	AND	100100	and	000
R-type	10	OR	100101	or	001
R-type	10	Set on less than	101010	Set on less than	111

طراحی واحد کنترل ALU

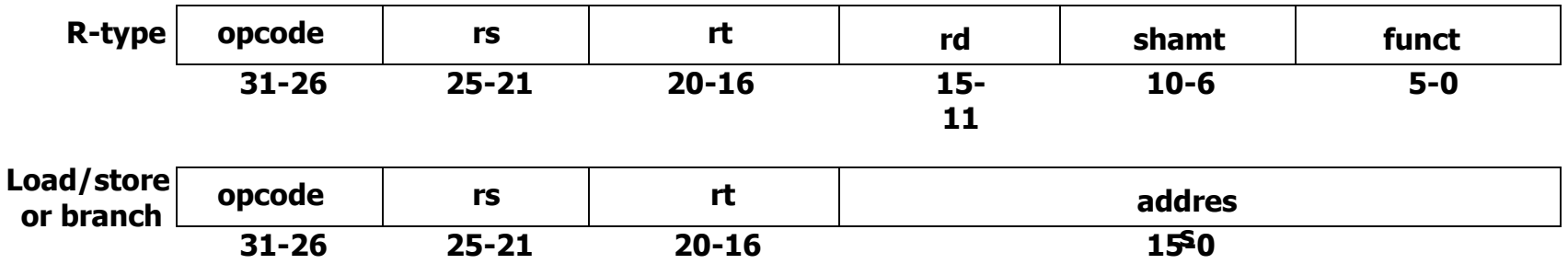
ALUOp		Funct فیلد						Operation (خطوط کنترل ALU)
ALUOp1	ALUOp0	F5	F4	F3	F2	F1	F0	
0	0	X	X	X	X	X	X	010
X	1	X	X	X	X	X	X	110
1	X	X	X	0	0	0	0	010
1	X	X	X	0	0	1	0	110
1	X	X	X	0	1	0	0	000
1	X	X	X	0	1	0	1	001
1	X	X	X	1	0	1	0	111



طراحی واحد کنترل ALU



Designing the Main Control



Observations about MIPS instruction format

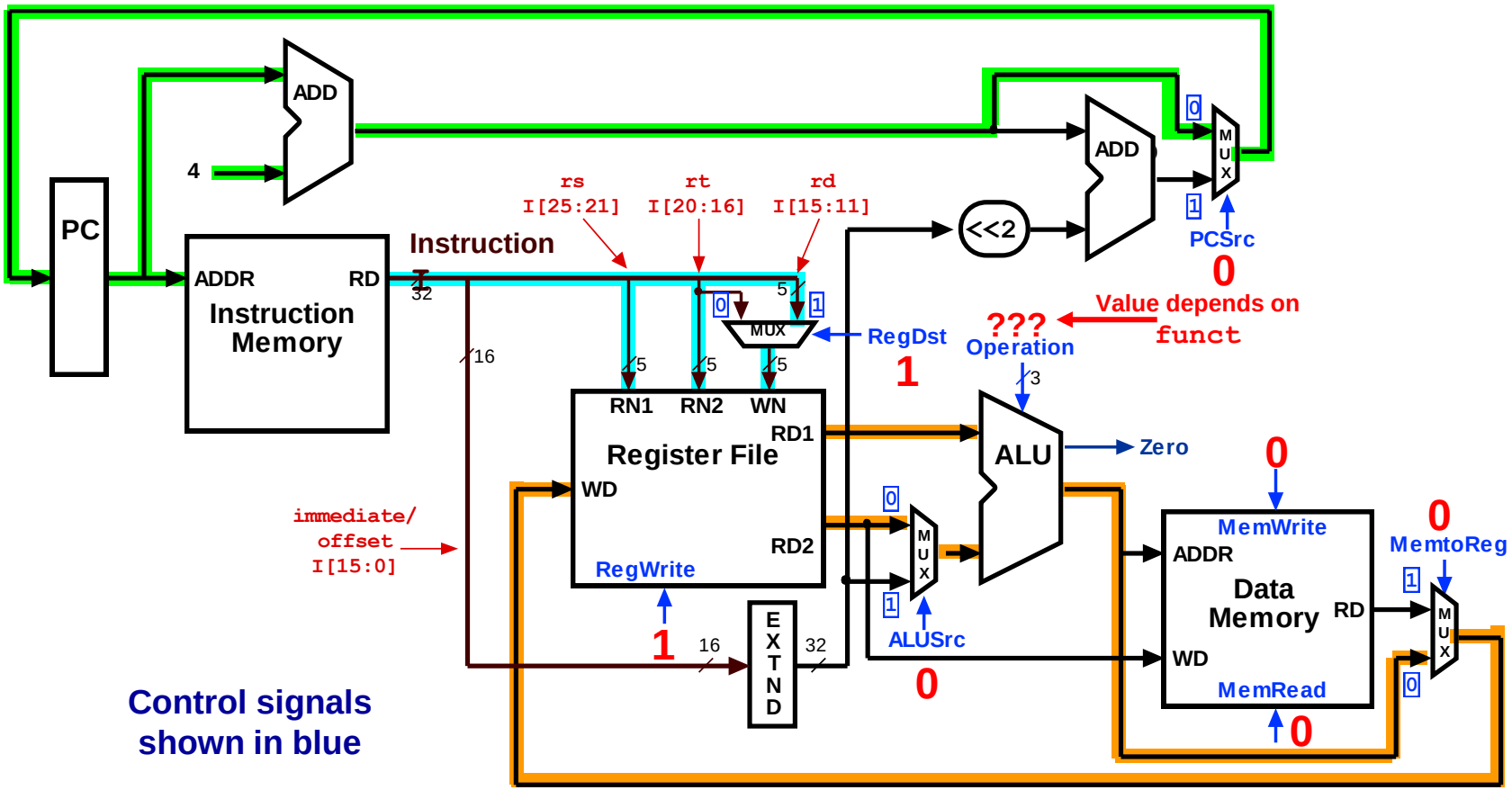
- opcode is always in bits 31-26
- two registers to be read are always rs (bits 25-21) and rt (bits 20-16)
- base register for load/stores is always rs (bits 25-21)
- 16-bit offset for branch equal and load/store is always bits 15-0
- destination register for loads is in bits 20-16 (rt) while for R-type instructions it is in bits 15-11 (rd) (*will require multiplexor to select*)

Control Signals

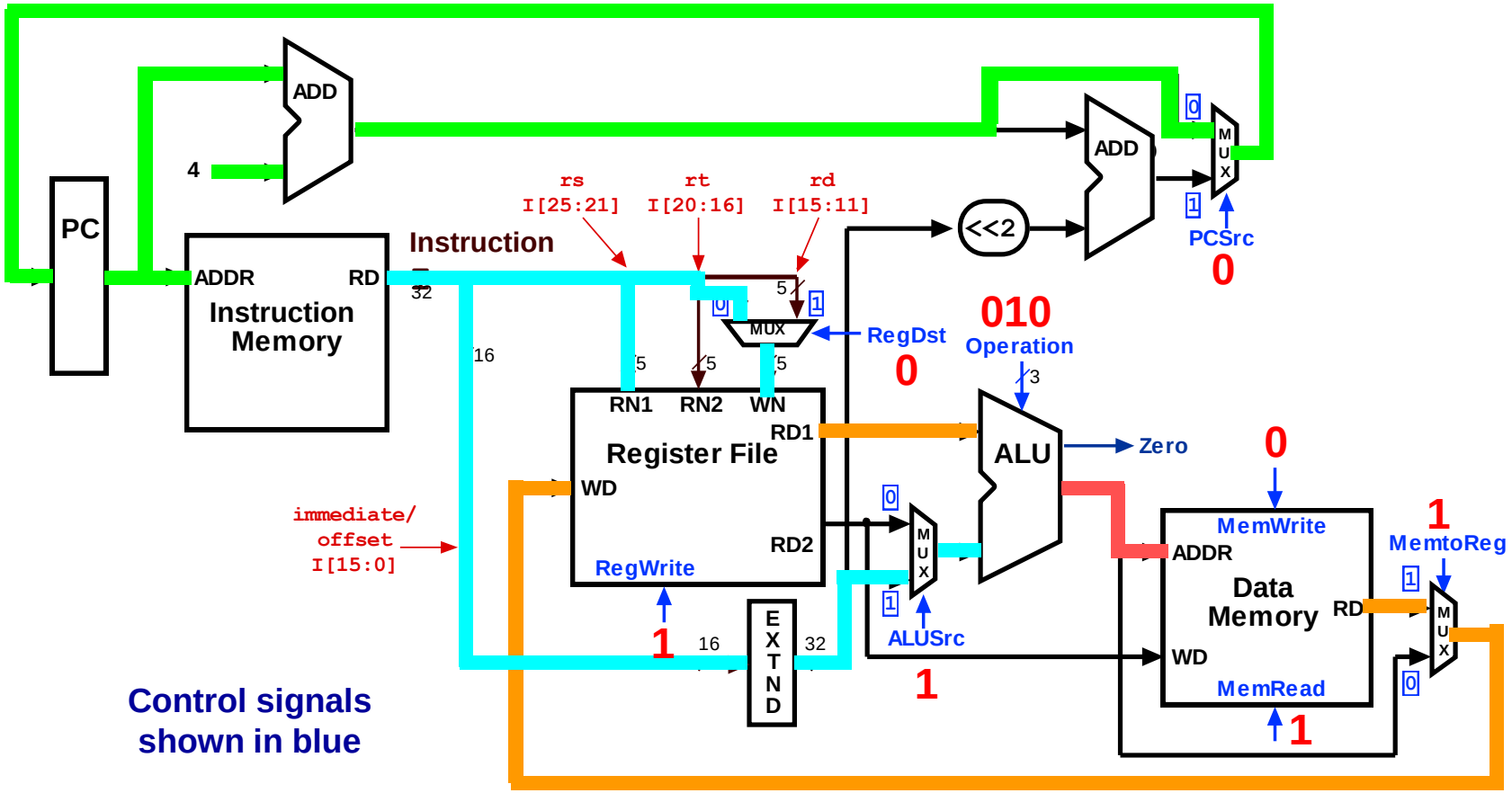
Signal Name	0	1
RegDst	The register destination number for the Write register comes from the rt field (bits 20-16)	The register destination number for the Write register comes from the rd field (bits 15-11)
AlLUSrc	The second ALU operand comes from the second register file output (Read data 2)	The second ALU operand is the sign-extended, lower 16 bits of the instruction
MemtoReg	The value fed to the register Write data input comes from the ALU	The value fed to the register Write data input comes from the data memory
RegWrite	None	The register on the Write register input is written with the value on the Write data input
PCSrc	The PC is replaced by the output of the adder that computes the value of PC + 4	The PC is replaced by the output of the adder that computes the branch target
MemRead	None	Data memory contents designated by the address input are put on the first Read data output
MemWrite	None	Data memory contents designated by the address input are replaced by the value of the Write data input

Instruction	OP-code	RegDst	ALUSrc	Memto-Reg	Reg Write	Mem Read	Mem Write	PCSrc	ALUOp 1,0
R-format	000000	1	0	0	1	0	0	0	1 0
lw	100011	0	1	1	1	1	0	0	0 0
sw	101011	X	1	X	0	0	1	0	0 0
beq	000100	X	0	X	0	0	0	1	0 1

Control Signals: R-Type Instruction



Control Signals: `lw` Instruction



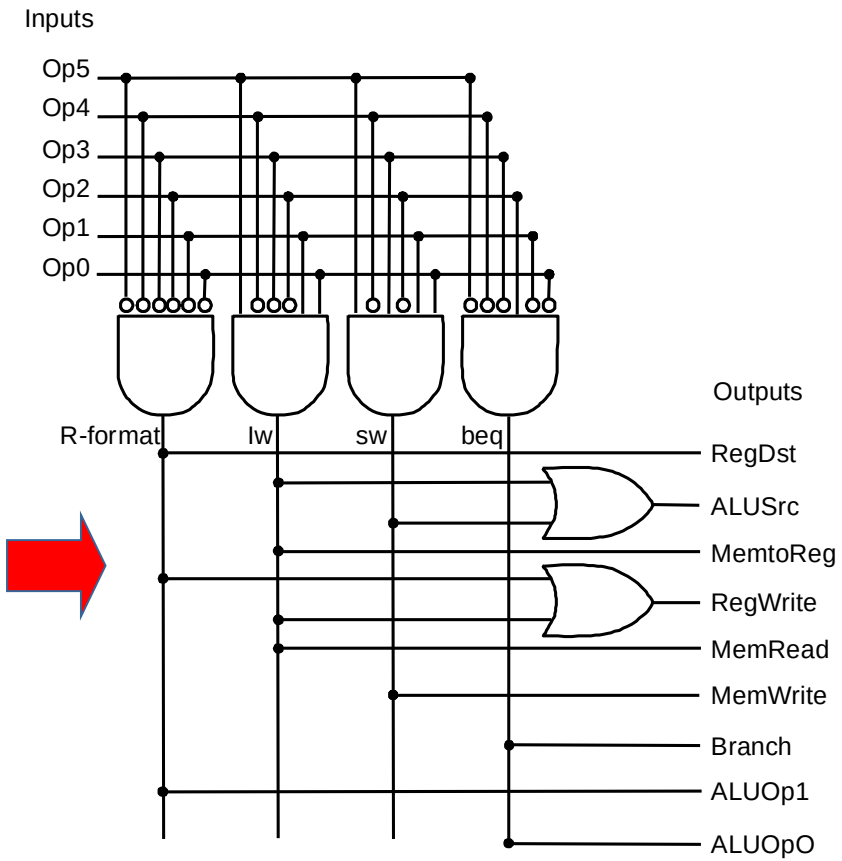
Single-cycle Implementation Notes

- *The steps are not really distinct* as each instruction completes in exactly one clock cycle – they simply indicate the sequence of data flowing through the datapath
- *The operation of the datapath during a cycle is purely combinational* – nothing is stored during a clock cycle
- Therefore, the machine is stable in a particular state at the start of a cycle and reaches a new stable state only at the end of the cycle
- *Very important for understanding single-cycle computing*:

Implementation: Main Control Block

	Signal	R-type	lw	sw	beq
Inputs	Op5	0	1	1	0
	Op4	0	0	0	0
	Op3	0	0	1	0
	Op2	0	0	0	1
	Op1	0	1	1	0
	Op0	0	1	1	0
Outputs	RegDst	1	0	x	x
	ALUSrc	0	1	1	0
	MemtoReg	0	1	x	x
	RegWrite	1	1	0	0
	MemRead	0	1	0	0
	MemWrite	0	0	1	0
	Branch	0	0	0	1
	ALUOp1	1	0	0	0
	ALUOP2	0	0	0	1

Truth table for main control signals



Main control PLA (programmable logic array): principle underlying PLAs is that any logical expression can be written as a sum-of-products

Single-Cycle Design Problems

- Assuming fixed-period clock every instruction datapath uses one clock cycle implies:
  - CPI = 1
  - cycle time determined by length of **the longest instruction path** (load)
    - but several instructions could run in a shorter clock cycle: ***waste of time***
    - consider if we have more complicated instructions like floating point!
  - resources used more than once in the same cycle need to be duplicated: ***waste of hardware and chip area***

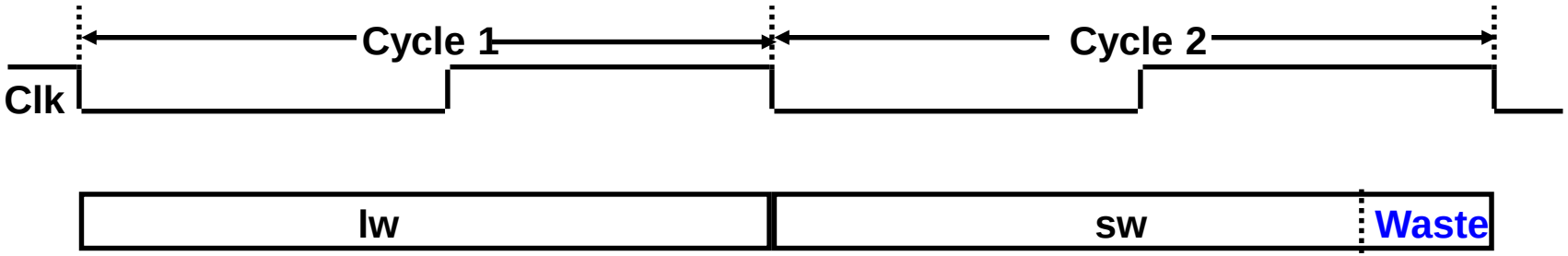
Fixing the problem with single-cycle designs

- One solution: a variable-period clock with different cycle times for each instruction class
  - *unfeasible*, as implementing a variable-speed clock is technically difficult
- Another solution:
  - use a smaller cycle time…
  - …have different instructions take different numbers of cycles

  by breaking instructions into steps and fitting each step into one cycle
  - *feasible: multicyle approach*!

مزایا و معایب معماری Single Cycle

- زمان کلاک بطور موثر استفاده نمیشود زیرا بر اساس طولانی ترین دستور تنظیم شده است. این امر در صورت داشتن دستورات پیچیده مثل دستورات اعشاری میتواند خیلی وخیم باشد.
- فضای بیشتری در روی چیپ لازم دارد زیرا تعداد بیشتری از المانهای سخت افزاری لازم دارد.
- ساده و قابل فهم است.

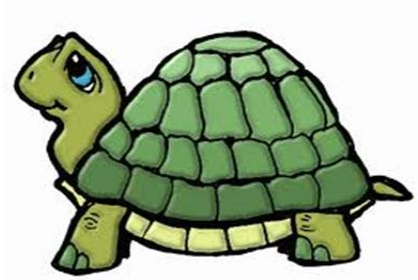


محاسبه طولانی ترین دستورالعمل

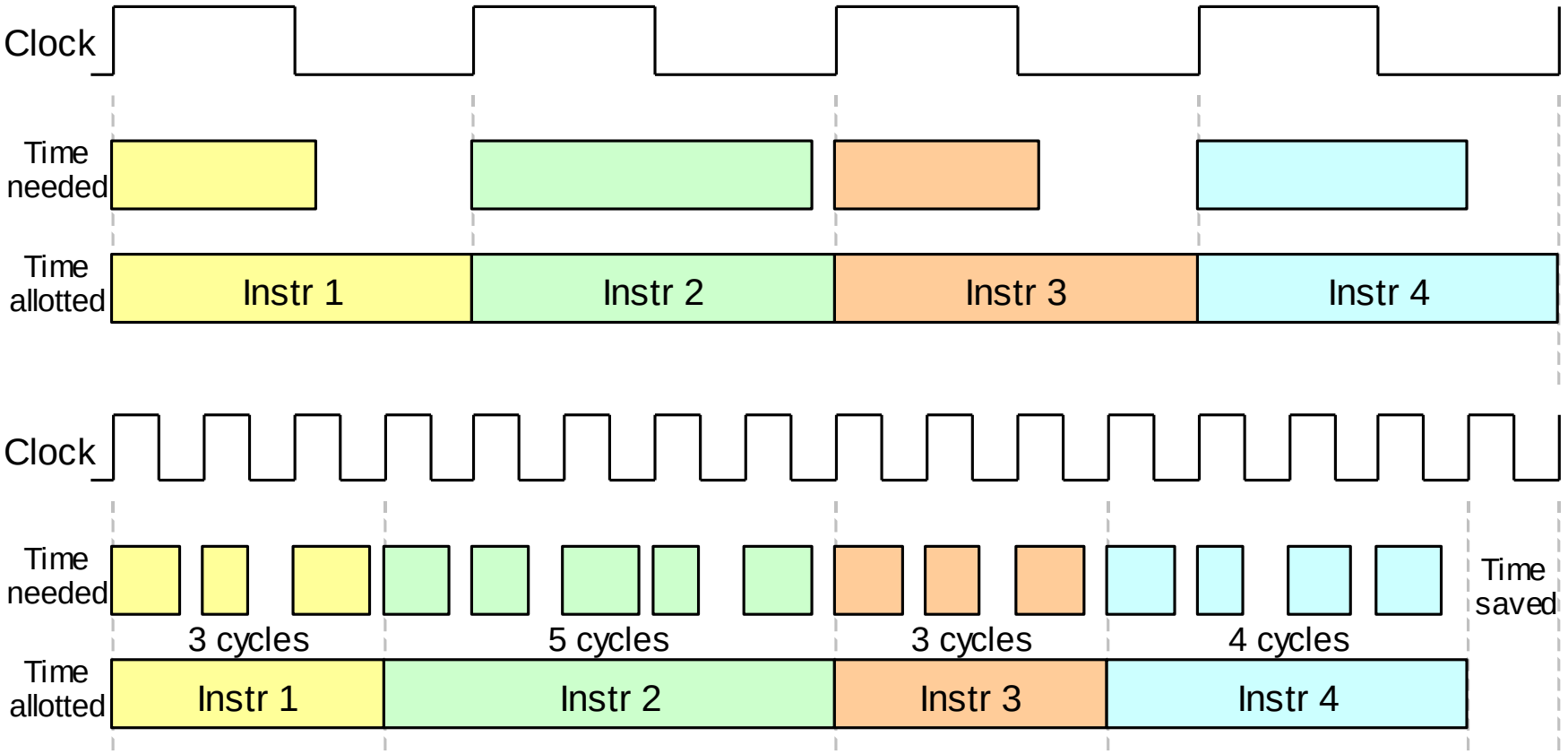
Instruction class	Instruction memory	Register read	ALU operation	Data memory	Register write	Total (ps)
R- type	200	50	100	0	50	400
lw	200	50	100	200	50	600
Sw	200	50	100	200		550
Branch	200	50	100			350

- طول کلاک باید بر اساس زمان لازم برای طولانی ترین دستور طراحی شود. 600 ps

Instruction	Frequency
Arithmetic	48%
Loads	22%
Stores	11%
Branches	19%



A Multicycle Implementation

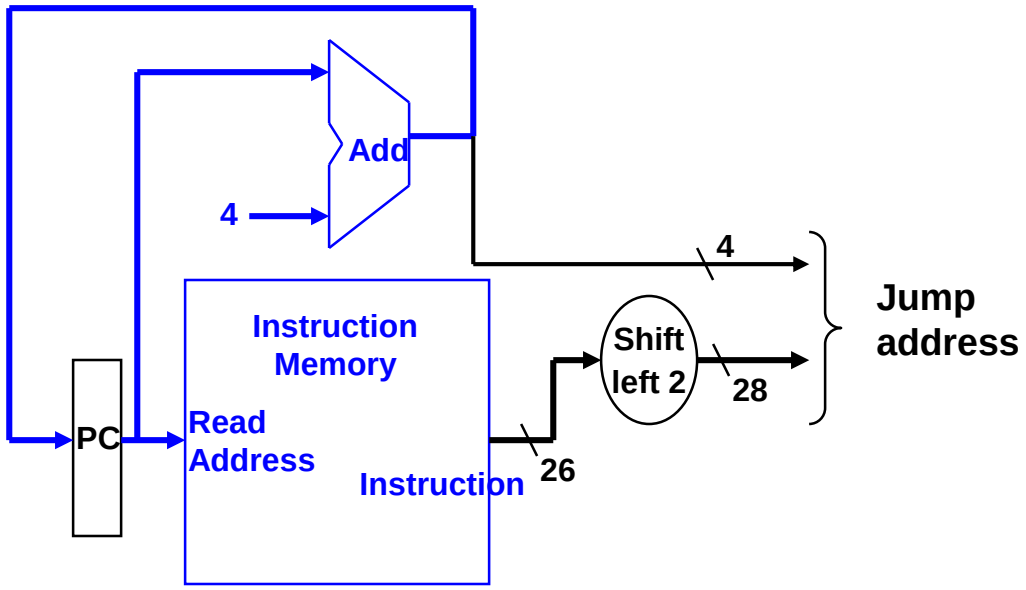


Single-cycle versus multicycle instruction execution.


Questions

Datapath Building Blocks: jump instruction

- اجرای دستور jump از طریق المانهای زیر انجام میشود:
- ۲۶ بیت مقدار موجود در دستورالعمل به اندازه ۲ بیت به سمت چپ شیفت داده شده و با ۲۸ بیت کم ارزشPC جایگزین میشود.



Example: Fixed-period clock vs variable-period clock in a single-cycle implementation

- Consider a machine with an additional floating point unit. Assume functional unit delays as follows
  - *memory*: 2 ns., *ALU* and *adders*: 2 ns., *FPU add*: 8 ns., *FPU multiply*: 16 ns., *register file access* (read or write): 1 ns.
  - *multiplexors*, *control unit*, *PC accesses*, *sign extension*, *wires*: no delay
- Assume instruction mix as follows
  - all loads take same time and comprise 31%
  - all stores take same time and comprise 21%
  - R-format instructions comprise 27%
  - branches comprise 5%
  - jumps comprise 2%
  - FP adds and subtracts take the same time and totally comprise 7%
  - FP multiplys and divides take the same time and totally comprise 7%
- *Compare the performance of (a) a single-cycle implementation using a fixed-period clock with (b) one using a variable-period clock where each instruction executes in one clock cycle that is only as long as it needs to be (not really practical but pretend it's possible!)*

Solution

Instruction class	Instr. mem.	Register read	ALU oper.	Data mem.	Register write	FPU add/ sub	FPU mul/ div	Total time ns.
Load word	2	1	2	2	1			8
Store word	2	1	2	2				7
R-format	2	1	2	0	1			6
Branch	2	1	2					5
Jump	2							2
FP mul/div	2	1			1		16	20
FP add/sub	2	1			1	8		12

- Clock period for fixed-period clock = longest instruction time = 20 ns.
- Average clock period for variable-period clock = $8 \times 31\% + 7 \times 21\% + 6 \times 27\% + 5 \times 5\% + 2 \times 2\% + 20 \times 7\% + 12 \times 7\%$ = 7.0 ns.
- Therefore, $\text{performance}_{\text{var-period}} / \text{performance}_{\text{fixed-period}} = 20/7 = 2.9$