هو العلیم

اصول سیستم‌های کامپیوتری

نیمسال اول سال تحصیلی ۱۴۰۲ – ۱۴۰۱

تمرینات ۳

مریم رضائی

۱. این برنامه بازگشتی را که به زبان C نوشته شده است و n اُمین عنصر دنباله فیبوناچی را محاسبه می‌کند، به برنامه‌ای به زبان RISC-V ترجمه کنید. خودتان شماره رجیسترهایی را که در برنامه RISC-V استفاده می‌کنید طبق قراردادها تعیین کنید.

جواب:

برای تبدیل این کد به RISC-V نیاز به ۲ رجیستر داریم؛ رجیستر x10 (از رجیسترهای پارامتر تابع) برای ذخیره ورودی تابع و در همان راستا تغییر یافتن برای تبدیل به خروجی تابع، و رجیستر x5 (از رجیسترهای موقتی) برای اعمال مقایسه و همچنین ذخیره‌ی مقدار x10 خروجی قبلی زمانی که آن را برای فراخوانی بازگشتی بخش بعد تغییر می‌دهیم. لازم به ذکر است که هر بار که تابع برای x10 بزرگ‌تر از ۱ فراخوانی می‌شود، آدرس پیشین فراخوانی بازگشتی و مقدار قبلی ورودی نیاز به ذخیره سازی در پشته دارند، تا بتوانیم در صورت بازگشت‌های تو در تو به ترتیب با آدرس‌ها از آخر به اول برگردیم و هر بار n ورودی قبل را برای فراخوانی دوباره بازیابی کنیم و خروجی تا به حال یافت شده را ذخیره کنیم. با توجه به این ملاحضات و مقدمات، برنامه RISC-V را با پیاده‌سازی زیر داریم:

۱) ابتدا ورودی را با صفر مقایسه می‌کنیم و در صورت برابری، به return می‌رویم و همان x10 ورودی بازگردانده می‌شود. در غیر این صورت ادامه می‌دهیم.

۲) رجیستر x5 را برای مقایسه ۱ می‌گذاریم و در صورت برابری با ورودی x10 به return می‌رویم و همان x10 ورودی بازگردانده می‌شود. در غیر این صورت ادامه می‌دهیم.

۳) در پشته به اندازه‌ی دو کلمه فضا باز کرده و ابتدا آدرس قبل بازگشت و سپس ورودی حال حاضر را ذخیره می‌کنیم، زیرا در ادامه فراخوانی بازگشتی داشته و آدرس تغییر خواهد کرد، و برای آن فراخوانی ورودی نیز کوچکتر می‌شود در حالی که این ورودی را برای فراخوانی بازگشتی دوم در همین سطح (یعنی یکبار برای fib(n – 1) و یکبار برای fib(n – 2) نیاز داریم.

۴) از x10 ورودی یک واحد برای فراخوانی بازگشتی اول کم می‌کنیم و دوباره به شروع تابع fib برای این ورودی پرش می‌کنیم. لازم به ذکر است که آدرس حال حاضر در x1 ذخیره شده (همان طور که گفتیم x1 تغییر کرد) و پس از اتمام این فراخوانی (یعنی زمانی که فراخوانی‌های تو در تو برای ورودی تا زمانی که به ۱ برسد به پایان رسید و هر بار آدرس‌ها در پشته گذاشته شد و برداشته شد) به این خط باز می‌گردیم و ادامه می‌دهیم.

۵) حال x10 قبل (یعنی n همین سطح که برای فراخوانی fib(n – 1) از آن استفاده کردیم) را بازیابی کرده و در x5 میریزیم و در عوض خروجی x10 (یعنی مقداری که فراخوانی fib(n – 1) این رجیستر را به آن تبدیل کرد) برای جمع کردن نهایی با نتیجه‌ی فراخوانی بازگشتی دوم ((fib(n – 2) + fib(n – 1)) در پشته ذخیره می‌کنیم.

۶) از x5 که حاوی n ورودی این سطح است دو واحد کم کرده و برای فراخوانی بازگشتی دوم با ورودی n – 2 در رجیستر x10 می‌ریزیم. سپس باز به شروع تابع برای محاسبه‌ی fib(n – 2) می‌رویم. (به مانند قبل، آدرس حال

حاضر در x1 ذخیره شده و پس از اتمام این فراخوانی (یعنی زمانی که فراخوانی‌ها برای ورودی تا زمانی که به ۰ برسد به پایان رسید و هر بار آدرس‌ها در پشته گذاشته شد و برداشته شد) به این خط باز می‌گردیم.

۷) حال x10 حاوی حاصل fib(n – 2) است (یعنی فراخوانی بازگشتی برای ورودی n – 2 رجیستر را هر بار تغییر داده تا در نهایت حاوی حاصل فراخوانی باشد) و حاصل fib(n – 1) را که در مرحله‌ی ۵ در پشته ذخیره کرده بودیم بازیابی کرده و در x5 می‌ریزیم.

۸) بنابراین x10 = fib(n – 2) و x5 = fib(n – 1) پس آن‌ها را با هم جمع کرده و برای حفظ حاصل آن را در خود x10 که پارامتر تابع است ذخیره می‌کنیم. (در اینجاست که می‌گوییم هر بار تابع مقدار خود رجیستر x10 را به مقدار خروجی تغییر می‌دهد، به جز حالات برابری ورودی با ۰ و ۱ که در آن حالت باز خروجی برابر با مقدار ورودیست که در منها کردن‌های سطح های بالا تر به آن رسیدیم.)

۹) در نهایت آخرین آدرس بازگشت ذخیره شده (که از تغییرات x1 در زمانی فراخوانی fib(n – 1) و fib(n – 2) محفوظ مانده) را بازیابی می‌کنیم، پشته را خالی کرده، و به آدرس باز می‌گردیم.

کد RISC-V اصلاح شده‌ی حل‌المسائل کتاب (که در بالای پشته درج و حذف نمی‌کرد و برای محاسبه‌ی offset برای ذخیره‌سازی در پشته مشکل داشت) همراه با کامنت‌های کامل‌تر در زیر آورده شده است.

برنامه C	برنامه RISC-V
int fib(int n) { **if** (n == 0) **return** 0; **else if** (n == 1) **return** 1; **else** **return** fib(n−1) + fib(n−2); }	FIB: beq x10, x0, RET // if n == 0, return 0 addi x5, x0, 1 // set x5 = 1 for comparison beq x10, x5, ret // if n == 1, return 1 addi sp, sp, -8 // make room for 2 in stack space sw x1, 4(sp) // save the return address sw x10, 0(sp) // save the current n addi x10, x10, -1 // x10 = n – 1 jal x1, FIB // call fib(n – 1) lw x5, 0(sp) // load old n from the stack sw x10, 0(sp) // push fib(n – 1) onto the stack addi x10, x5, -2 // x10 = n – 2 jal x1, FIB // call fib(n – 2) lw x5, 0(sp) // x5 = fib(n – 1) add x10, x10, x5 // x10 = fib(n – 1) + fib(n – 2) lw x1, 4(sp) // load last saved return address addi sp, sp, 8 // empty two words from the stack RET: jalr x0, 0(x1) // return to call address

---

مراجع: حل‌المسائل کتاب منبع درس:

Computer Organization and Design (RISC-V Edition) – Patterson & Hennessy

۲. معماران RISC-V ، بعضی از دستورات ۳۲ بیتی را در قالب ۱۶ بیتی نیز تعریف کرده‌اند. مزیت استفاده از چنین دستوراتی که به آنها **دستورات فشرده**[1] گفته می‌شود، این است که باعث کاهش اندازه برنامه‌های RISC-V می‌شوند؛ و البته ممکن است که بتوان مبتنی بر چنین دستوراتی، رایانه‌هایی با هزینه کمتر یا مصرف انرژی کمتر یا کارایی بالاتر ساخت.

متناظر با بیشتر دستورات پایه‌ای RISC-V (نه همه آنها) یک دستور فشرده نیز تعریف شده است. معماران برای نمایش ۱۶ بیتی دستورات ۳۲ بیتی، چند چیز را در دستورات ۳۲ بیتی محدوده کرده‌اند: در هر دستور فشرده از یک یا دو رجیستر می‌توان استفاده کرد؛ از ۳ بیت برای شماره‌گذاری رجیسترها می‌توان استفاده کرد (تنها می‌توان از رجیسترهای x8 تا x15 استفاده کرد)؛ مؤلفه opcode دستورات ۲ بیتی است؛ بعضی از دستورات (مثل sub و and) یک مؤلفه ۲ بیتی به نام funct2 دارند؛ و اینکه مؤلفه imm دستورات عددی ۶ تا ۱۱ بیتی است.

در این جدول، تعدادی از دستورات پایه‌ای ۳۲ بیتی و دستورات فشرده ۱۶ بیتی معادل با آنها آورده شده است. دستورات فشرده با پیشوند .c شروع می‌شوند.

دستورات پایه‌ای ۳۲ بیتی	دستورات فشرده ۱۶ بیتی
add rd, rd, rs2	c.add rd, rs2
addi rd, rd, SignExt(imm)	c.addi rd, imm
addi rd, x0, SignExt(imm)	c.li rd, imm
sub rd, rd, rs2	c.sub rd, rs2
slli rd, rd, imm	c.slli rd, imm
lw rd, (ZeroExt(imm)*4)(rs1)	c.lw rd, imm(rs1)
sw rs2, (ZeroExt(imm)*4)(rs1)	c.sw rs2, imm(rs1)
beq rs1, x0, label	c.beqz rs1, label
jal x0, label	c.j label

می‌توان برنامه‌های RISC-V را با ترکیبی از دستورات فشرده و غیرفشرده نوشت. شما این برنامه‌ای را که به زبان C نوشته شده است، با ترکیبی از دستورات فشرده و غیرفشرده به برنامه‌ای به زبان RISC-V تبدیل کنید. یعنی در صورتی که متناظر با یک دستور ۳۲ بیتی، یک دستور ۱۶ بیتی تعریف شده باشد و بتوان از آن استفاده کرد، از آن دستور ۱۶ بیتی استفاده کنید نه از دستور ۳۲ بیتی.

[1] **compressed instructions**

فرض کنید که آرایه array2 قبل از آنکه در حلقه استفاده شود، مقداردهی اولیه شده باشد. خودتان شماره رجیسترهایی را که در برنامه RISC-V استفاده می‌کنید طبق قراردادها تعیین کنید.

جواب:

فرض می‌کنیم شروع آرایه‌های ۱ و ۲ به ترتیب در رجیسترهای x10 و x11 ذخیره شده باشند. از آنجا که قصد نوشتن برنامه‌ای فشرده است، برای ترجمه‌ی کد از C به RISC-V دقت می‌کنیم که کمترین تعداد دستورات را استفاده کنیم، حلقه دارای کمترین تعداد ممکن دستورات باشد، و بیشترین تعداد ممکن دستورات بتوانند به ۱۶ بیتی تبدیل شوند.

برای این کار، به جای شروع حلقه از i = 0 که یکی یکی پیش رود و تعبیه رجیستری دیگر حاوی ۲۰۰ که با آن مقایسه شود، حلقه را از ۲۰۰- شروع کرده، اضافه می‌کنیم، و هر بار با x0 مقایسه می‌کنیم. همچنین درون حلقه به جای استفاده از slli برای ۴ برابر کردن مقدار i، ذخیره آن در رجیستری دیگر تا i تغییر نکند، و در نهایت جمع آن با رجیستر حاوی شروع آرایه، به ازای هر اجرای حلقه تنها رجیستر حاوی شروع آرایه را با عدد ۴ جمع می‌کنیم؛ توجه می‌کنیم که جمع را پس از خواندن و نوشتن حافظه انجام می‌دهیم تا برای تکرار اول حلقه از عنصر صفر شروع شود.

لازم به ذکر است که با این رویه، رجیستر حاوی شروع آرایه تغییر کرده و در پایان حلقه بایستی ۲۰۰×۴ از آن کم کنیم تا شروع اصلی بازیابی شود. همچنین اعمال مقداردهی اولیه ۲۰۰- و در انتها جمع‌های ۸۰۰- دارای immediate هایی بزرگ هستند که استفاده از دستورات ۱۶ بیتی برای این خطوط را غیرممکن می‌کند، اما به علت بیرون بودن این ۳ دستور از حلقه، به تعداد زیاد تکرار نمی‌شوند. برنامه‌ی RISC-V در زیر قابل مشاهده است.

برنامه C	برنامه RISC-V
int i; int array1[200]; int array2[200]; . . . for (i = 0; i < 200; i = i + 1) array1[i] = array2[i];	addi x8, x0, -200 // set i = –200, can’t compress bc of imm LOOP: c.lw x9, 0(x11) // load array2[i] where i = 0 to 199 c.addi x11, 4 // move forward one word in array2 c.sw x9, 0(x10) // store in array1[i] where i = 0 to 199 c.addi x10, 4 // move forward one word in array1 c.addi x8, 1 // set i += 1 c.beqz x8, END // if i == 0 end loop c.j LOOP // else rerun loop END: addi x10, x10, -800 // reset x10 to beginning of array1 addi x11, x11, -800 // reset x11 to beginning of array2

مراجع:

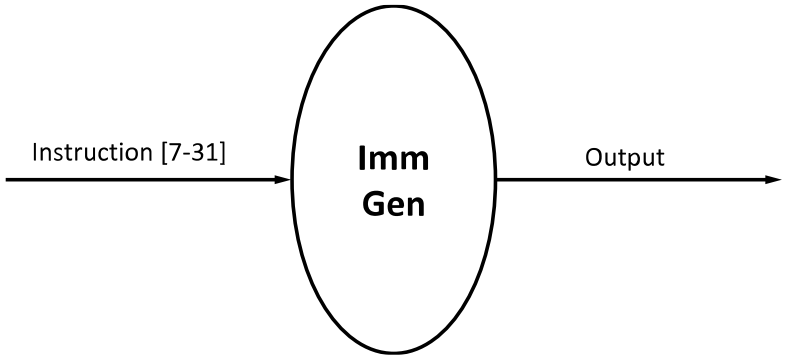
۳. یکی از واحدهای پردازنده‌ای که در فصل ۴ طراحی شده است، واحد Imm Gen است. واحد Imm Gen یک مدار منطقی است که از روی بیت‌های شماره ۷ تا شماره ۳۱ یک دستور و یک خط کنترلی از واحد کنترل، مقدار مؤلفه immediate دستور را با بیت علامتش بسط می‌دهد تا آن را ۳۲ بیتی کند. به بیان دقیق‌تر، واحد Imm Gen

- مؤلفه immediate ۱۲ بیتی دستور lw را ۳۲ بیتی می‌کند و آن را به عنوان یکی از دو ورودی به مالتی‌پلسکر ALUSrc می‌فرستد.
- مؤلفه immediate ۱۲ بیتی دستور sw را ۳۲ بیتی می‌کند و آن را به عنوان یکی از دو ورودی به مالتی‌پلسکر ALUSrc می‌فرستد.
- مؤلفه immediate ۱۲ بیتی دستور beq را ۳۲ بیتی می‌کند و آن را به عنوان یکی از دو ورودی به واحد adder راستی می‌فرستد.

واحد Imm Gen را به صورت یک مدار منطقی طراحی کنید. (از آنجا که این واحد کنترل است که باید خروجی واحد Imm Gen را مشخص کند، لازم است یک خط کنترلی از واحد کنترل به واحد Imm Gen کشیده شود تا طبق مقدار آن خط، واحد Imm Gen ، خروجی خود را تعیین کند. خط کنترلی را ImmSrc بنامید.)

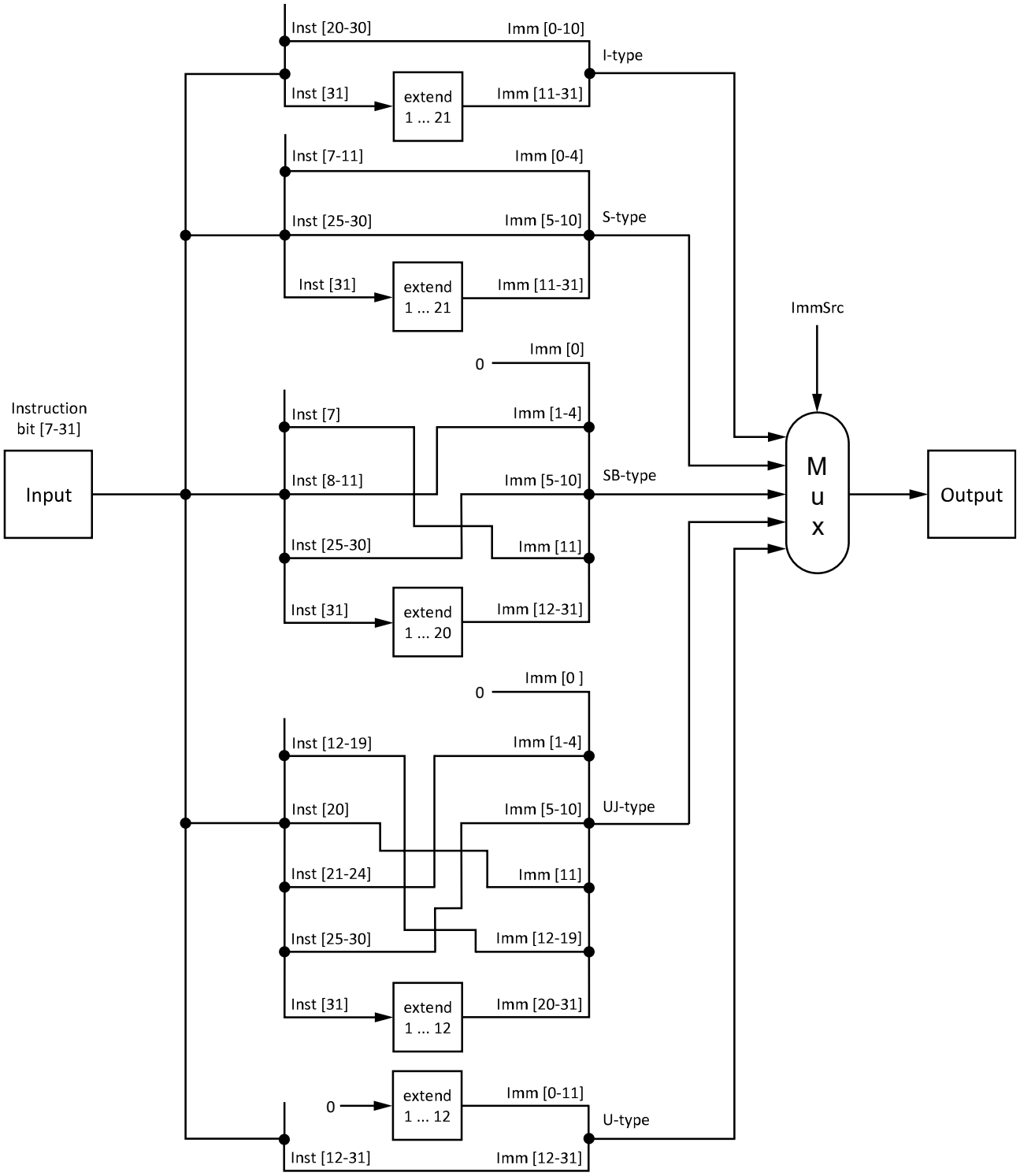
جواب:

برای طراحی یک واحد تولید عدد ثابت، ابتدا به ساختار کلی آن توجه می‌کنیم. در شکل زیر در نظر گرفته‌ایم بیت‌های ۷ تا ۳۱ دستور (یعنی به جز opcode که جدا به واحد کنترل می‌رود) ورودی واحد مورد نظر هستند. بایستی واحد به صورتی طراحی شود که دستور را برای ۵ نوع دستور دارای immediate شکسته و ۵ immediate ممکن را تولید کند، سپس با توجه به خط کنترلی نوع دستور را تعیین و immediate تولید شده مطابق ساختار آن را انتخاب کند.



بنابراین، با توجه به ساختار immediate دستورهای I و S و SB و UJ و U، بیت‌های ورودی را شکسته و برای ۴ نوع دستور اول چپ‌ترین بیت حاصل را به تعداد مورد نیاز تا ۳۲ بیتی شدن حاصل با واحد extend گسترش داده و تکرار می‌کنیم (مثلاً برای دستورات I بیت آخر را از ۱ به ۲۱ بیت گسترش می‌دهیم) و برای دستورات U نیز ۰ را راست‌ترین

بیت قرار داده و به همان شکل تکرار می‌کنیم. در نهایت، ۵ دسته خط حاصل را به مالتی‌پلکسر برده و با خطوط ImmSrc از واحد کنترل از میان آن‌ها انتخاب می‌کنیم. مدار منطقی این طراحی واحد، در زیر بعد درج شده است.



در ساخت مالتی‌پلکسر دقت می‌کنیم که برای انتخاب میان ۵ دسته خط حاصل، نیاز به ۳ بیت انتخاب کننده از واحد کنترل داریم زیرا $3 = \lceil \log_2 5 \rceil$؛ یعنی تعداد خطوط ImmSrc باید سه عدد باشند و بر اساس ترکیب مقادیر آن‌ها

چهارتا از immediateهای حاصل را صفر کرده و یکی را نگه داریم. جدول مالتی‌پلکسر در زیر قابل مشاهده است؛ می‌بینیم در ۳ ترکیب خطوط کنترلی به خروجی کاری نداشته و به عبارتی معادل زمانی هستند که دستور R-type باشد و واحد تولید عدد ثابت برای ما مهم نباشد (زیرا ALUSrc صفر است و مالتی‌پلکسر رجیستر را انتخاب می‌کند).

ImmSrc0	ImmSrc1	ImmSrc2	Output
0	0	1	I
0	1	0	S
1	0	0	SB
0	1	1	UJ
1	1	0	U
1	0	1	x
1	1	1	x
0	0	0	x

البته این مالتی‌پلکسر بزرگ از مالتی‌پلکسرهای کوچک ۲ در یک (یعنی دو بیت ورودی و یک بیت خروجی) ساخته شده است پس در اصل هر بیت کنترلی عمل مورد نظر را جدا جدا بر تک تک بیت‌های حاصل اعمال می‌کند. به عبارت دیگر منظور از انتخاب یک حاصل و صفر کردن بقیه، انجام عمل یکسان بر هر ۳۲ بیت آن حاصل است.

مراجع:

https://5hdaniel.github.io/cpu.html

۴. پردازنده‌ای که در فصل ۴ طراحی شده است، می‌تواند هفت دستور از مجموعه دستورات RISC-V را اجرا کند:

- دسته ۱ : دو دستور مراجعه به حافظه lw (load word) و sw (store word) .
- دسته ۲ : دو دستور حسابی add و sub ؛ و دو دستور منطقی and و or .
- دسته ۳ : دستور پرش شرطی beq (branch if equal) .

ما می‌خواهیم پردازنده را به نحوی بسط دهیم که بتواند دستورات دیگری را نیز اجرا کند. این چهار دستور RISC-V را در نظر بگیرید:

- andi
- xor
- bne
- sll

مشخص کنید که برای اجرای هر یک از این چهار دستور، کدام واحدهای سخت‌افزاری موجود را و دقیقاً چگونه باید آنها را بازطراحی کنیم.

جواب:

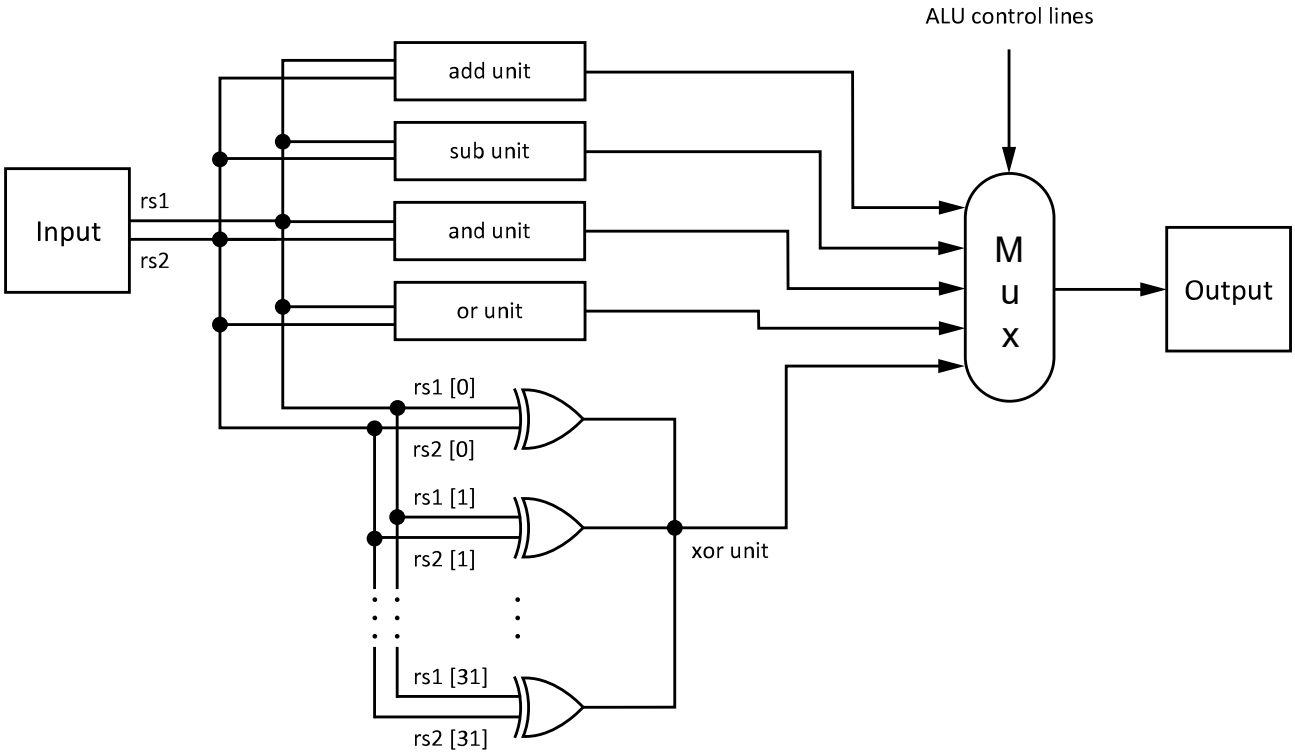
اضافه کردن هر دستور را جدا بررسی می‌کنیم؛ یعنی می‌بینیم نزدیک‌ترین دستور به دستور مورد نظر که پردازنده توان اجرای آن را دارد، چیست و بر آن اساس پردازنده نیاز به چه تغییری برای بسط داده شدن دارد.

- **دستور andi**: نزدیک‌ترین دستور به آن دستور and است که سخت افزار به کمک واحد ALU اجرا می‌کند. پس برای اضافه کردن andi سخت افزار مثل and عمل می‌کند و تنها سه نکته مورد توجه است؛ (۱) واحد Imm Gen به واحدهای فعال اضافه شده و با طراحی که در تمرین قبل نشان داده شد مقدار عدد ثابت را شکل می‌دهد، (۲) پیش از ورودی دوم ALU مالتی‌پلکسر ۳۲ بیت مقدار ورودی را از میان مقدار رجیستر و مقدار حاصل از Imm Gen انتخاب می‌کند، (۳) برای این کار واحد کنترل با بررسی دستور تعیین می‌کند ورودی دوم از عددی ثابت است، و با تغییر خط کنترلی ALUSrc از 0 به 1 این امر را به مالتی‌پلکسر رسانده و انتخاب انجام می‌شود. در ادامه، عمل ALU تعیین شده توسط ALUOp (جمع) و سپس ذخیره‌سازی یکسان است.
- **دستور xor**: نزدیک‌ترین دستور به آن or و and هستند که پردازنده به وسیله‌ی واحد ALU محاسبه می‌کند، پس برای اضافه کردن xor نیاز است این واحد را بسط دهیم. یعنی به طور کلی، پردازنده در ابتدا به مانند دستور or و and شروع به کار کرده با همان مقادیر واحد کنترل پیش می‌رود تا به ALU برسد. در اینجا عمل تعیین شده برای ALU توسط ALUOp برای تمامی دستورات نوع R از جمله xor یکسان است، پس پردازنده نیاز به تغییری در بخش واحد کنترل ندارد. سپس به عوامل دارای نیاز به تغییر می‌رسیم:

(۱) واحد کنترل ALU بایستی خط کنترلی ورودی ALU را با توجه به دستور جدید تغییر دهد، یعنی در جدول زیر (از شکل ۴.۱۲ کتاب) که خط کنترلی ALU را بر اساس func3 و func7 برای چهار دستور پیشین داریم سطری جدید اضافه می‌کنیم.

ALU control lines	Function
0000	AND
0001	OR
0011	XOR
0010	ADD
0110	SUB

(۲) داخل واحد ALU تغییر می‌کند. یعنی علاوه بر بخش‌های add، subtract، and و or بخش جدیدی برای xor به آن اضافه می‌شود و در آخر خروجی توسط مالتی‌پلکسر به کمک خطوط کنترلی از میان آنها انتخاب می‌شود. دقت می‌کنیم که مانند تمرین پیشین مالتی‌پلکسر در اصل از مالتی‌پلکسرهای کوچک با ۲ بیت ورودی و ۱ بیت خروجی ساخته شده است، پس یعنی عمل صفر کردن یا تغییر ندادن تعیین شده توسط هر بیت خط کنترلی بر تک تک ۳۲ بیت خروجی پنج بخش واحد ALU اعمال می‌شود. شکل درون ALU در زیر قابل مشاهده است.



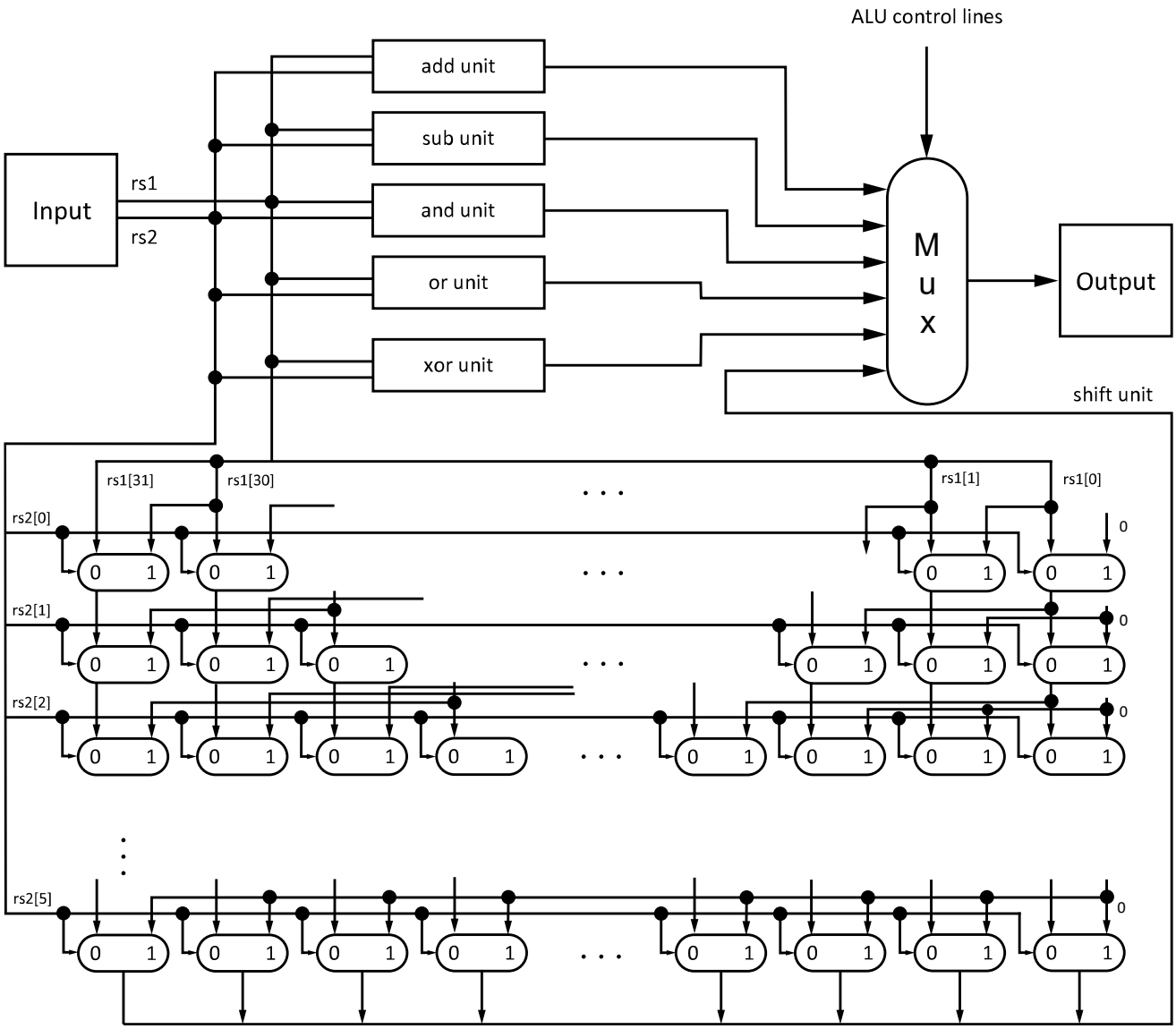
- **دستور bne**: نزدیک‌ترین دستور به آن beq است و به کمک واحد ALU انجام می‌شود؛ به این صورت که با تفریق دو ورودی از هم در صورت برابری (صفر شدن حاصل) Zero status واحد (یعنی صفری حاصل واحد) برابر یک شده و بالعکس. این status از ALU به گیت and بالایی ارسال می‌شود تا در نهایت در مالتی‌پلکسر تعیین کنیم جواب کدام adder (یعنی آن که PC+4 را محاسبه کرده و برای عدم برقراری شرط است، یا آنکه PC را با مقدار مورد نظر برای پرش جمع کرده است) انتخاب شود و در نهایت در PC جایگزین شود.

برای بسط این فرآیند به bne، (بنابر جدول شکل ۴.۱۲ کتاب) نیاز است خط کنترلی ALUOp مانند beq برابر 01 باشد و خطوط کنترلی ALU همان 0110 را به ALU بفرستند تا پاسخ عمل تفریق برای خروجی انتخاب شود. اما این بار باید عکس Zero status به گیت and ارسال شود، زیرا bne عکس beq است و در صورت برابری (صفر شدن حاصل تفریق و یعنی یک شدن Zero status) نباید عمل شاخه‌شدن انجام شود. بنابراین نیاز به خط کنترلی جدیدی از واحد کنترل ALU داریم که بنابر func3 دستور bne بودن آن را تشخیص دهد و به کمک مالتی‌پلکسری تعیین کنیم آیا خود خروجی Zero status یا not آن به گیت فرستاده شود تا در نهایت در صورت یک بودن این مقدار و مقدار خط Branch از واحد کنترل، حاصل adder دوم (یعنی مقدار برای PC در صورت شاخه‌شدن) انتخاب شود و در غیر این صورت PC+4.

- **دستور sll**: برای این دستور نیز بایستی دقیقا مثل xor واحد ALU را بسط دهیم. یعنی از آنجا که دستور همچنان R-type است، کارکرد کلی، واحد کنترل اصلی و مقدار ALUOp تغییری نمی‌کند، اما در کنترل و کار خود ALU تغییر داریم. به طور دقیق‌تر ابتدا واحد کنترل ALU و مقدار خطوط کنترلی آن تغییر کرده و بر اساس func3 و func7 جدول قبل را برای مثال به شکل زیر بسط می‌دهیم.

ALU control lines	Function
0000	AND
0001	OR
0011	XOR
0010	ADD
0110	SUB
0111	SHIFT

آنگاه در ساختار واحد ALU نیز shift را اضافه می‌کنیمتا در نهایت از میان شش حاصل بخش‌های ALU به کمک مالتی‌پلکسر انتهای ALU و خطوط کنترلی آن، حاصل بخش shift را انتخاب کنیم. برای این کار از یک barrel shifter استفاده می‌کنیم که ترکیبیاتی‌ست، سرعت بالایی داشته و از مالتی‌پلکسرها ساخته شده است و برای استفاده در ALU رایج‌ترین گزینه است. شکل تغییر یافته‌ی ALU در صفحه‌ی بعد قابل مشاهده است.



در barrel shifter تعداد سطوح جابه‌جایی‌ها را بر اساس تعداد بیت‌های عدد دوم تعیین می‌کنیم و تعداد جابه‌جایی در هر سطح برابر با چندمین بیت از راست بودن آن است، یعنی برای مثال برای انجام ۵ جابه‌جایی (یعنی دودویی 101) یک جابه‌جایی یک‌تایی در سطح اول و یک جابه‌جایی چهارتایی در سطح سوم نیاز داریم. در هر سطح نیز ۳۲ مالتی‌پلکسر (به تعداد بیت‌های عدد اول) داشته و ورودی مالتی‌پلکسرهای سطوح پایین‌تر بر اساس سطوح بالاتر تعیین می‌شود؛ پس با توجه به اینکه عدد ۳۲ بیتی در بیشترین حالت ۳۲ جابه‌جایی می‌تواند داشته باشد که در آخری تمامی بیت‌ها صفر می‌شود و همچنین $۳۲ = ۲^۵$ پس نیاز به شش سطح مالتی‌پلکسر داریم که در سطح آخر عدد صفر به عنوان یکی از انتخاب‌های تمامی مالتی‌پلکسرها قرار می‌گیرد.

مراجع:

Computer Organization and Design (RISC-V Edition) – Patterson & Hennessy

۵. الف) معماری x86 نمونه‌ای از یک معماری CISC است. مهم‌ترین وجوه تفاوت معماری x86 با معماری‌هایی مثل ARM و MIPS و RISC-V که نمونه‌هایی از معماری RISC هستند را مشخص کنید.

ب) مهم‌ترین وجوه تشابه و مهم‌ترین وجوه تفاوت مجموعه دستورات x86 با مجموعه دستورات RISC-V را مشخص کنید. دو معماری را از نظر تعداد رجیسترها، تعداد عملوندهای دستورات، مکان‌های عملوندها (ثابت و در خود دستور یا در رجیستر یا در حافظه) ، اندازه عملوندها، نوع دستورات (ساده یا پیچیده) و اندازه دستورات با هم مقایسه کنید.

پ) وجه تسمیه معماری x86 چیست؟ معماری x86 در دو دهه گذشته برای ساخت چه نوع رایانه‌هایی مورد استقبال شرکت‌های سازنده رایانه قرار گرفته است و به چه دلائلی هنوز هم مورد توجه است؟ در حال حاضر، برجسته‌ترین شرکت‌های سازنده پردازنده‌های x86 کدامند و برجسته‌ترین پردازنده‌های x86 چیستند؟

ت) معماری x86 برای ساخت چه نوع رایانه‌هایی و به چه دلائلی مورد استقبال شرکت‌های سازنده رایانه قرار نگرفته است؟ ممکن است که در آینده، سازندگانی که در حال حاضر برای ساخت رایانه از معماری x86 استفاده می‌کنند، معماری RISC-V را جایگزین معماری x86 کنند؟

جواب:

الف) x86 یک معماری با دستورات پیچیده (CISC) است در حالی که ARM، MIPS و RISC-V معماری‌های کم دستور (RISC) هستند. تفاوت این دو سبک معماری در تمرکز بر دستورات قدرتمند و در عین حال کندتر در برابر دستورات ساده‌تر و در عین حال سریع‌تر است. در معماری x86، هدف کاهش تعداد دستورات اجرایی برنامه است و دستورات توان بیشتری دارند. این امر سادگی دستورات را که ARM، MIPs و RISC-V دارند از بین می‌برد و باعث طولانی‌تر شدن زمان اجرای برنامه به علت دور ساعت‌های کندتر و یا تعداد دور ساعت‌های بیشتر، می‌شود. یعنی در معماری RISC ماشین‌ها یک دستور در یک دور ساعت اجرا می‌کنند اما در x86 هر پردازنده تعدادی زیادی عمل انجام می‌دهد که کل دستور چندین دور ساعت طول می‌کشد.

ب) تفاوت‌ها و شباهت‌ها در بخش‌های مختلف را برای ورژن ۳۲ بیتی دو معماری را به شکل زیر داریم:

۱) **رجیسترها:** معماری x86 ۳۲ بیتی (80386) تمامی رجیسترهای ۱۶ بیتی‌اش را به ۳۲ بیتی مانند RISC-V تعمیم داد که کاربرد عمومی دارند. اما این معماری تنها ۸ رجیستر عمومی دارد در حالی که تعداد رجیسترهای عمومی RISC-V ۴ برابر آن است.

۲) **تعداد عملوندها:** دستورات حسابی در x86 دارای دو عملوند هستند، یعنی یکی از آن‌ها باید هم منبع و هم مقصد باشد. در حالی که RISC-V برای هر کدام اجازه وجود رجیسترهای متفاوت برای منبع و مقصد می‌دهد و اعمال حسابی را با سه عملوند انجام می‌دهد.

۳) **مکان عملوندها:** معماری x86 و RISC-V هردو اجازه وجود عملوند ثابت و در رجیستر را می‌دهند، اما x86 اجازه وجود یک عملوند حافظه کنار یک عملوند رجیستر را می‌دهد که در RISC-V ممکن نیست.

۴) **اندازه عملوند:** برای هر نوع عملوند، اندازه‌های روبه‌رو را داریم: مقادیر ثابت در x86 طول ۸، ۱۶، یا ۳۲ بیت دارند اما در RISC-V طولشان ۱۲ یا ۲۰ بیت است؛ رجیسترها در x86 از ۱۴ رجیستر اصلی (بعضی ۳۲ بیت و بعضی ۱۶ بیت) بوده اما در RISC-V از ۳۲ رجیستر (۳۲ بیت) است؛ آدرس حافظه در x86 دارای حالت‌های متفاوت آدرس‌دهی‌ست که علاوه بر آدرس ذخیره شده در رجیستر (که خود رجیستر ۱۶ یا ۳۲ بیتی است) طول جابه‌جایی ۸ یا ۳۲ بیت می‌تواند باشد، اما در RISC-V تنها یک حالت وجود دارد که آدرس در یک رجیستر ۳۲ بیتی با یک جابه‌جایی ۱۲ بیتی است.

۵) **نوع دستورات:** معماری x86 معماری CISC بوده و دستورات آن حالت‌ها و قدرت بیشتری داشته و پیچیده‌اند، اما معماری RISC-V دستورات خلاصه‌تر با توان کمتر داشته و نوعشان ساده (RISC) است.

۶) **اندازه دستورات:** در RISC-V تمامی دستورات ۳۲ بیتی هستند اما در معماری x86 بسته به نوع دستور، طولشان از ۱ تا ۱۵ بایت (۴ تا ۱۲۰ بیت) متغیر است. این امر به علت پارامترهای پیشوندی ۱ بیتی اضافه‌تر x86 (مانند تعیین مسیر از یا به حافظه، یک بیتی یا دوکلمه‌ای بودن عمل)، ثابت‌های ۳۲ بیتی در بعضی دستورات، جابه‌جایی آدرس ۳۲ بیتی، opcode دو بیتی، و یک بایت حاصل از حالت اندیس مقیاس شده در زمان دسترسی به حافظه می‌باشد.

پ) معماری x86 (یا 80x86) این نام را گرفت زیرا در آخر نام چندین پردازنده قبل 8086 اینتل همگی 86 وجود داشت، مانند 80186، 80286 و غیره، و تنها عدد قبل از آن تغییر می‌کرد.

این معماری در اکثر کامپیوترهای میزی و لپ‌تاپ امروزه و همچنین ایستگاه کارها، سرورها، و خوشه‌های ابرکامپیومتر استفاده می‌شود، با وجود اینکه در ابتدا برای سیستم‌های تعبیه‌شده و یارانه‌های کوچک تک یا چندکاربره توسعه داده شده بود. تلاش‌های متعددی توسط شرکت‌ها از جمله خود Intel برای توقف حضور برجسته‌ی این معماری بی‌ظرافت که به طور مستقیم از ریزپردازنده‌های ۸ بیتی ابتدایی توسعه داده شده بود در بازار شده است، اما این تلاش‌ها موفق نبوده‌اند. علت این امر، عواملی مانند اصلاح ممتد ریزمعماری‌های x86، مدار و ساخت نیم‌رساناها، به علاوه‌ی توسعه و تعمیم معماری و توانایی مقیاس شدن چیپ‌های آن برای پردازنده‌های چند هسته‌ای مدرن می‌باشد؛ به زبان دیگر توسعه و تعهد جمعی صنعت به این معماری و وجود زمینه‌ای وسیع از پشتیبانی آن را از نیاز به رقابت با معماری‌های جدید تا حد زیادی محفوظ داشته است.

از شرکت‌هایی که از این معماری برای تولید پردازنده استفاده کرده و می‌کنند می‌توان خود Intel و همچنین IBM، VIA، NEC، AMD، TI، STM، Fujitsu، OKI، Siemens، Cyrix، Intersil، C&T، NexGen، UMC، و DM&P را نام برد که تنها Intel، AMD، VIA و DM&P پروانه‌ی این معماری را دارند که فقط دو شرکت اولی در حال حاضر به طور ممتد طراحی‌های مدرن تولید می‌کنند. همچنین تعدادی از پردازنده‌های معروف در طول زمان شامل 5x86 (از IBM و Cyrix)، ادامه: x86۶ کارا و x86MX۶ (از Cyrix)، k86۵ پیشرفته و سری K6 و نتایج آن Athlon و Opteron

(از AMD)، Nx586 (از NexGen)، C3 و C7 بهینه در مصرف انرژی (از VIA)، و خانواده‌ی ۷ پردازنده‌های x86 (از Zhaoxin بر مبنا و با همکاری VIA) که در انتظار انتشار است، می‌باشند.

ت) برای دستگاه‌های موبایل مانند گوشی‌های هوشمند یا تبلت‌ها معماری ARM بر x86 بسیار پیشی دارد و ساخت این نوع دستگاه‌ها با معماری x86 غیرکاربردی‌ست. زیرا مصرف انرژی این معماری بسیار بالاست و برای دستگاه‌های موبایل که مصرف انرژی بهینه‌تر و کمتر نیاز دارند کارآمد نیست. علاوه بر آن، مجوز ساخت چیپ‌های x86 در دست شرکت‌های معدودی است.

این ناکارآمدی مصرف انرژی x86 علتی‌ست که شرکت Apple در حال بهینه‌سازی ARM برای عملکرد سطح بالا بوده و از آن در لپ‌تاپ‌های خود استفاده می‌کند و زمینه‌ی جایگزینی x86 در حیطه‌ی معماری کامپیوترها را فراهم شده است. معماری RISC-V نیز که به سرعت در حال رشد است و پردازنده‌های قوی بیشتری در حال استفاده از آن هستند توانایی جایگزینی x86 را در ساخت یارانه‌ها را داراست، زیرا ساختار این معماری (RISC بودن معماری RISC-V و عواملی دیگر مانند مدل حافظه‌ی ضعیف استفاده شده در آن) پیچیدگی و قدم‌های پیاده‌سازی محاسبات موازی را کاهش داده و باعث شده این معماری پر سرعت‌تر و در مصرف انرژی بهینه‌تر باشد.

البته تا وقوع این اتفاق زمان زیادی باقی مانده است، زیرا هنوز نه تنها مجموعه دستورات RISC-V به ثباتی نرسیده که به برنامه‌نویس‌ها اطمینانی درمورد ساخت سیستم دهد، بلکه این معماری برای عملکرد سطح بالای نیاز در کامپیوترهای خانگی و لپ‌تاپ‌ها بهینه‌سازی و پیاده‌سازی نشده است.

مراجع:

کتاب منبع درس:

Computer Organization and Design (RISC-V Edition) – Patterson & Hennessy

منابع اینترنتی:

https://en.m.wikipedia.org/wiki/X86
https://www.thephonetalks.com/is-risc-v-replacing-x86-arm-high-performance
https://www.reddit.com/r/RISCV/comments/kd321u/any_reason_for_riscv_or_anything_to_replace_x86