به نام خدا

درس معماری کامپیوتر
نیم‌سال دوم ۰۴-۰۳
استاد: دکتر اسدی

دانشکده مهندسی کامپیوتر

پاسخنامه تمرین سری پنجم

---

۱. (آ) **swap $rs,$rt,imm**

برای این دستور میتوانیم یک مالتی پلکسر پشت ورودی اول ALU بذاریم که ورودی صفر آن از خروجی اول رجیستر فایل و ورودی یک آن از خروجی دوم رجیستر فایل می‌آید. برای سیگنال کنترلی آن نیز، یک سیگنال swap از Control unit خروجی میدهیم که حاصل دیکود شدن opcode این دستور در Control Unit است.
سپس باید یک مالتی پلکسر در پشت ورودی داده حافظه قرار دهیم که ورودی صفر آن از خروجی دوم رجیستر فایل و ورودی یک آن از خروجی اول رجیستر فایل می‌آید. سیگنال کنترلی آن را نیز همان swap قرار میدهیم. حال باید یک مالتی پلکسر در پشت ورودی آدرس رجیستر فایل قرار دهیم که ورودی صفر آن از خروجی مالتی‌پلکسر قبلی که در پشت ورودی آدرس قرار داشت می‌آید و ورودی یک آن نیز به بیت‌های ۲۱ تا ۲۵ دستور وصل می‌شوند.

Jump	Branch	ALUSrc	MemRead	regDst
۰	۰	۱	۱	x

swap	regWrite	ALUOp	MemWrite	MemToReg
۱	۱	۱۰	۱	۱

(ب) **addnz $rs,$rt,imm**

ابتدا خروجی دوم رجیستر فایل را به یک گیت OR میدهیم تا متوجه شویم که محتوای $rt مخالف صفر است یا خیر. همچنین از Control Unit یک سیگنال addnz خروجی میگیریم که از دیکود شدن Opcode به دست می‌آید . حال از یک مالتی پلکسر در پشت ورودی آدرس رجیستر فایل استفاده میکنیم که ورودی صفر آن از مالتی پلکسر قبلی و ورودی یک آن از بیت‌های ۲۱ تا ۲۵ دستور می‌آید. برای سیگنال کنترلی این مالتی پلکسر از سیگنال addnz استفاده میکنیم. برای ورودی سیگنال RegWrite، مقدار سیگنال RegWrite را با خروجی گیت OR، AND می‌کنیم.

Jump	Branch	ALUSrc	MemRead	regDst
۰	۰	۱	۰	x

addnz	regWrite	ALUOp	MemWrite	MemToReg
۱	۱	۱۰	۰	۰

(ج) **loadpc $rd**

در اینجا صرفا یک مالتی پلکسر در پشت ورودی داده رجیستر فایل قرار میدهیم که ورودی صفر آن خروجی مالتی پلکسر Write Back و ورودی یک آن نیز خروجی آن Adder است که محتوای PC را با ۴ جمع میکند.

Jump	Branch	ALUSrc	MemRead	regDst
۰	۰	x	۰	۱

regWrite	ALUOp	MemWrite	MemToReg
۱	x	۰	x

(د) brsumz $rs, $rt, offset

در اینجا یک سیگنال brsumz که حاصل دیکود شدن Opcode است را از Control Unit خروجی میگیریم و سپس آن را با سیگنال Zero که از ALU خروجی میگیریم، AND میکنیم و در نهایت خروجی آن را با سیگنال کنترلی که قبلا فرآیند Branch را مدیریت میکرد، OR میکنیم.

Jump	Branch	ALUSrc	MemRead	regDst
۰	۰	۰	۰	x

brsumz	regWrite	ALUOp	MemWrite	MemToReg
۱	۰	۱۰	۰	x

۲. (آ) زمان چرخه ساعت در پردازنده باید به اندازه‌ای باشد که طولانی‌ترین عملیات را در خود جا بدهد. در پردازنده MIPS دستور lw طولانی‌ترین زمان را برای اجرا شدن صرف می‌کند. بنابراین زمان چرخه ساعت پردازنده باید به اندازه زمان اجرا شدن دستور lw باشد.
برای اجرای دستور lw، ابتدا I-Mem و سپس Register File مورد دسترسی قرار می‌گیرد. در حین دسترسی به Register-File، تاخیرهای Mux + Sign-Extend و ALU-Control سپری می‌شود و دیگر نیازی به محاسبه این تاخیرها نیست. سپس تاخیر ALU را نیز اضافه می‌کنیم. در ادامه به D-MEM و سپس به یک Mux بر می‌خوریم. در نهایت باید تاخیر Register File را نیز اضافه کنیم. بنابراین زمان چرخه ساعت پردازنده برابر می‌شود با:

$$I - Mem + Regs + ALU + D - Mem + Mux + Regs$$

$$= 500 + 220 + 180 + 1000 + 100 + 220 = 2220ns$$

(ب) سیگنال MemWrite پس از fetch شدن دستور می‌تواند تولید شود و تا قبل از پایان چرخه ساعت باید تولید شده باشد. بنابراین سیگنال MemWrite باید در بازه [۲۲۲۰، ۵۰۰] نانوثانیه تولید شود.

(ج) تمامی سیگنال‌های کنترلی پس از fetch شدن دستور تولید می‌شوند. سیگنال‌هایی که در انتهای مسیر اجرای یک دستور هستند، بیشترین زمان برای تولید شدن را خواهند داشت. بنابراین سیگنال‌های MemWrite و RegWrite که تنها در آخر چرخه ساعت مورد نیاز هستند، بیشترین زمان را دارند که این بازه زمانی در بخش قبل محاسبه شده است.

(د) تولید سریع اولین سیگنالی که در مسیر اجرای یک دستور قرار می‌گیرد ضروری است. بنابراین لازم است سیگنال ALUSrc به عنوان اولین سیگنال کنترلی در مسیر داده، سریع‌تر از سایر سیگنال‌ها تولید شود. این سیگنال پس از fetch شدن دستور تا ۱۰۰ نانوثانیه قبل از شروع عملیات ALU باید آماده شود پس:
۵۰۰ + ۲۲۰ - ۱۰۰ = ۶۲۰
بنابراین سیگنال ALUSrc باید در بازه [۶۲۰، ۵۰۰] نانوثانیه تولید شود.

۳. (آ) دستور از نوع I-Type است.

(ب) مطابق نرم دستورات Rs و Rt وارد ALU می‌شوند. کافی‌ست $ALUSrc = 0$ قرار بگیرد تا این دو وارد ALU شوند.

برای اینکه ALU عملیات تفریق انجام دهد باید $ALUOp = 01$ قرار دهیم.

برای تعیین مقدار بعدی PC باید یک سیگنال کنترلی جدید مثل Branch برای beq ایجاد کنیم که تعیین کند در حالت دستور forr هستیم. اسم آنرا BNE می‌گذاریم. خروجی Zero از ALU با این سیگنال AND می‌شود. سپس این سیگنال را به بیت انتخاب‌گر مالتی‌پلکسری که بین $PC + 4$ و branch PC انتخاب می‌کند وصل می‌کنیم. دقت کنید از قبل سیگنالی حاصل از AND شدن سیگنال‌های Branch و Zero به این انتخاب‌گر وصل است. کافیست هردوی این‌ها را بصورت OR شده وصل کنیم.

از طرفی باید خروجی ALU روی Rs نوشته شود پس باید یک مالتی‌پلکسر دیگر به Write Register اضافه کنیم و با استفاده از سیگنال BNE که از قبل داشتیم کنترل کنیم که Rs تبدیل به write register شود.

سیگنال‌های کنترلی:

RegDst = X, ALUSrc = 0, MemtoReg = 0, RegWrite = 1, MemRead = 0, MemWrite = 0, Branch = 0, ALUOp = ”Sub”, bne = 1

۴. (آ) دستور lui (Load Upper Immediate) در معماری MIPS به این صورت است که عدد ثابت ۱۶ بیتی را در ۱۶ بیت بالایی رجیستر مقصد قرار می‌دهد و ۱۶ بیت پایین را صفر می‌کند، بدون آنکه نیازی به عملیات حافظه باشد.

(ب) جدول بدین شکل می‌شود:

Instr	RegDst	ALUSrc	Mem toReg	Reg Write	Mem Read	Mem Write	Branch	ALUOp ۱	ALUOp ۲	سیگنال جدید
R-type	۱	۰	۰	۱	۰	۰	۰	۱	۰	۰
lw	۰	۱	۱	۱	۱	۰	۰	۰	۰	۰
sw	x	۱	x	۰	۰	۱	۰	۰	۰	x
beq	x	۰	x	۰	۰	۰	۱	۰	۱	x
lui	۰	x	x	۱	۰	۰	۰	x	x	۱

۵. ۱. RegDst

(آ) stuck-at-0
امکان اجرای دستورات R-type از بین می‌رود. چرا که ثبات مقصد به درستی مشخص نمی‌شود.

(ب) stuck-at-1
ثبات مقصد در دستورات لود به درستی مشخص نمی‌شود.

۲. Jump

(آ) stuck-at-0
امکان اجرای دستور پرش از بین می‌رود.

(ب) stuck-at-1
پردازنده پیوسته با توجه به دستور در حال اجرا پرش انجام می‌دهد. آدرسی که برای پرش انتخاب می‌شود انکود شده بخش‌های یک دستور دیگر هستند. لذا این اتفاق همه دستورات بجز پرش را دچار مشکل می‌کند.

۳. Branch

(آ) stuck-at-0
امکان اجرای دستورات برنچ از بین می‌رود.

(ب) stuck-at-1
به ازای برخی دستورات مانند sub که خروجی zero واحد ALU را فعال می‌کنند برنچ ناخواسته صورت می‌گیرد. لذا این اتفاق می‌تواند بر تمامی دستورات دیگر با توجه به وضعیت سیگنال‌های دیگر و ماشین کد داده شده اثر بگذارد.

۴. MemRead

(آ) stuck-at-0
امکان خواندن از حافظه از بین می‌رود. لذا دستور لود دچار مشکل می‌شود.

(ب) stuck-at-1
این اتفاق آسیب منطقی نمی‌زند، ولی به ازای تمامی دستورات از حافظه عملیات خواندن انجام می‌دهد که سربار ایجاد می‌کند و همچنین ممکن است باعث افزایش miss-rate داخل حافظه نهان شود.

۵. MemToReg

(آ) stuck-at-0
امکان انتقال داده خوانده شده از حافظه به رجیستر از بین رفته و عملا دستور لود نخواهیم داشت.

(ب) stuck-at-1
امکان اجرای دستوراتی که خروجی ALU را در ثبات می‌ریزند، یا به عبارتی دستورات R-type از بین می‌رود.

۶. MemWrite

(آ) stuck-at-0
امکان اجرای دستور store از بین می‌رود.

(ب) stuck-at-1
همواره مقادیر خروجی ALU در آدرس رندومی (در واقع این آدرس رندوم بخش انکود شده یک دستور نامرتبط است) نوشته می‌شود. این اتفاق باعث می‌شود عملکرد هر دستوری بجز دستورات استور آسیب زننده باشد.

۷. ALUSrc

(آ) stuck-at-0
در دستورات لود و استور که آفست داریم، این اختلال باعث می‌شود از مقدار داخل ثبات دوم استفاده شود که مقداری رندوم با توجه به مقدار آفست انکود شده است.

(ب) stuck-at-1

در دستورات برنچ و R-type که هر دو ورودی ALU از بانک ثبات می‌آیند، اشتباها ورودی دوم به جای بانک ثبات از بخش آفست گرفته می‌شود که ایراد منطقی دارد و با توجه به دستور انکود شده یک عملکرد ناخواسته انجام می‌دهد.

۸. RegWrite

(آ) stuck-at-0
امکان نوشتن در بانک ثبات از دست می‌رود و دستورات لود و R-type از کار می‌افتند.

(ب) stuck-at-1
همواره مقداری در حافظه نوشته می‌شود که این باعث می‌شود مقادیر رندومی در دستورات استور و برنچ به صورت ناخواسته در بانک ثبات ریخته شود.

۹. PCSrc

(آ) stuck-at-0
در اجرای دستورات branch به مشکل می‌خوریم چون هیچگاه آدرس پرش انتخاب نمی‌شود.

(ب) stuck-at-1
در اجرای دستوراتی که branch نیستند به مشکل می‌خوریم و به آدرس‌های رندوم می‌پریم.

۶. (آ) سیگنال‌های کنترلی به شرح زیر است:

inst	PCSrc	ALUSrc	ALUOp	MemWrite	MemRead	MemToReg	RegDst	RegWrite
lwd	0	0	add	0	1	1	1	1

شکل ۱: سیگنال‌های کنترلی

(ب) از آنجا که single-cycle است، داریم $CPI = 1$ و همچنین clock cycle time = latency of lowest instruction، که در اینجا lw کندترین دستور است.

$lwlatency = 5 + 3 + 4 + 5 + 3 = 20ns$

$totallatency = 5 * 20ns = 100ns$

(ج) اگر ALU به اندازه ۲۵ درصد سریعتر باشد، یعنی تاخیر آن 3.2ns خواهد بود که زمان یک کلاک (تاخیر دستور lw) مقدار 19.2ns می‌شود که یعنی:

$speedup = \frac{20}{19.2}$