دانشگاه صنعتی شریف
دانشکده مهندسی کامپیوتر

پروژه پایانی

گزارش کار پروژه پایانی درس معماری کامپیوتر

نگارش

سارا قضاوی، زهرا قصابی، آرش قوامی، پارسا نوروزی منش

استاد

دکتر حسین اسدی

تیر ۱۴۰۴

فهرست مطالب

۱ **پردازنده Multi-Core** ۱

۱-۱ گام اول: ایجاد ساختار جدید و خواندن همزمان دو پردازنده ۱

۱-۲ گام دوم: طراحی بافر واسط برای نوشتن در حافظه ۲

۱-۳ گام سوم: پیاده‌سازی دستور cpuid ۳

۱-۴ گام چهارم: پیاده‌سازی دستور sync ۴

۱-۵ توضیح کد تست‌بنچ ۵

۱-۶ ضرب دو ماتریس ۸ * ۸ . . . . . ۶

۲ **بخش امتیازی** ۱۱

۲-۱ مشکل دسترسی همزمان دو هسته به بخشی مشترک از حافظه ۱۱

۲-۱-۱ شرح مشکل ۱۱

۲-۱-۲ نشان دادن مشکل برای پردازنده‌ی دوهسته‌ای ۱۲

۲-۲ حل مشکل دسترسی همزمان دو هسته به بخش یکسانی از حافظه ۱۲

۲-۲-۱ شرح راه‌حل ۱۲

۲-۲-۲ حل مشکل ۱۵

فصل ۱

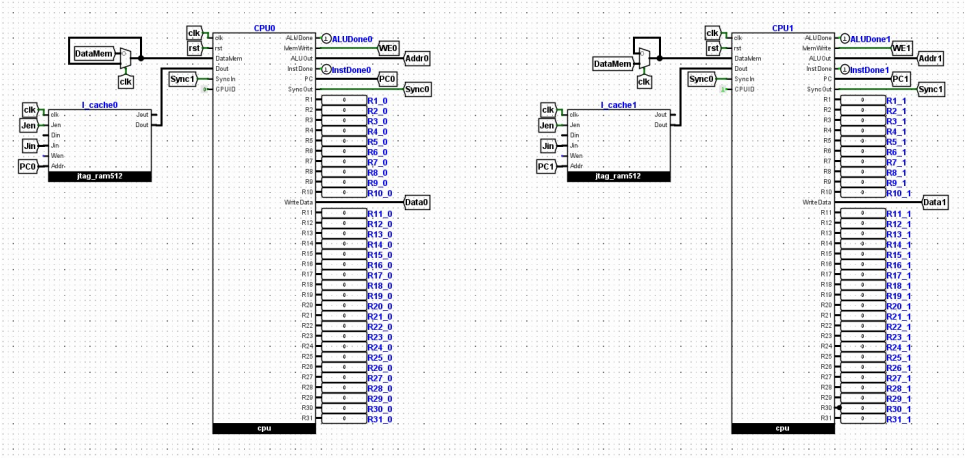
پردازنده Multi-Core

در این پروژه، با استفاده از دو پردازنده Multi-Cycle که در تمرین‌های درس طراحی شده بود، یک پردازنده Multi-Core با دو هسته و یک حافظه مشترک طراحی کردیم. به‌جز حافظه، هر هسته دارای واحد محاسباتی (ALU)، اشاره‌گر دستورها (PC)، ثبات‌ها (Register File) و واحد کنترلی مستقل است. خواندن و نوشتن در حافظه مشترک و نگهداری و همگام‌سازی داده‌ها با استفاده از دستورات معینی که به پردازنده اضافه شده است کنترل می‌شود.

۱-۱ گام اول: ایجاد ساختار جدید و خواندن همزمان دو پردازنده

در این مرحله، دو پردازنده CPU0 و CPU1 به‌صورت مستقل اما هم‌زمان طراحی شدند. هر پردازنده دارای اجزای اختصاصی شامل ALU، PC، فایل ثبات‌ها و کنترل‌گر مخصوص به خود است. این دو هسته به یک حافظه داده مشترک متصل‌اند که از طریق باس مشترک، امکان خواندن اطلاعات را به‌صورت هم‌زمان فراهم می‌کند.

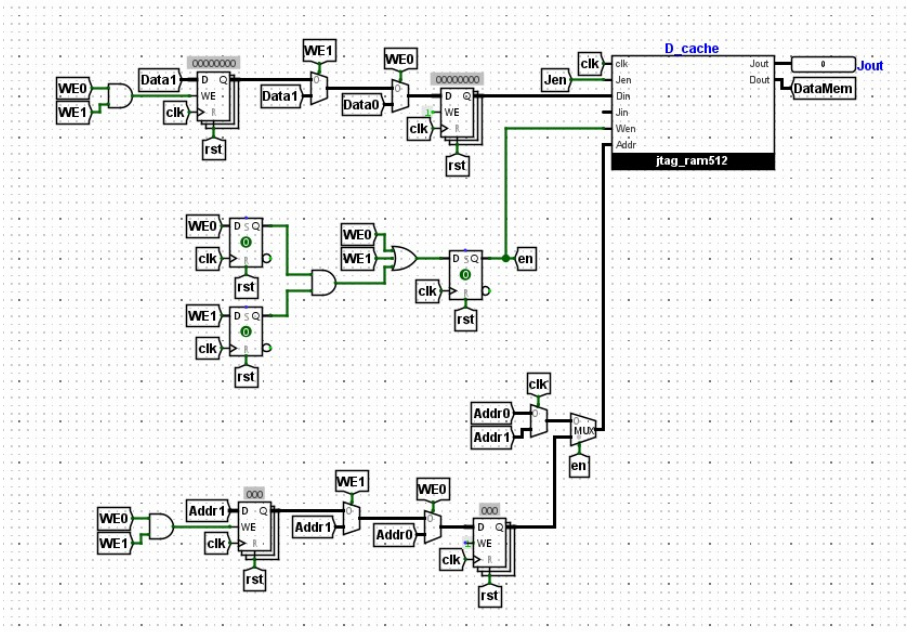
برای جلوگیری از تداخل در خواندن، خروجی‌های آدرس و داده از هر پردازنده با استفاده از مالتی‌پلکسرها ترکیب شده‌اند تا داده‌ی مربوط به هر پردازنده در زمان مناسب و بدون ایجاد تداخل به حافظه منتقل شود. در طراحی فعلی، فرض شده که عملیات خواندن حافظه از سوی هر دو پردازنده به‌طور مستقل و بدون نیاز به هماهنگ‌سازی خاص انجام می‌شود به طوری که خواندن برای پردازنده ی اول در سطح ۰ کلاک و برای پردازنده دوم در سطح ۱ کلاک انجام میشود.



شکل ۱-۱: ساختار کلی پردازنده دو هسته‌ای با حافظه مشترک

۱-۲ گام دوم: طراحی بافر واسط برای نوشتن در حافظه

برخلاف خواندن، عملیات نوشتن به حافظه نیاز به کنترل بیشتری دارد تا از تداخل بین دو پردازنده جلوگیری شود. برای حل این مشکل، یک مدار واسط برای نوشتن طراحی شد. این مدار شامل سیگنال‌های کنترل مانند Write Enable برای هر هسته، یک مالتی‌پلکسر برای انتخاب آدرس و داده‌ی مناسب، و سیگنال enable نهایی برای فعال‌سازی حافظه است. در مدار فعلی اگر در یک کلاک یک پردازنده در حال نوشتن باشد داده آن وارد بافر نوشتن شده و اگر دو پردازنده در حال نوشتن باشند داده هر دو وارد صف نوشتن شده و به صورت FIFO وارد حافظه میشوند.



شکل ۱-۲: مدار کنترل نوشتن برای دو پردازنده

۳-۱ گام سوم: پیاده‌سازی دستور cpuid

برای تشخیص هویت هر پردازنده و امکان تفکیک آن‌ها در برنامه، دستور cpuid به مجموعه دستورالعمل‌های پردازنده افزوده شد. این دستور شناسه‌ی هر پردازنده را (۰ برای CPU0 و ۱ برای CPU1) در یکی از ثبات‌های رجیستر فایل ذخیره می‌کند.

برای پیاده‌سازی این دستور، یک بیت به نام cpu_id به‌صورت ورودی به هر پردازنده داده شده و در هنگام اجرای دستور، مقدار این بیت به‌عنوان خروجی cpuid در رجیستر مقصد ذخیره می‌شود. این امکان را به برنامه‌نویس می‌دهد تا هر بخش از برنامه را فقط توسط یک پردازنده خاص اجرا کند یا داده‌ای خاص به هر پردازنده اختصاص دهد.

برای بررسی صحت عملکرد مدار، با استفاده از تست بنچ زیر آن را آزمایش کردیم:

```
instructions[0] = 32'b100000_00100_00000000000000000000;  // cpuid $4
instructions[1] = 32'b000101_00100_00000000000000001001;  // bne $4 , $0, 9

// core 0
instructions[2] = 32'b001000_00000_01000_0000000000000101;  // addi $8, $0, 5     -> $8 = 5
instructions[3] = 32'b001000_00000_01001_0000000000001010;  // addi $9, $0, 10    -> $9 = 10
instructions[4] = 32'b001000_11101_11101_1111111111111000;  // -> $29 = 1111111111111000
instructions[5] = 32'b001000_11110_11110_1111111111111100;  // -> $30 = 1111111111111100
instructions[6] = 32'b101011_11101_01000_0000000000000000;  // sw $8, 0($29)
instructions[7] = 32'b101011_11110_01001_0000000000000000;  // sw $9, 0($30)
instructions[8] = 32'b100011_11101_01010_0000000000000000;  // lw $10, 0($29)
instructions[9] = 32'b100011_11110_01011_0000000000000000;  // lw $11, 0($30)
instructions[10] = 32'b000010_00000000000000000000_010011;  // j end

//core 1
instructions[11] = 32'b001000_00000_01010_0000000000000111;  // addi $10, $0, 7  -> $10 = 7
instructions[12] = 32'b001000_00000_01011_0000000000000100;  // addi $11, $0, 4  -> $11 = 4
instructions[13] = 32'b001000_10000_10000_1111111111110000;  // -> $16 = 1111111111110000
instructions[14] = 32'b001000_10001_10001_1111111111110100;  // -> $17 = 1111111111110100
instructions[15] = 32'b101011_10000_01010_0000000000000000;  // sw $10, 0($16)
instructions[16] = 32'b101011_10001_01011_0000000000000000;  // sw $11, 0($17)
instructions[17] = 32'b100011_10000_01000_0000000000000000;  // lw $8, 0($16)
instructions[18] = 32'b100011_10001_01001_0000000000000000;  // lw $9, 0($17)

// end :
last_instr = 19;
```

شکل ۱-۳: تست بنچ بررسی صحت عملیات نوشتن و خواندن

همانطور که مشاهده می شود، هر پردازنده آیدی مربوط به خود را در ثبات چهارم ذخیره می کند و با توجه به مقدار این ثبات دستورات مخصوص خود را انجام می دهد. هر دو پردازنده به صورت موازی مقادیر مشخصی را در حافظه نوشته و سپس آن را می خوانند. همچنین برای بررسی دقیق تر، دو دستور sw و دو دستور lw پشت سر هم قرار داده شده تا پردازنده به درستی آزمایش شود. نتیجه این تست در ترمینال به صورت زیر است:

```
ipc0 :   0
ipc1 :   0
ipc0 :   1
wat          0 !=          0          9
ipc1 :   1
wat          1 !=          0          9
ipc0 :   2
ipc1 :  11
ipc0 :   3
ipc1 :  12
ipc0 :   4
ipc1 :  13
ipc0 :   5
ipc1 :  14
ipc0 :   6
stor 000001fe
ipc1 :  15
stor 000001fc
ipc0 :   7
stor 000001ff
ipc1 :  16
stor 000001fd
ipc0 :   8
load 000001fe
ipc1 :  17
load 000001fc
ipc0 :   9
load 000001ff
ipc1 :  18
load 000001fd
ipc0 :  10
ACCEPTED CORE 1, steps :          10
```

شکل ۱-۴: نتیجه تست سه گام اول

۱-۴ گام چهارم: پیاده‌سازی دستور sync

برای پیاده‌سازی دستور sync، یک وضعیت (state) جدید به ماشین حالت پردازنده اضافه کردیم. در صورت شناسایی کد عملیات این دستور، سیگنال syncOut پردازنده فعال می‌شود (مقدار یک می‌گیرد) و پردازنده تا زمانی که سیگنال ورودی syncIn مقدار صفر داشته باشد، در این وضعیت باقی می‌ماند و از واکشی دستور بعدی خودداری می‌کند.

برای همگام‌سازی میان دو هسته، سیگنال syncOut هر پردازنده به عنوان سیگنال syncIn

پردازنده‌ی مقابل متصل شده است. به این ترتیب، زمانی که هر دو پردازنده به دستور sync برسند، هر دو سیگنال syncIn مقدار یک خواهند گرفت و پردازنده‌ها می‌توانند از وضعیت انتظار خارج شده و اجرای دستورات بعدی را ادامه دهند.

این مکانیزم باعث همگام‌سازی دقیق بین دو پردازنده می‌شود و تضمین می‌کند که هیچ‌کدام پیش از رسیدن دیگری به نقطه‌ی همگام‌سازی، اجرای دستورات خود را ادامه ندهد.

برای تست این دستور مجموعه دستورات زیر را ران میکنیم :

```
instructions[0] = 32'b100000_00100_00000000000000000000;  // cpuid $4
instructions[1] = 32'b000101_00100_00000000000000000100;  // bne $4 , $0, 4
// core 0
instructions[2] = 32'b001000_00000_01000_00000000000000101;  // addi $8, $0, 5     -> $8 = 5
instructions[3] = 32'b101010_00000000000000000000000000;    // sync
instructions[4] = 32'b001000_00000_01001_00000000000001010;  // addi $9, $0, 10    -> $9 = 10
instructions[5] = 32'b000010_00000000000000000000_001011;  // j end
// core 1
instructions[6] = 32'b001000_00000_01010_00000000000000111;  // addi $10, $0, 7  -> $10 = 7
instructions[7] = 32'b001000_00000_01001_00000000000000100;  // addi $9, $0, 4  -> $9 = 4
instructions[8] = 32'b001000_00000_01100_00000000000001101;  // addi $12, $0, 13  -> $12 = 13
instructions[9] = 32'b101010_00000000000000000000000000;    // sync
instructions[10] = 32'b000000_01010_01001_01010_00000_100010;  // sub $10, $9, $12  -> $10 = 3
// end :
```

شکل ۱-۵: تست گام چهارم

همانطور که در نتیجه تست مشخص است مقادیر ثبات ها درست است و core 0 تا زمانی که core 1 هم به دستور sync برسد اجرای دستوراتش متوقف شده.

```
ipc0 :   0
ipc1 :   0
ipc0 :   1
wat         0 !=          0           4
ipc1 :   1
wat         1 !=          0           4
ipc0 :   2
ipc1 :   6
ipc0 :   3
ipc1 :   7
ipc1 :   8
ipc1 :   9
ipc0 :   4
ipc1 :  10
ipc0 :   5
[8] =          5, [9] =         10, [10] =          3, [9] =          4, [12] =         13
```

شکل ۱-۶: نتیجه تست گام ۴

۱-۵ توضیح کد تست‌بنچ

در این قسمت، کد تست‌بنچ برای شبیه‌سازی عملکرد دو هسته‌ی موازی طراحی شده است. این تست‌بنچ اجرای همزمان دو پردازنده را بررسی می‌کند و نتیجه‌ی عملکرد آن‌ها را با مقدار مورد انتظار مقایسه می‌نماید.

در ابتدا حافظه‌ی دستورالعمل و حافظه‌ی داده (هر کدام با ۵۱۲ خانه‌ی ۳۲-بیتی) مقداردهی اولیه می‌شوند. سپس، برای هر هسته، مجموعه‌ای از ثبات‌ها تعریف و مقداردهی اولیه می‌شود. همچنین دو شمارنده‌ی دستورالعمل (IPC) برای پیگیری مکان فعلی اجرای دستور هر پردازنده استفاده شده‌اند.

کد شامل دو تسک اصلی به نام‌های exec internal0 و exec internal1 است که به ترتیب وظیفه‌ی اجرای دستورالعمل‌های هسته‌ی اول و دوم را بر عهده دارند. این تسک‌ها با توجه به نوع دستورالعمل (از جمله R type، I type و J type) عملیات لازم را انجام داده و مقادیر ثبات‌ها یا حافظه را به‌روزرسانی می‌کنند. از جمله دستورالعمل‌های پشتیبانی‌شده می‌توان به add، sub، lw، sw، beq، bne، jal، cpuid و sync اشاره کرد.

برای همگام‌سازی اجرای دستورالعمل‌ها در دو هسته، از دستور sync استفاده شده که با اضافه کردن وضعیت جدیدی به ماشین حالت پردازنده پیاده‌سازی شده است. زمانی که هر دو هسته به دستور sync برسند، تا رسیدن دیگری در همان حالت باقی می‌مانند.

در انتهای تست‌بنچ، اجرای هر دو هسته به صورت موازی با استفاده از دو بلاک initial کنترل می‌شود. پس از اجرای هر دستور، وضعیت ثبات‌های داخلی با خروجی پردازنده واقعی مقایسه شده و در صورت عدم تطابق پیام خطا چاپ می‌شود. در صورت موفقیت کامل، پیام ACCEPTED برای هر هسته چاپ خواهد شد.

۶-۱ ضرب دو ماتریس ۸ * ۸

در اینجا ابتدا به کمک یک کد java ضرب دو ماتریس 8*8 را پیاده سازی کردیم. سپس می خواهیم یک testbench طراحی کنیم که از درستی این مدارات اطمینان پیدا کنیم. پس عملا کد زده شده به زبان java را به صورت اسمبلی mips تبدیل کرده و در نهایت به صورت باینری تبدیل کردیم.

عملا کد testbench به صورتی طراحی شده است که یک task داریم که عملا یک دستور را اجرا می کند. در واقع دستوری که pc روی آن قرار دارد را بارکشی کرده و اجرا می کند. در یک بخش initial یک حلقه داریم که تا زمانی که به آخرین دستور نرسیده ایم و یا به ایرادی برنخورده ایم ادامه پیدا می کند. ایراد در کد را به کمک یک متغیر به نام donefail$_{flag}$.fail$_{flag}$. به درستی در مدار تغییر نمی کرد و در دستوراتی ممکن بود دستور هنوز به پایان نرسیده باشد ولی این سیگنال یک شود. با ایجاد تغیرات اندکی در مدار این مشکل نیز حل شد. در نهایت یک آرایه از دستورات به نام instructions داریم که باینری به دست آمده از دستورات را در آن قرار دادیم که مرحله به مرحله اجرا شود.

دقت کنید که نکته این موضوع که مدار multi-core بوده است این است که چهار سطر ابتدایی ماتریس توسط یکی از هسته ها و چهار سطر دیگر توسط هسته دیگر بررسی می شود. عملا ضرب هر چهار سطر در کل ستون ها توسط یک هسته انجام شده است. در نهایت لازم به ذکر است که مقادیری که ضرب شده اعدادی بوده اند که توسط یک حلقه با یک الگوی خاص تولید شده اند و در حافظه قرار گرفته

اند. می توان به جای این مقدار ها هر مقدار دیگری را نیز وارد حافظه کرد و ضرب ماتریس های دیگری را هم داشت! بخشی از کد های اسمبلی را که در آرایه instructions قرار دارد را می توانید در زیر مشاهده کنید:

```
instructions[0] = 32'b00100000000000110000000000000000; // addi $3, $0, 0
instructions[1] = 32'b00100000000001000000000100000000; // addi $4, $0, 256
instructions[2] = 32'b10101100011000110000000000000000; // sw $3, 0($3)
instructions[3] = 32'b00100000110001100000000000000100; // addi $3, $3, 4
instructions[4] = 32'b00010100011001001111111111111101; // bne $3, $4, -3
instructions[5] = 32'b00100000000000110000000000000000; // addi $3, $0, 0
instructions[6] = 32'b10101100011000110000000100000000; // sw $3, 256($3)
instructions[7] = 32'b00100000110001100000000000000100;// addi $3, $3, 4
instructions[8] = 32'b00010100011001001111111111111101; // bne $3, $4, -3
instructions[9] = 32'b10000000100000000000000000000000;  // cpuid $4
instructions[10] = 32'b00010100100000000000000000011110;  // bne $4 , $0, 30
// core 0
instructions[11] = 32'b00100000000000110000000000000000;// addi $3, $0, 0
instructions[12] = 32'b00100000000001000000000000000100;// addi $4, $0, 4
instructions[13] = 32'b00100000000001010000000000000000;// addi $5, $0, 0
instructions[14] = 32'b00100000000001100000000000001000;// addi $6, $0, 8
instructions[15] = 32'b00100000000001110000000000000000;// addi $7, $0, 0
instructions[16] = 32'b00100000000001000000000000001000;// addi $8, $0, 8
instructions[17] = 32'b00100000000001001000000000000000;// addi $9, $0, 0
instructions[18] = 32'b00100000000001010000000000001000;// addi $10, $0, 8
instructions[19] = 32'b00000000000000111010000011000000; // sll $10, $3, 3
instructions[20] = 32'b00000000101000111010000000100000; // add $10, $10, $7
instructions[21] = 32'b00000000000001010010100001000000;// sll $10, $10, 2
instructions[22] = 32'b10001101010010100000000000000000; // lw $10, 0($10)
instructions[23] = 32'b00100000000001011000000000001000;// addi $11, $0, 8
instructions[24] = 32'b00000000000000111010110001100000;// sll $11, $7, 3
instructions[25] = 32'b00000000101100101010110000100000;// add $11, $11, $5
instructions[26] = 32'b00000000000001011010110001000000;// sll $11, $11, 2
instructions[27] = 32'b10001101011010110000000000000000;// lw $11, 0($11)
instructions[28] = 32'b01110001010010110101000000000000;// mul $10, $10, $11
instructions[29] = 32'b00000000100101010010010000100000;// add $9, $9, $10
instructions[30] = 32'b00100000111001110000000000000001;// addi $7, $7,1
instructions[31] = 32'b00010101000001111111111111110010;// bne $8, $7, -14
instructions[32] = 32'b00100000000001010000000000001000;// addi $10, $0, 8
instructions[33] = 32'b00000000000000111010000011000000;// sll $10, $3, 3
instructions[34] = 32'b00000000101000101010100000100000;// add $10, $10, $5
instructions[35] = 32'b00000000000001010010100001000000;// sll $10, $10, 2
instructions[36] = 32'b10101101010010010000001000000000;// sw $9, 512($10)
instructions[37] = 32'b00100000101001010000000000000001;// addi $5, $5, 1
instructions[38] = 32'b00010100110001011111111111101000;// bne $5, $6, -24
instructions[39] = 32'b00100000110001100000000000000001;// addi $3, $3, 1
instructions[40] = 32'b00010100100000111111111111100100;// bne $4, $3, -28
instructions[41] = 32'b00000100000000000000000000101010; // j end ..............
//core 1
instructions[41] = 32'b00100000000000110000000000000100;// addi $3, $0, 4
instructions[42] = 32'b00100000000001000000000000001000;// addi $4, $0, 8
instructions[43] = 32'b00100000000001010000000000000000;// addi $5, $0, 0
instructions[44] = 32'b00100000000001100000000000001000;// addi $6, $0, 8
instructions[45] = 32'b00100000000001110000000000000000;// addi $7, $0, 0
instructions[46] = 32'b00100000000001000000000000001000;// addi $8, $0, 8
instructions[47] = 32'b00100000000001001000000000000000;// addi $9, $0, 0
instructions[48] = 32'b00100000000001010000000000001000;// addi $10, $0, 8
instructions[49] = 32'b01110001010000110101000000000000;// mul $10, $10, $3
instructions[50] = 32'b00000000101000111010100000100000;// add $10, $10, $7
instructions[51] = 32'b10001101010010100000000000000000;// lw $10, 0($10)
instructions[52] = 32'b01110001010000100101000000000000;// mul $10, $10, $4
```

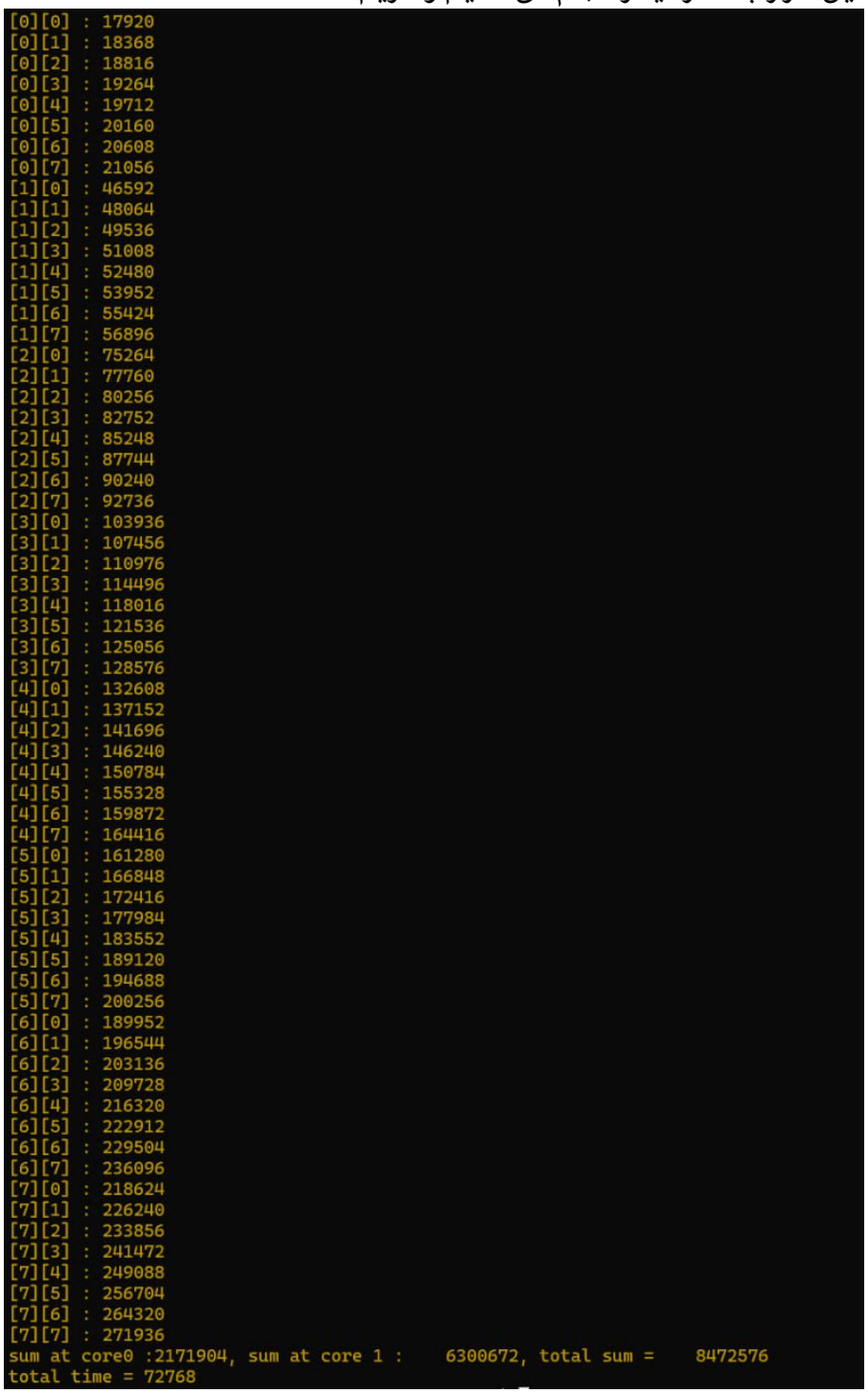
شکل ۱-۷: بخشی از کد های اسمبلی برای انجام ضرب ماتریس

در ادامه قصد داریم که این معماری چند هسته که هر هسته آن به صورت multi cycle پیاده سازی شده است را با معماری پردازنده تک هسته multi cycle مقایسه کنیم. برای این کار ابتدا بخش های instructions را در تست بنچ این مدار تک هسته با این دستورات در مدار چند هسته جایگزین می کنیم که عملیات یکسانی انجام شود. سپس زمان ابتدایی و انتهایی را نگه می داریم تا اختلاف را پیدا کنیم. سپس همین کار را برای مدار چند هسته انجام میدهیم. برای مدار تک هسته داریم:

```
ipc :  45
ipc :  46
ipc :  47
ipc :  48
load 0000009c
ipc :  49
ipc :  50
ipc :  51
wat         8 !=          5          -8
ipc :  44
ipc :  45
ipc :  46
ipc :  47
ipc :  48
load 0000009d
ipc :  49
ipc :  50
ipc :  51
wat         8 !=          6          -8
ipc :  44
ipc :  45
ipc :  46
ipc :  47
ipc :  48
load 0000009e
ipc :  49
ipc :  50
ipc :  51
wat         8 !=          7          -8
ipc :  44
ipc :  45
ipc :  46
ipc :  47
ipc :  48
load 0000009f
ipc :  49
ipc :  50
ipc :  51
wat         8 !=          8          -8
ipc :  52
ipc :  53
wat         4 !=          4         -12
ipc :  54
ipc :  55
stor 00000000
ACCEPTED
       4586 /       4586
total time:74640
arash@server:~/CA/SUT_CA_4032_ProfAsadi_Judgement_System$
```

شکل ۱-۸: مدت زمان مورد نیاز برای یک پردازنده تک هسته

و حالا همین کار را با مدار دیگر انجام می دهیم و داریم:



شکل ۱-۹: مدت زمان مورد نیاز برای یک پردازنده با دو هسته

همانظور که مشاهده می کنید زمان مورد نظر برای انجام این محاسبه در حالتی که از دو تا هسته استفاده کردیم کاهش یافت.

چالش اصلی و به شدت جدی که در این بخش داشتیم اول تبدیل کد اسمبلی به باینری بود که دارای

خطای زیادی بود. به این دلیل که این پردازنده را خود ما طراحی کرده بودیم و نمی توانسیتم کد ها را به مفسر هایی مانند mars بدهیم تا برایمان کد باینری ایجاد کند.

از طرف دیگر در testbench که نوشته بودیم طول هر دستور را برای beq و یا bne به صورت یک کلمه یک بایت در نظر گرفته بودیم. ولی در دستورات lw, sw طول هر کلمه چهار بایت بود. بنابرین با مشکلی رو به رو شدیم که به کمک یک shifter آن را به صورت سخت افزاری حل کردیم.

فصل ۲

بخش امتیازی

۲-۱ مشکل دسترسی همزمان دو هسته به بخشی مشترک از حافظه

۲-۱-۱ شرح مشکل

برای شرح این مشکل، یک مثال می‌زنیم که در درس برنامه‌نویسی پیشرفته در مبحث threadها نیز با آن آشنا شدیم.فرض کنید دوthread داریم که هر کدام صرفا یک حلقه ۱۰۰۰ تایی دارند که در هر بار آن یک متغیر یکسان را به اضافه یک می‌کنند.حال دو thread شروع به کار می‌کنند و منتظر می‌شویم که کار هر دو تمام شود اما با تعجب می‌بینیم که پس از اتمام کار هر دو مقدار آن متغیر که ابتدا صفر بود با کمال تعجب کمتر از ۲۰۰۰ است. دلیل این مورد این است که بخش critical section کد که همان به علاوه ۱ کردن آن متغیر است که خود شامل ۳ مرحله‌ی خواندن مقدار متغیر از حافظه، به اضافه ۱ کردن آن و استور کردن حاصل در حافظه است، اگر به شکل atomic پیاده‌سازی نشود و هر دو thread بتوانند همزمان از حافظه بخوانند و در آن بنویسند مشکل به وجود می‌آید. برای مثال فرض کنید ابتدا هر دو thread مقدار ۰ را به عنوان مقدار اولیه متغیر از حافظه می‌خوانند. سپس هر دو این مقدار را به اضافه ۱ کرده و هر دو مقدار ۱ را در حافظه می‌نویسند. با این که در مجموع دو حلقه از حلقه‌های دو thread مصرف شده است و انتظار داشتیم مقدار متغیر ۲ واحد زیاد شود اما به دلیل دسترسی همزمان دو thread به حافظه مقدار متغیر تنها ۱ واحد زیاد شد.

۲-۱-۲ نشان دادن مشکل برای پردازنده‌ی دوهسته‌ای

حال همان مثالی که در زیرقسمت قبل زدیم را با اسمبلی پیاده‌سازی می‌کنیم. یعنی صرفا یک حلقه که ۱۰۰۰ بار اجرا می‌شود و در هر بار اجرای حلقه مقدار یک خانه ثابت از حافظه خوانده شده، به اضافه ۱ شده و سپس دوباره در حافظه نوشته می‌شود. سپس کد را به حالت باینری تبدیل کردیم و در تست‌بنچ قرار دادیم:

```
instructions[0] = 32'b001000_10000_10000_0000000000010100; // addi $s0, $zero, 20
instructions[1] = 32'b001000_00000_10001_0000001111101000; // addi $s1, $zero, 1000
instructions[2] = 32'b000100_00000_10001_0000000000000101; // beq $s1, $zero, exit
instructions[3] = 32'b100011100000100000000000000000000; // lw $t0, 0($s0)
instructions[4] = 32'b001000010000100000000000000000001; // addi $t0, $t0, 1
instructions[5] = 32'b101011100000100000000000000000000; // sw $t0, 0($s0)
instructions[6] = 32'b001000100011000111111111111111111; // addi $s1, $s1, -1
instructions[7] = 32'b000010000000000000000000000000010; // j loop

last_instr = 8;
```

شکل ۲-۱: کد ایجادکننده‌ی مشکل

حال تست‌بنچ را با پردازنده‌ی دوهسته‌ای خود اجرا می‌کنیم و می‌بینیم که نتیجه‌ی نهایی کمتر از ۲۰۰۰ است:

```
stor 00000005
ipc1 :   5
stor 00000005
ipc0 :   6
ipc1 :   6
ipc0 :   7
ipc1 :   7
ipc0 :   2
wat         0 ==          0          5
ipc1 :   2
wat         0 ==          0          5
value of memory cell:       1000
./HW6/bad_code_tb.v:382: $finish called at 80000 (1ps)
```

شکل ۲-۲: نتیجه اشتباه برنامه

۲-۲ حل مشکل دسترسی همزمان دو هسته به بخش یکسانی از حافظه

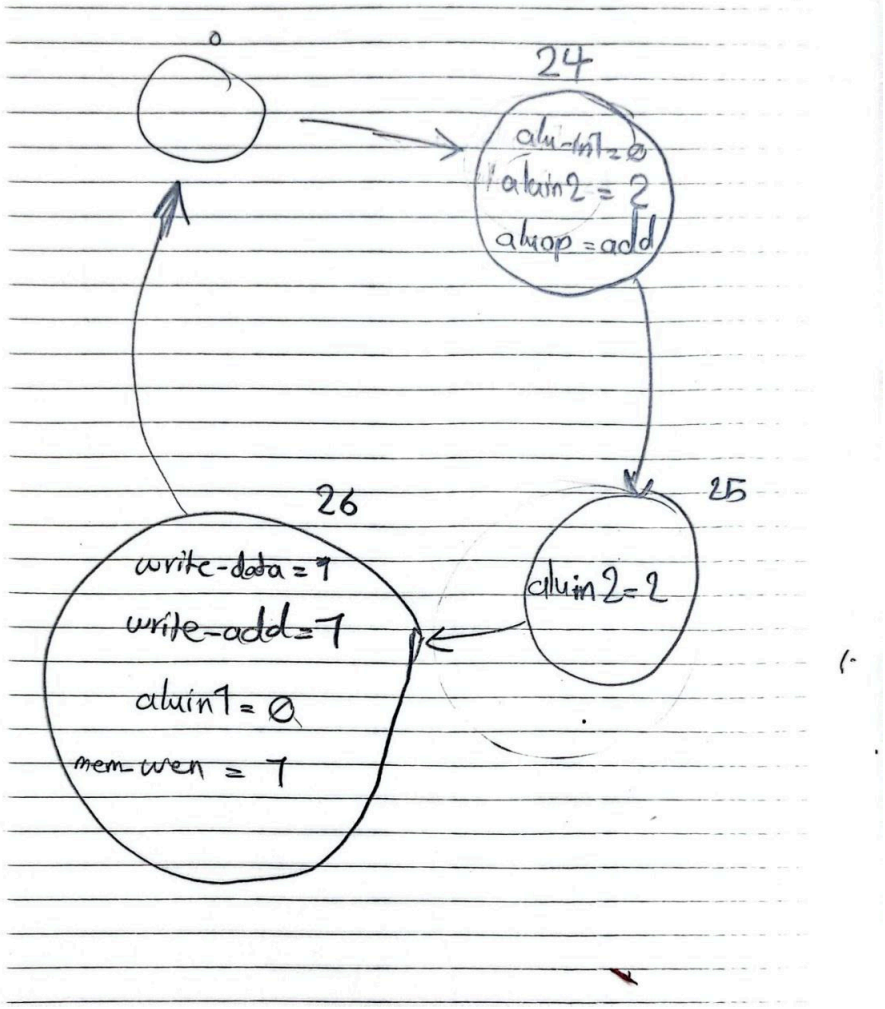
۲-۲-۱ شرح راه‌حل

مشکل از آن‌جایی ایجاد شد که دو هسته ممکن بود به طور همزمان به بخشی مشترکی از حافظه دسترسی پیدا کنند. حال برای حل این مشکل یک دستور atomic به صورت exchng rt, imm(rs) پیاده‌سازی می‌کنیم که اولا شرح کارش به این صورت است که مقدار موجود در ثبات rt و خانه‌ی حافظه با آدرس rs + imm را به یک‌باره با هم جابه‌جا می‌کند و ثانیا atomic است یعنی در حین اجرای این دستور

توسط یکی از هسته‌ها هسته‌ی دیگر نمی‌تواند از حاظه بخواند یا بنویسد. پس از پیاده‌سازی این دستور باید با استفاده از آن یک spinlock طراحی کنیم تا مشکل موجود را با آن برطرف کنیم.

پیاده‌سازی دستور exchng

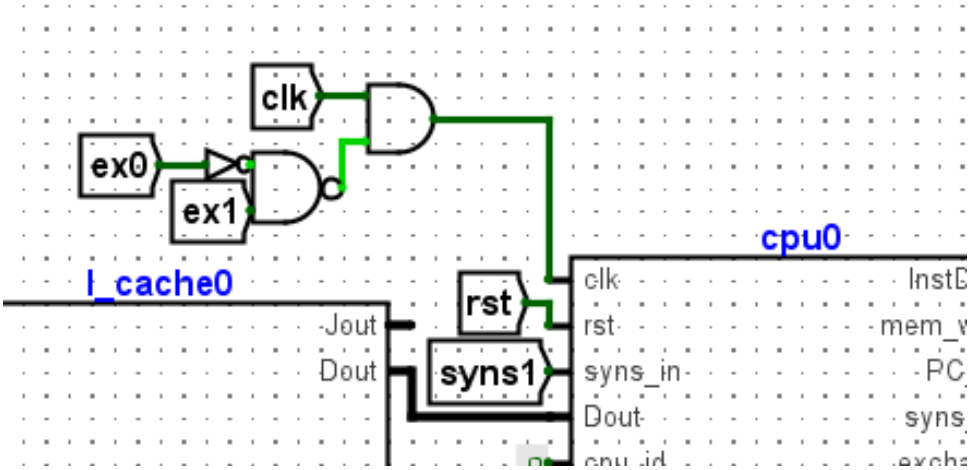
چون معماری پردازنده multi-cycle است، برای اضافه کردن دستور جدید باید stateهای موردنیاز آن را به FSM خود اضافه کنیم. حالت اولیه مانند همه‌ی دستورات حالت صفر است. با توجه به این که تا حالت ۲۳ برای دستورات قبلی استفاده شده از حالت ۲۴ شروع می‌کنیم. در حالت ۲۴ (چرخه‌ی دوم اجرای دستور exchng)، آدرس دسترسی به خانه‌ی حافظه که برابر $rs + imm$ است محاسبه می‌شود. در چرخه‌ی سوم و حالت ۲۵، مقدار موجود در آن خانه از حافظه خوانده شده و همچنین دوباره آدرس آن خانه از حافظه را حساب می‌کنیم چون در چرخه‌ی بعد هم نیاز داریم. در نهایت هم در چرخه‌ی چهارم و حالت ۲۶، همزمان مقدار موجود در ثبات rt در آن خانه از حافظه و مقدار خوانده شده در چرخه قبل از آن خانه‌ی حافظه در ثبات rt نوشته می‌شود. به طور خلاصه این بخش به FSM اضافه می‌شود:



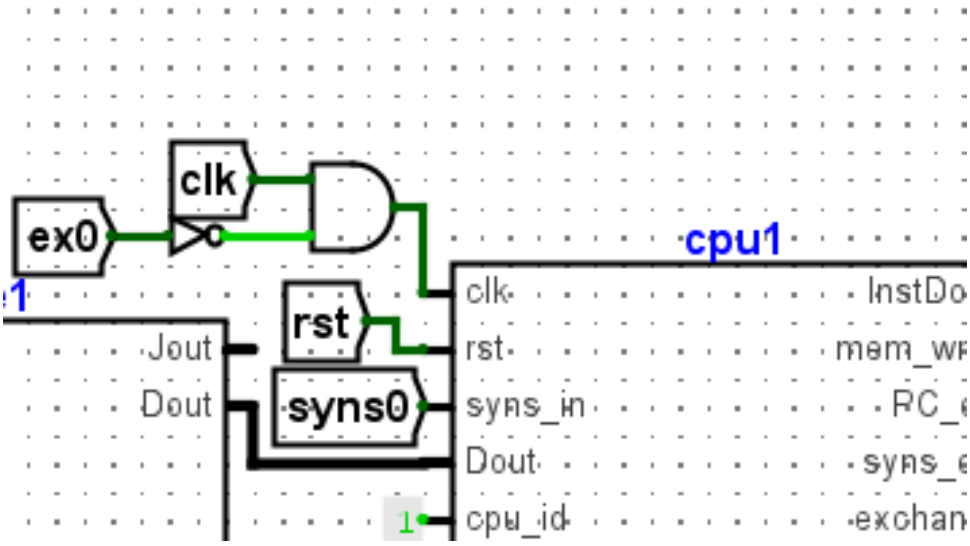
شکل ۲-۳: حالت‌های اضافه شده به اف‌اس‌ام

و با اضافه کردن این حالات به control unit این دستور پیاده‌سازی می‌شود.

حال برای atomic بودن این دستور تنها کافی است هر وقتی یک هسته به این دستور می‌رسید یک بیت به نام همین دستور از خروجی آن هسته برابر ۱ شده و این باعث می‌شود که عدد صفر با ورودی clock هسته دیگر and گرفته شود و انگار هسته دیگر اصلا کلاک نمی‌خورد و عملا stall می‌شود. همچنین اگر هر دو همزمان به این دستور رسیدند اولویت ادامه دادن با هسته شماره صفر است و در این شرایط هسته ۱ stall می‌خورد. در این عکس‌ها هم نحوه این پیاده‌سازی آمده است



شکل ۲-۴: ورودی کلاک تغییریافته‌ی هسته‌ی صفر



شکل ۲-۵: ورودی کلاک تغییریافته‌ی هسته‌ی یک

طراحی spinlock

طراحی spinlock یک روش برای حل مشکل دسترسی همزمان هسته‌ها به بخش یکسانی از حافظه است. این نوع قفل به این صورت عمل می‌کند که یک هسته در صورتی که بخواهد به یک خانه از حافظه

دسترسی پیدا کند، تا زمانی که قفل آن باز نشده و این هسته نتواند قفل را به دست بگیرد در یک حلقه وایل گیر می‌کند تا قفل که توسط یک هسته دیگر گذاشته شده آزاد شود. سپس این هسته خودش قفل می‌گذارد و زمانی که دسترسی‌اش به پایان برسد قفل را آزاد می‌کند. کد هر بخش lock و release برای طراحی یک spinlock را به این صورت پیاده‌سازی می‌کنیم(همانطور که گفته شد تا زمانی که خانه‌ی نگهدارنده قفل برابر ۱ باشد یعنی قفل باشد در حلقه گیر می‌کند و پس از آزاد شدن قفل، خودش قفل را ۱ می‌کند. برای آزاد کردن هم آن خانه را صفر می‌کند):

```
// lock
instructions[3] = 32'b001000_00000_01001_0000000000000001; // addi $t1, $zero, 1
instructions[4] = 32'b100010_00000_01001_0000000000101000; // exchng $t1, 40($zero)
instructions[5] = 32'b00010100000010011111111111111110; // bne $t1, $zero, lock
```

شکل ۲-۶: کد مربوط به lock

```
// release
instructions[9] = 32'b101011_00000_00000_0000000000101000;  // sw $0, 40($0)
```

شکل ۲-۷: کد مربوط به release

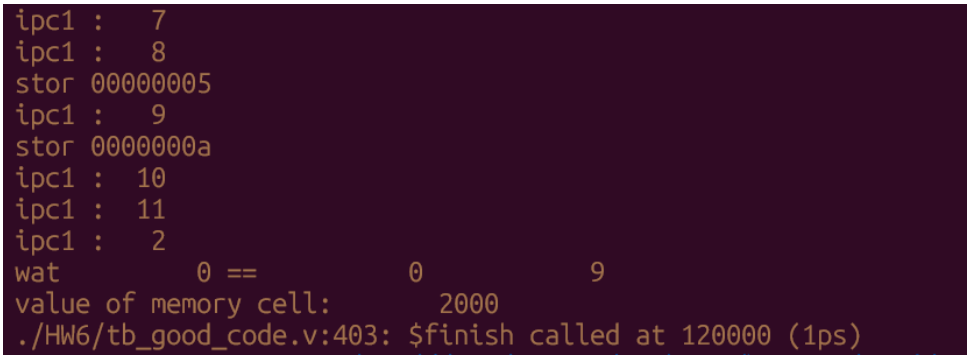
۲-۲-۲ حل مشکل

حال با spinlock طراحی شده صرفا در کد قبلی قبل از critical section که همان بخش خواندن یک خانه از حافظه، به اضافه ۱ کردن آن و نوشتن حاصل در حافظه است، بخش مربوط به lock را نوشته تا تنها یک هسته بتواند وارد critical section شود و بعد از آن هم بخش مربوط به release را می‌نویسیم تا پس از بیرون آمدن از critical section قفل را آزاد کند. کد نهایی به این شکل است:

```
instructions[0] = 32'b00100000000100000000000000010100; // addi $s0, $zero, 20
instructions[1] = 32'b00100000000100010000001111101000; // addi $s1, $zero, 1000
instructions[2] = 32'b000100_00000_10001_0000000000001001; // beq $s1, $zero, exit

// lock
instructions[3] = 32'b001000_00000_01001_0000000000000001; // addi $t1, $zero, 1
instructions[4] = 32'b100010_00000_01001_0000000000101000; // exchng $t1, 40($zero)
instructions[5] = 32'b00010100000010011111111111111110; // bne $t1, $zero, lock

instructions[6] = 32'b10001110000010000000000000000000; // lw $t0, 0($s0)
instructions[7] = 32'b00100001000010000000000000000001; // addi $t0, $t0, 1
instructions[8] = 32'b10101110000010000000000000000000; // sw $t0, 0($s0)

// release
instructions[9] = 32'b10101100000000000000000000101000; // sw $zero, 40($zero)
instructions[10] = 32'b00100010001100011111111111111111; // addi $s1, $s1, -1
instructions[11] = 32'b00000100000000000000000000000010; // j loop

last_instr = 12; // change if "sync" is added
```

شکل ۲-۸: کد نهایی

و با اجرای آن با پردازنده‌ی دوهسته‌ای جدید به نتیجه درست ۲۰۰۰ می‌رسیم:



شکل ۲-۹: نتیجه درست