دانشکده مهندسی کامپیوتر

معماری کامپیوتر

آزمون میان‌ترم

نام و نام خانوادگی:

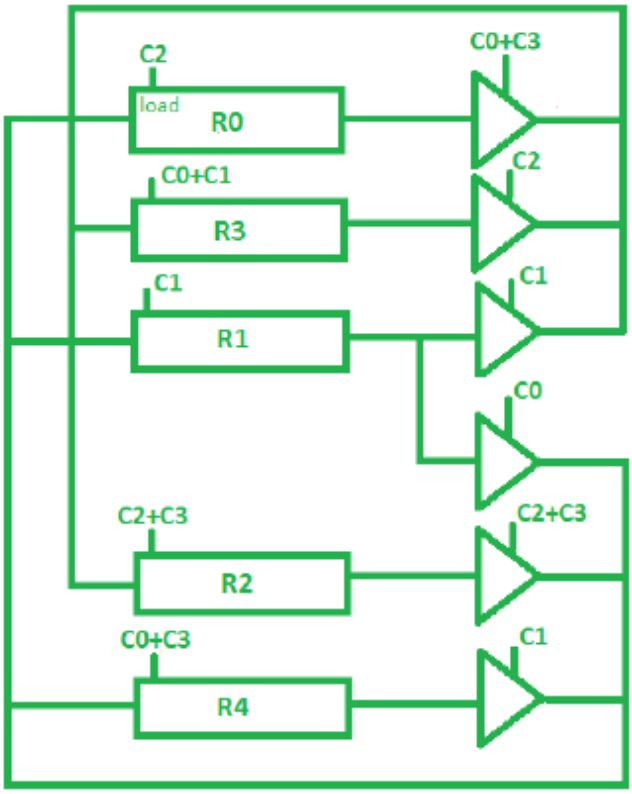
شماره دانشجویی:

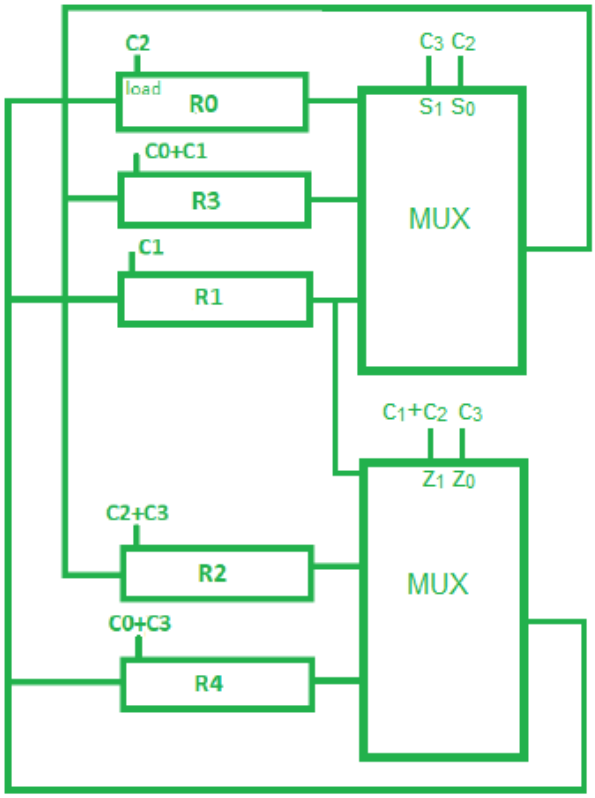
اردیبهشت ۱۴۰۳ | زمان آزمون: ۱۲۰ دقیقه | مدرس: لاله ارشدی

۱- (۱۰ نمره) ثبات‌های R0 تا R4 را با حداکثر دو گذرگاهِ مشترک، طوری به هم متصل کنید که بتوان توصیفِ RTL زیر را روی آن اجرا کرد. این دو گذرگاه را یک بار با بافرهای سه‌حالته و یک بار با مالتی‌پلکسر بسازید. فرض کنید ثبات‌ها n بیتی هستند و هر کدام یک ورودی load دارند. همچنین فرض کنید سیگنال‌های C0 تا C3 هرگز همزمان یک نمی‌شوند.

```
C0: R3 ← R0, R4 ← R1
C1: R3 ← R1, R1 ← R4
C2: R2 ← R3, R0 ← R2
C3: R2 ← R0, R4 ← R2
```

پاسخ:





۲- (۱۰ نمره) پردازنده‌ای داریم که ۲۰٪ دستورات آن در یک چرخه اجرا می‌شوند و طبعا نمی‌توان آنها را سریع‌تر از این اجرا کرد. نصفِ بقیهٔ دستورات در ۲ و نصفِ دیگر آنها در ۴ چرخه اجرا می‌شوند. ما می‌توانیم تغییراتی در این پردازنده بدهیم که تعداد چرخه‌های این دستورات را نصف کنیم اما در مقابل باید چرخهٔ ساعت را افزایش دهیم.

الف- آیا می‌توانید بدون محاسبه بگویید که اگر چرخهٔ ساعت دو برابر شود، کارایی پردازنده در مجموع بهتر خواهد شد یا بدتر؟ توضیح دهید.

ب- حداکثر نسبت چرخهٔ ساعت جدید به قدیم چقدر باشد که به تسریعی بیش از یک برسیم؟

پاسخ:

الف- ۲۰٪ از دستورات در یک چرخه اجرا می‌شوند و نمی‌توانیم آنها را سریع‌تر اجرا کنیم، بنابراین فقط تعداد چرخه‌های ۸۰٪ از دستورات نصف می‌شود ولی طبعا چرخهٔ ساعت برای همهٔ دستورات دو برابر می‌شود، پس در مجموع کارایی کاهش می‌یابد.

ب-

$$CPI_{avg1} = 0.2 \times 1 + 0.4 \times 2 + 0.4 \times 4 = 0.2 + 0.8 + 1.6 = 2.6$$
$$T_1 = 2.6 \times CC_1$$

$$CPI_{avg2} = 0.2 \times 1 + 0.4 \times 1 + 0.4 \times 2 = 0.2 + 0.4 + 0.8 = 1.4$$
$$T_2 = 1.8 \times CC_2$$

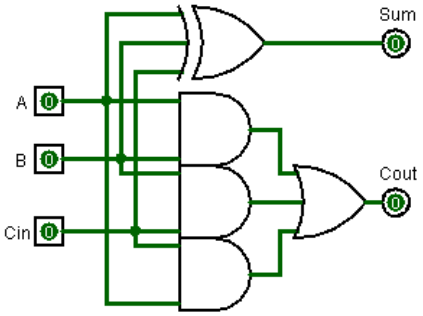
$$T_1 \geq T_2 \quad \Rightarrow \quad 2.6 \times CC_1 \geq 1.4 \times CC_2 \quad \Rightarrow \quad \frac{CC_2}{CC_1} \leq 1.86$$

بارم‌بندی:

الف- ۲ نمره برای استدلال درست و ۲ نمره برای این که کارایی کم می‌شود.

ب- ۲ نمره برای محاسبهٔ هر کدام از CPIها و ۲ نمره برای پاسخ نهایی.

۳- (۱۵ نمره) می‌خواهیم چهار عددِ چهار بیتی A، B، D و E را با هم جمع کنیم. برای این منظور از دو روش استفاده می‌کنیم. در روشِ اول اعداد را دوتا دوتا جمع می‌کنیم تا به دو عددِ X و Y برسیم. سپس X و Y را جمع می‌کنیم. در روشِ دوم از جمع‌کننده‌های Carry-Save-Adder برای به دست آوردنِ دو عددِ X و Y استفاده می‌کنیم و سپس این دو عدد را با یک جمع‌کنندهٔ عادی با هم جمع می‌کنیم. فرض کنید، برای انجامِ این کار تعدادِ کافی تمام‌افزا (Full Adder) در اختیار داریم که مطابقِ شکلِ زیر عمل می‌کند و نیز فرض کنید تاخیرِ گیتِ XOR برابر با تاخیرِ یک گیتِ AND و یک گیتِ OR است. این تاخیر را d بنامید.



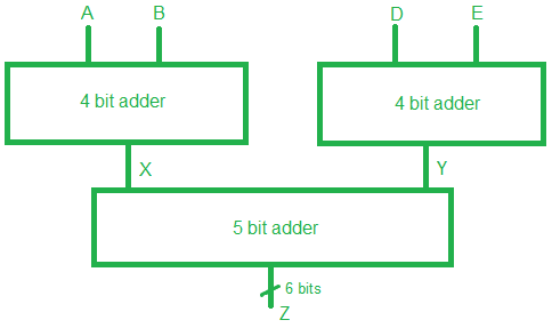
الف- عملکردِ هر یک از دو روش را با رسمِ شکل توضیح دهید.

ب- تاخیرِ هر یک از دو روش را محاسبه و با هم مقایسه کنید.

ج- اگر جمع‌کننده‌های روشِ اول و جمع‌کننده‌ای که در روشِ دوم برای جمعِ X و Y استفاده می‌شود از نوع Carry-Look-Ahead باشند، دوباره تاخیر هر یک از دو روش را محاسبه و با هم مقایسه کنید.

(به تعدادِ بیت‌های جمع‌کننده‌ها در هر مرحله توجه کنید.)

پاسخ:



در روش اول مطابق شکل، A+B و D+E همزمان محاسبه می‌شوند و سپس نتایج ۵ بیتی آنها با هم جمع می‌شوند. هر کدام از جمع‌کننده‌ها ۴ یا ۵ بیتی هم از کنار هم گذاشتن تمام‌افزاهایی مطابق با شکل بالا ساخته می‌شوند.

بیت‌های Y و X را به ترتیب Y_0-Y_4 و X_0-X_4 و بیت‌های Z را Z_0-Z_5 می‌نامیم. همچنین بیت‌های نقلی جمع‌کنندهٔ ۵ بیتی را C_0-C_5 می‌نامیم و توجه داریم که Z_5 همان C_5 است.

می‌دانیم که در هر مرحله $Z_i = X_i + Y_i + C_i, C_{i+1}$، بنابراین تاخیر بیت‌های Zi به این شکل حساب می‌شود:

$$D(Y_0) = D(X_0) = D(C_0) = d$$
$$D(Z_0) = D(C_1) = D(X_1) = D(Y_1) = 2d$$
$$D(Z_1) = D(C_2) = D(X_2) = D(Y_2) = 3d$$
...
$$D(Z_4) = D(C_5) = D(X_4) + d = 6d$$

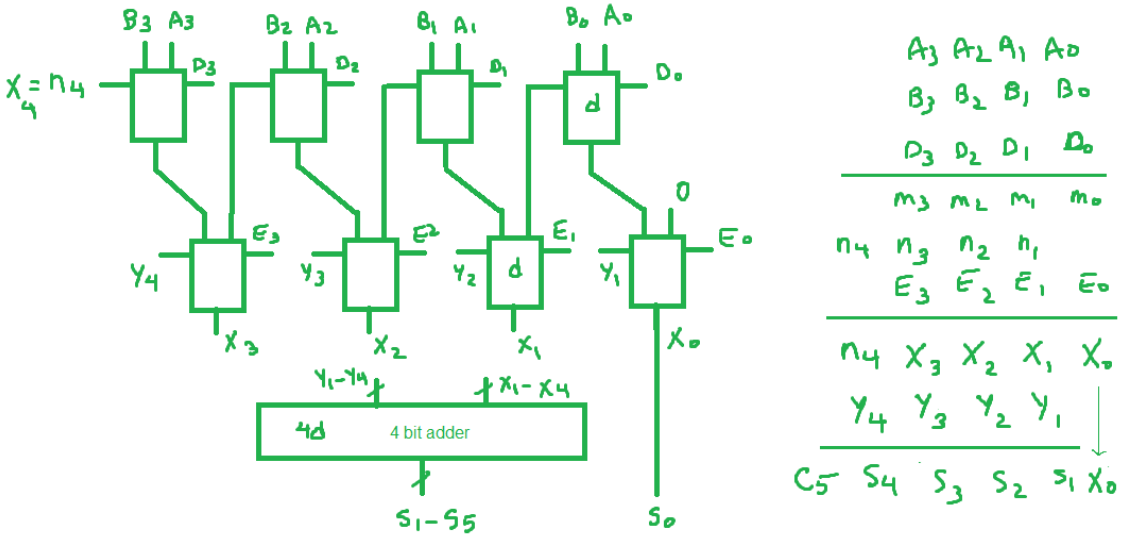
به عبارت دیگر، بیت‌های جمع‌کنندهٔ ۵ بیتی پایینی، هر کدام فقط به اندازهٔ d دیرتر از جمع‌کننده‌های ۴ بیتی بالا آماده می‌شوند، بنابراین تاخیر نهایی این روش $6d$ است.

حال، اگر جمع‌کننده‌ها از نوع CLA باشد، همهٔ بیت‌های X و Y همزمان (بعد از تاخیر یک جمع‌کنندهٔ CLA) تولید می‌شوند و سپس به اندازهٔ تاخیر یک جمع‌کنندهٔ CLA دیگر طول می‌کشد تا بیت‌های Z_0-Z_3 تولید شوند. بیتِ پنجم (Z_4) یک تاخیر اضافه خواهد داشت که توضیح خواهیم داد. تاخیر CLA برابر با $3d$ خواهد بود:

$$a_i, b_i \xrightarrow{d} p_i, g_i \xrightarrow{d} c_i \xrightarrow{d} s_i$$

بنابراین بیت‌های Z_0-Z_3 بعد از $6d$ حاضر می‌شود. برای محاسبهٔ بیت Z_4 باید بیتِ نقلی پیش از آن حاضر شود و سپس $2d$ صرف کنیم. یعنی بیت Z_4 پس از $3d + 2d + 2d = 7d$ حاضر می‌شود.

در روش دوم جمع طبق شکل زیر انجام می‌شود، بنابراین تاخیر نهایی این روش هم $6d$ است.



حال، اگر جمع‌کنندهٔ مرحلهٔ آخر از نوع CLA باشد، تاخیر کل برابر با $d + d + 3d = 5d$ خواهد بود.

بنابراین با به‌کارگیری جمع‌کننده‌های CLA، تاخیرِ روشِ اول بیشتر و تاخیر روشِ دوم کمتر می‌شود.

بارم‌بندی:

رسم شکل برای هر کدام از دو روش: ۲ نمره (در مجموع ۴ نمره)

محاسبهٔ تاخیر در روش اول، بدون CLA: ۳ نمره

محاسبهٔ تاخیر در روش اول، با CLA: ۳ نمره

محاسبهٔ تاخیر در روش دوم، بدون CLA: ۳ نمره

محاسبهٔ تاخیر در روش دوم، با CLA: ۲ نمره

اگر در روش اول تاخیر را برابر با تاخیر دو جمع‌کنندهٔ کامل درنظر بگیرند، فقط ۱ نمره داده شود.

اگر در روش اول با CLA، متوجه تاخیر اضافه در محاسبهٔ بیتِ آخر نباشند، ۱ نمره کسر شود.

اگر تاخیر CLA را اشتباه حساب کنند، فقط یک بار ۲ نمره کم شود.

۴- (۵ نمره) در روشی مشابه با استاندارد IEEE 754 برای نمایش اعداد ممیز شناور ۱۶ بیتی طول بخش‌های علامت (Sign)، نما (Exponent) و کسری (Fraction) به ترتیب یک، ۵ و ۱۰ بیت است.

دو عدد ۰٫۲ و ۰٫۰۵ را در این استاندارد نمایش دهید. سپس آنها را با طبقِ الگوریتمِ جمعِ اعدادِ ممیز شناور با هم جمع کنید. نتیجه را با این استاندارد نمایش دهید و معادلِ دهدهی آن را به دست آورید.

پاسخ:

$$0.2 = (0.001100110011)_2 = (1.1001100110)_2 \times 2^{-3}$$
$$0.05 = (0.00001100110011)_2 = (1.1001100110)_2 \times 2^{-5}$$

در این روش، چون ۵ بیت به نما اختصاص داده‌ایم، مقدار bias برابر با $۲^{۴} - ۱ = ۱۵$ است، بنابراین:

$$Exp(0.2) = (-3 + 15)_{10} = (12)_{10} = (01100)_2$$
$$Exp(0.0.5) = (-5 + 15)_{10} = (10)_{10} = (01010)_2$$

بنابراین دو عدد را به این شکل نمایش می‌دهیم:

$$(0.2)_{10} = 0\ 01100\ 1001100110$$
$$(0.0.5)_{10} = 0\ 01010\ 1001100110$$

برای جمع کردن این دو عدد باید توان‌های دو عدد را یکی کنیم، به این ترتیب که عدد دوم را به اندازهٔ اختلاف توان‌ها (۲) به سمت چپ شیفت بدهیم و سپس دو عدد را با هم جمع کنیم:

$$(0.2 + 0.05)_{10} = (1.1001100110)_2 \times 2^{-3} + (0.011001100110)_2 \times 2^{-3}$$
$$= (1.1111111111)_2 \times 2^{-3} = (2 - 2^{-10}) \times 2^{-3} = \frac{2}{8} - 2^{-13} \approx 0.25$$

می‌بینیم که پاسخ نهایی کمی کمتر از پاسخ درست و دقیق یعنی عدد ۰٫۲۵ است و علت این اختلاف این است که دو عدد ۰٫۲ و ۰٫۰۵ را نتوانستیم به صورت دقیق نماش دهیم.

نمایش نتیجه در این استاندارد به این شکل است:

$$(0.2 + 0.05)_{10} = 0\ 01100\ 1111111111$$

بارم‌بندی:

نمایش درست دو عدد هر کدام ۱ نمره

روش درست محاسبهٔ پاسخ نهایی: ۱ نمره

پاسخ نهایی در فرمت ممیز شناور استاندارد: ۱ نمره

پاسخ نهایی به صورت دهدهی: ۱ نمره

محاسبه نادرست bias: کسر ۱ نمره

۵- (۲۰ نمره) بلوک دیاگرامِ مسیر داده و کنترل پردازندهٔ ساده‌شدهٔ MIPS را در شکل ۱ می‌بینید. عملیاتِ کنترل در این شکل توسط تعدادی سیگنالِ کنترلی انجام می‌شود که در جداولِ ۱ و ۲ آمده است.

الف- مراحلِ اجرای دستورِ زیر را شرح دهید.

```
slt rd,rs,rt    ; if rs<rt rd=1, else rd=0
```

000000	rs	rt	rd	0	101010

پاسخ: (۵ نمره)

ابتدا دستور از روی حافظهٔ دستور واکشی (fetch) می‌شود. سپس شمارهٔ دو ثبات rs و rt به فایل ثبات داده می‌شود تا مقدار آنها در خروجی ثبات فایل به‌دست آید. همزمان واحد کنترل دستور را کدگشایی کرده و خطوط کنترلی را تولید می‌کند. چون دستور از نوع r-type است، ALUsrc=0 و ALUOp=10 است، بنابراین مقدار دو ثبات وارد ALU می‌شود و ضمنا ALU از روی بیت‌های صفر تا ۵ دستور می‌فهمد که باید slt انجام دهد. چون MemtoReg=0 است، نتیجهٔ ALU وارد فایل ثبات می‌شود و RegWrite=1 و RegDst=1 است، نتیجه روی rd نوشته می‌شود. ضمنا خطوط Branch، MemRead و MemWrite هم صفر هستند.

ب- می‌خواهیم دستوراتِ زیر را به مجموعهٔ دستوراتِ این پردازنده اضافه کنیم:

```
bgt rs,rt,offset    ; if rs>rt goto (PC+4)+(offet*4)
```

000110	rs	rt	16 bits offset

```
lui rt,cnst         ; rt(31-16) ← cnst
```

110000	0	rt	16 bits cnst

```
addd rs,rt,cnst     ; rs ← rs+rt+cnst
```

111000	rs	rt	16 bits cnst

چه تغییراتی باید در شکل و جداول بدهیم؟ مسیر دادهٔ اجرای این دستور را برمبنای شکل توضیح دهید و مشخص کنید برای اجرای این دستور، مقادیر هر یک از سیگنال‌های کنترلی زیر چه باید باشد؟

```
RegDst, Branch, MemRead, MemtoReg, MemWrite, ALUSrc, RegWrite, ALUOp
```

پاسخ: (هر دستور ۵ نمره)

برای اجرای دستور bgt باید محتوای دو ثبات rs و rt از روی ثبات فایل خوانده شود و وارد ALU شود. ALU باید rs-rt را محاسبه کند. ALU می‌تواند یک خروجی علامت (S) هم داشته باشد که اگر نتیجه منفی بود، یک شود. طبعا اگر rs بزرگتر باشد، Z=S=0 خواهد شد، بنابراین می‌توانیم Z و S را با هم OR کرده و نتیجه را با یک سیگنال خروجی از واحد کنترل که مخصوص این دستور است و مثلا می‌توانیم آن را BGT بنامیم، AND کنیم و BGTTrue بنامیم. ورودی انتخاب مالتی‌پلکسر بالا سمت راست را می‌توانیم با OR کردن BGTTrue و خط انتخاب فعلی بسازیم تا دستور bgt به درستی اجرا شود. بنابراین، سیگنال ALUsrc باید صفر باشد.

سایر سیگنال‌های کنترلی این مقادیر را دارند:

RegDst=MemtoReg=×, Branch=MemRead=MemWrite=RegWrite=0, ALUOp=01

برای اجرای دستور lui نیازی نیست محتوای هیچ ثباتی را بخوانیم، اما می‌توانیم مقدار ثبات rs را که صفر است بخوانیم و به ALU دستور جمع بدهیم و عملوند دوم ALU را هم برابر با مقدار ثابت قرار بدهیم (البته پس از ۱۶ شیفت به راست) که برای این کار باید ALUSrc=1 باشد. در این صورت کافی است ALUOp را برابر ۰۰ قرار بدهیم و MemtoReg را صفر کنیم که نتیجهٔ ALU در فایل ثبات نوشته شود. طبعا باید RegWrite=1 و RegDst=0 باشد.
برای پیاده‌سازی شیفتِ راست کار خاصی نیاز نیست انجام بدهیم اما بعد از واحد Sign-extend باید یک مالتی‌پلکسر بگذاریم که یکی از ورودی‌های آن خروجی واحد Sign-extend است و ورودی دیگر شیفت‌یافتهٔ مقدار ثابت است. یک سیگنال کنترلی هم باید در واحد کنترل تولید کنیم که هنگام اجرای دستور lui یک باشد که شیفت‌یافتهٔ عدد ثابت را به ALU بدهد. سایر سیگنال‌های کنترلی هم عبارتند از:
Branch=MemRead=MemWrite=RegWrite=0

اگر بخواهیم lui را طوری اجرا کنیم که محتوای ۱۶ بیت کم‌ارزش rt تغییر نکند، اجرای دستور lui کمی پیچیده‌تر می‌شود. چون در این صورت باید مقدار خود rt به ALU وارد شود و بیت‌های پرارزش آن با مقدار ثابت جایگزین شود. در این صورت باید در ALU یک عمل جدید تعریف شود که دقیقا همین کار را می‌کند (بیت‌های کم‌ارزش را حفظ می‌کند و بیت‌های پرارزش را جایگزین می‌کند)، بنابراین واحد کنترل باید یک ALUOp جدید (مثلا ۱۱) برای آن تعریف کند. ضمنا در ورودی اول ALU باید یک مالتی‌پلکسر اضافه کنیم که بین Read data1 و Read data2 یکی را انتخاب کند یا در ورودی Read Register 1 یک مالتی‌پلکسر قرار دهیم که بین بیت‌های ۲۱ تا ۲۵ و بیت‌های ۱۶ تا ۲۰ دستور یکی را انتخاب کند. چون خود ALU می‌داند که باید بخشی از بیت‌های عملوند دوم را در بایت‌های پرارزش عملوند اول قرار دهد، نیازی نیست تغییری در خروجی Sign-extend بدهیم. بقیهٔ بیت‌های کنترلی مثل حالت قبل خواهند بود.

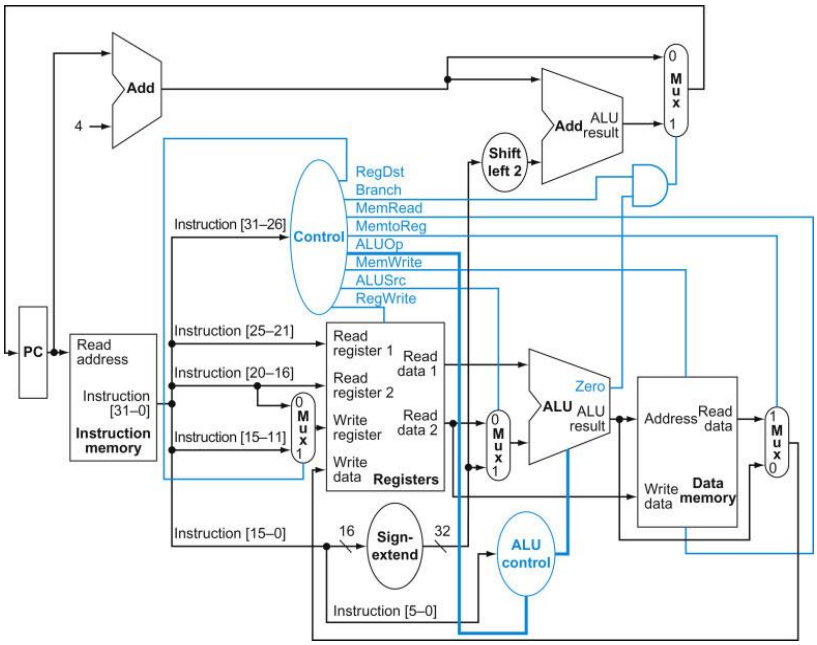
برای اجرای دستور addd باید سه عدد را با هم جمع کنیم. می‌توانیم تغییراتی در همین ALU بدهیم و سه ورودی برای آن بگذاریم و با ALUOp=11 برایش مشخص کنیم که باید هر سه ورودی را با هم جمع کند.
راه دیگر این است که در یکی از ورودی‌ها (یا خروجی) این ALU یک جمع‌کننده قرار بدهیم و خودمان دو تا از اعداد را با هم جمع کنیم. مثلا، حاصل‌جمع rt و مقدار ثابت را در آن جمع‌کننده به‌دست آوریم و با یک مالتی‌پلکسر به ورودی دوم ALU بدهیم که در خود ALU این حاصل‌جمع با rs جمع شود.
از طرفی چون نتیجه باید در rs نوشته شود، باید یک مالتی‌پلکسر دیگر به ورودی Write Reg اضافه کنیم که بتوانیم بیت‌های ۲۱ تا ۲۵ دستور را به آن بدهیم و طبعا سیگنال کنترلی این مالتی‌پلکسر هم باید در واحد کنترل تولید شود.
مقدار سیگنال‌های کنترلی برای اجرای این دستور عبارتند از:
RegDst =×, RegWrite=1, MemtoReg=RegDst=Branch=MemRead=MemWrite=0, ALUOp=01

جدول ۱- شرح ارتباط سیگنال‌های واحد ALU Control در شکل ۱

Instruction opcode	ALUOp	Instruction operation	Funct field	Desired ALU action	ALU control input
LW	00	load word	XXXXXX	add	0010
SW	00	store word	XXXXXX	add	0010
Branch equal	01	branch equal	XXXXXX	subtract	0110
R-type	10	add	100000	add	0010
R-type	10	subtract	100010	subtract	0110
R-type	10	AND	100100	AND	0000
R-type	10	OR	100101	OR	0001
R-type	10	set on less than	101010	set on less than	0111

جدول ۲- شرح ارتباط سیگنال‌های واحد Control در شکل ۱

Input or output	Signal name	R-format	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
	ALUOp0	0	0	0	1



شکل ۱- بلوک دیاگرام مسیر داده و کنترل پردازنده ساده MIPS

موفق باشید