به نام خدا

|  |  |  |  |
| --- | --- | --- | --- |
| مدرس: مظفر بگ محمدي | دانشگاه ايلام | ترم دوم سال تحصيلي 96-95 | پروژه ی معماری کامپیوتر |

**توضیحات کلی**

در این پروژه یک پردازنده ی 16 بیتی را طراحی می کنید و آنرا با استفاده از [Logisim](http://ozark.hendrix.edu/~burch/logisim/)  شبیه سازی می نمایید. این پردازنده شبیه MIPS است با این تفاوت که هم دستورالعملها و هم مسیر داده 16 بیتی هستند. این پردازنده 4 رجیستر دارد و آدرسهای حافظه نیز 16 بیتی هستند.

این پردازنده دارای دو سیکل است. در سیکل اول دستورالعمل از حافظه واکشی می شود و در رجیستر IR قرار می­گیرد. در سیکل دوم، دستورالعمل دیکد و اجرا می گردد. به علاوه، شما باید نسخه ی خط لوله ی این پردازنده را نیز طراحی کنید. دقت کنید که این پردازنده هازارد داده ای ندارد. اما هازارد کنترلی وجود دارد و دستورات انشعاب و پرش دارای یک سیکل تاخیر هستند. حل آنها به ترتیب زیر است. البته دقت کنید که شما می­توانید در کامپایلر بعد از هر انشعاب یا پرش یک nop قرار دهید تا این دستورات برای شما مشکلی ایجاد نکنند. اما در صورتی که، پاراگراف بعدی را بتوانید پیاده کنید نمره ی اضافی دارد.

هنگامی که ما متوجه می شویم با دستور انشعاب یا پرش سروکار داریم در کلاک دوم هستیم، ما دستور العمل بعدی را واکشی کرده­ایم و ممکن است دستور را از جای غلط برداشته باشیم. در اینجا تنها کاری که از دستمان بر می­آید این است که نگذاریم دستور وارد مرحله­ی اجرا شود و به جای آن nop وارد مرحله ی دوم گردد. دقت کنید که در مورد دستورات انشعاب، اگر انشعاب NT باشد (یعنی انجام نشود) نیازی به ورود nop نیست و می توانیم دستورالعمل بعدی را با خیال راحت اجرا کنیم. در واقع ما همیشه فرض می­کنیم که انشعاب انجام نمی­شود. به علاوه در شروع راه­اندازی پردازنده، مقدار PC باید صفر باشد. از آنجا که خود شبیه­ساز هنگام ریست شدن اینکار را انجام می­دهد، نیازی نیست که شما کار خاصی انجام دهید.

*بانک رجیسترها*

این بانک دارای 4 رجیستر 16 بیتی است. ورودیهای این ماژول به صورت زیر هستند:

|  |  |
| --- | --- |
| Read Register 1 (2 bits) | شماره ی رجیستر خواندنی اول |
| Read Register 2 (2 bits) | شماره ی رجیستر خواندنی دوم |
| Write Register (2 bits) | شماره ی رجیستر نوشتن |
| write data (16 bits) | داده ای که باید در رجیستر نوشتن نوشته شود. |
| RegWrite (1 bit) | اگر این سیگنال یک باشد، write data در رجیستر نوشتن نوشته می­شود. |
| clk (1 bit) | کلاک |

این بانک دارای خروجیهای زیر است.

|  |  |
| --- | --- |
| Read Data 1 (16 bits) | مقدار رجیستر خواندنی اول |
| Read Data 2 (16 bits) | مقدار رجیستر خواندنی وم |

*واحد محاسبات و منطق ALU*

کار این واحد انجام محاسبات منطقی و حسابی است. ورودیهای واحد ALU عبارتند از:

|  |  |
| --- | --- |
| X (16 bits) | ورودی 16 بیتی |
| Y (16 bits) | ورودی 16 بیتی |
| switch (S) (3 bits) | عملی که باید روی ورودیهای فوق انجام شود. |

خروجیهای ALU عبارتند از:

|  |  |
| --- | --- |
| signed overflow (1 bit) | سرریز |
| Result (16 bits) | نتیجه 16 بیتی |
| equal (1 bit) | اگر X و Y برابر باشند، این خروجی 1 خواهد شد. |

عملی که ALU باید انجام دهد از طریق ورودی سه بیتی switch معلوم می­گردد:

|  |  |
| --- | --- |
| **switch** | **معنی** |
| 0 | or: result = X | Y |
| 1 | and: result = X & Y |
| 2 | add: result = X + Y |
| 3 | sub: result = X - Y |
| 4 | sllv: result = X << Y |
| 5 | srlv: result = X >> Y (zero-fill) |
| 6 | srav: result = X >> Y (sign-fill) |
| 7 | slt: result = (X < Y) ? 1 : 0 (treat X and Y as *signed*) |

*پردازنده*

پردازنده به واحد حافظه وصل است و دستورات را از آنجا بر می­دارد. ورودیهای پردازنده عبارتند از:

|  |  |
| --- | --- |
| From Instr Mem (16 bits) | این ورودی توسط حافظه­ی دستور العمل تغذیه می­شود و 16 بیت است. |
| clk (1 bit) | کلاک |

خروجیهای پردازنده عبارتند از:

|  |  |
| --- | --- |
| d0 (16 bits) | خروجی LED |
| To Instr Mem (16 bits) | آدرس حافظه­ی دستورالعمل |
| Data Mem Data (16 bits) | داده­ای که قرار است در حافظه­ی داده نوشته شود. |
| Data Mem Addr (16 bits) | آدرس حافظه­ داده |
| Data Mem Write (1 bit) | سیگنال نوشتن در حافظه |

**مجموعه­ی دستورالعملها ISA**

دستورالعملهای این پردازنده 16 بیتی هستند. به علاوه، آدرسهای تولید شده توسط پردازنده نیز 16 بیتی هستند. لذا، ما عرض حافظه­ی دستورالعمل و داده را برابر 16 بیت در نظر می گیریم. به عنوان مثال، منظور ما از آدرس 000A یازدهمین خانه­ی حافظه است که یک کلمه­ی 16 بیتی در آن قرار دارد. در 10 خانه­ی قیلی (یعنی 0000 تا 0009) نیز 10 کلمه ی 16 بیتی قرار گرفته­اند. ساختار دستورالعمل ها در جدول زیر داده شده است. چهار بیت باارزش دستورالعمل نشان دهنده ی اسم آن است. دقت کنید تمام دستورات نوع R دارای کد صفر هستند و از طریق 6 بیت کم ارزش کار آنها مشخص می گردد که در جدول بعدی داده شده است.

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **15-12** | **11** | **10** | **9** | **8** | **7** | **6** | **5** | **4** | **3** | **2** | **1** | **0** |
| 0 | rs | | rt | | rd | | funct | | | | | | دستورات نوع R |
| 1 | rs | | rt | | immediate (unsigned) | | | | | | | | دستور disp rs برای نمایش مقدار rs روی نمایشگر |
| 2 | rs | | rt | | immediate (unsigned) | | | | | | | | دستور lui rt imm برای بار گذاری نیمه ی بالای rt با imm  $rt = imm << 8 |
| 3 | rs | | rt | | immediate (unsigned) | | | | | | | | دستور ori rt rs imm برایor کردن rs و imm و ذخیره ی نتیجه در rt  $rt = $rs | imm |
| 4 | rs | | rt | | immediate (signed) | | | | | | | | دستور addi rt rs imm  $rt = $rs + imm |
| 5 | rs | | rt | | immediate (unsigned) | | | | | | | | دستور andi rt rs imm مثل ori است. |
| 6 | target address | | | | | | | | | | | | دستور jal برای پرش به target و ذخیره­ی آدرس برگشت در $r3 |
| 7 | target address | | | | | | | | | | | | دستور jump برای پرش به target |
| 8 | rs | | unused | | | | | | | | | | دستور jr rs برای پرش به مقدار rs |
| 9 | rs | | rt | | offset (signed) | | | | | | | | دستور beq. این دستور شبیه MIPS است با این تفاوت که offset در 4 ضرب نمی شود. |
| 10 | rs | | rt | | offset (signed) | | | | | | | | دستور bne |
| 11 | rs | | rt | | immediate (signed) | | | | | | | | دستور lw rt imm(rs)  $rt = MEM[$rs + imm] |
| 12 | rs | | rt | | immediate (signed) | | | | | | | | دستور sw rt imm(rs)  MEM[$rs+imm] = $rt |

دستورات نوع R

|  |  |
| --- | --- |
| **funct** | **معنی** |
| 0 | or: $rd = $rs | $rt |
| 1 | and: $rd = $rs & $rt |
| 2 | add: $rd = $rs + $rt |
| 3 | sub: $rd = $rs - $rt |
| 4 | sllv: $rd = $rs << $rt |
| 5 | srlv: $rd = $rs >> $rt |
| 6 | srav: $rd = $rs >> $rt |
| 7 | slt: $rd = ($rs < $rt) ? 1 : 0 |

در ادامه راجع به بعضی دستورالعملها نکاتی ذکر می­شود. اولاً، بدستور بعدی در PC+1 قرار دارد نه در PC+4. در دستور jump مقدار target در 4 ضرب نمی­شود. هم چنین، 4 بیت باارزش آدرس پرش از روی PC+1 به دست می­آید. به عوان مثال اگر PC=F001 و target=2A3 باشد، آدرس پرش با F2A3 برابر است.

در دستورات beq و bne مقدار offset به صورت نسبی و علامت­دار در نظر گرفته می­شود و در 4 ضرب نمی­شود. هم­چنین، با PC+1 جمع می­شود. به عنوان مثال، beq $r3, $r1, 100 به این معنا است که اگر $r3 و $r1 برابر بوند به آدرس PC+101 پرش کنید. در غیر این صورت، دستور بعدی در PC+1 است.

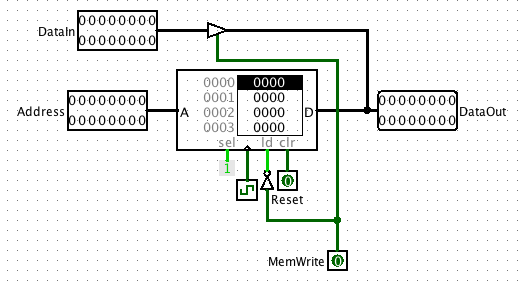
دقت کنید فیلد immediate  یک فیلد 8 بیتی است. لذا، لازم است که قبل از ارسال به ALU گسترش داده شود. در جاهایی که immediate  بدون علامت است، باید 8 صفر به آن اضافه شود. در جاهایی که immediate  دارای علامت باشد، باید طبق بیت علامت گسترش داده شود.

**شبیه ساز Logisim**

اگر در استفاده از Logisim دچار مشکل شدید، آن را دوباره اجرا و مدار را دوباره بارگذاری کنید!. وقت خود را با دنبال کردن باگهای این برنامه هدر ندهید. اگر با اجرای دوباره مشکل حل نشد احتمالا پروژه­ی شما دچار ایراد است.

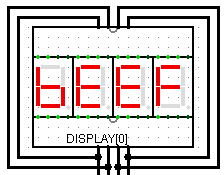
*ماژول RAM*

ماژول RAM در یک کتابخانه­ی پیش­ساخته­ی حافظ Logisim موجود است. از طریق "Project/Load Library/Built-in Library..." این مازول حافظه را به پروژه اضافه کنید. از RAM به عنوان حافظه­ی داده استفاده کنید. برای حافظه­ی دستورالعمل از ROM استفاده کنید.



از آنجا که ماژول RAM شبیه یک حافظه­ی ایده­آل نیست، ممکن است در استفاده از آن دچار مشکل شوید. شکل فوق نحوه­ی استفاده از RAM را نشان می­دهد.

* ورودی sel تعیین می­کند که آیا حافظه فعال است یا نه؟
* ورودی A مشخص کننده ی آدرس مورد نظر از حافظه است.
* ورودی clr تمام عناصر حافظه را برابر 0 قرار می­دهد.
* ورودی ld مشخص می­کند آیا می­خواهیم در حافظه بنویسیم یا از حافظه داده بخوانیم. اگر ld یک باشد، خروجی out برابر محتویات حافظه در خانه­ی A خواهد بود. اگر ld صفر باشد، داده­ی D در خانه­ی A حافظه نوشته می­شود.
* پورت D هم به عنوان ورودی عمل می­کن و هم به عنوان خروجی. با استفاده از یک بافر سه حالته کاری کرده­ایم که بین DataIn و DataOut هیچ تعارضی پیش نیاید.
* می­توانید از ابزار poke برای تغییر محتوی حافظه استفاده کنید. همچنین، می­توانید روی حافظه کلیک راست کنید و برای بارگذاری مقادیر حافظه از یک فایل گزینه­ی Load Image را انتخاب کنید.



*نمایش دستورات و سرریز از طریق LED ها*

برای راحتی کار شما یک فایل به اسم [converter.circuit](https://inst.eecs.berkeley.edu/~cs61c/sp11/projects/04/Converter.circ) به شما داده می­شود که 16 بیت را به 4 رقم هگزادسیمال تبدیل می­کند و روی چهار LED نمایش داده می­شود. به یاد بیاورید که دستور disp مقدار $rs را روی LED ها نمایش می­دهد. در هنگام شروع به کار CPU، مقدار 0000 را نمایش دهید. به علاوه، یک LED دارید که اگر سریز رخ دهد روشن می­شود.

**آزمایش**

برای ازمایش پردازنده، از برنامه­ی ساده­ی زیر استفاده کنید:

lui $r0, 0x33 2033

ori $r0, $r0, 0x44 3044

lui $r1, 0x33 2133

ori $r1, $r1, 0x44 3544

self: beq $r0, $r1, self 91FF

کد اسمبلی برنامه در سمت راست داده شده است. چک کنید که اسمبلر کد را درست تولید کند. این برنامه CPU را در یک حلقه­ی بی­نهایت قرار می­دهد. باید این برنامه را در ROM قرار دهید و پردازنده را روشن کنید. برای تست CPU، برنامه­های زیر را بنویسید:

* MEM[0] را با MEM[1] جمع کرده و نتیجه را در MEM[3] قرار دهید.
* برنامه­ای بنویسید که از بین MEM[3]، MEM[4] و MEM[5] مقدار کوچکتر را پیدا کرده و نمایش دهد.

**اسمبلر**

برای این که بتوانید دستورات را به زبان ماشین ترجمه کنید یک اسمبلر نوشته شده است که می­توانید آنرا از [اینجا](https://inst.eecs.berkeley.edu/~cs61c/sp11/projects/04/assembler.py) دنلود کنید. ورودی اسمبلر شبیه کد زیر است:

lui $r0, 85

ori $r0, $r0, 68

lui $r1, 85

ori $r1, $r1, 68

self: beq $r0, $r1, self

برای فراخوانی اسمبلر از پیتون استفاده کنید و دستور python assembler.py input.s [-o output.hex]  را اجرا کنید.

*تفاوتهای اساسی با* **MIPS**

1. رجیستر $r0 خاص نیست و مقدار آن صفر نیست.
2. هر کلمه­ی حافظه 16 بیت است و آدرسهای حافظه نیز 16 بیتی هستند.
3. به جای 32 رجیستر 4 رجیستر داریم.
4. حافظه­ی داده و دستورالعمل جدا هستند.
5. دستور jr از نوع r نیست. اینکار برای راحتی شما انجام شده است.

*برنامه­ریزی برای انجام پروژه*

اگر احساس می­کنید انجام این پروژه سخت است، می­توانید از مراحل زیر پیروی کنید:

1. ابتدا کار با RAM را در Logisim یاد بگیرید.
2. ابتدا نحوه­ی کار با رجیسترها در Logisim را یاد بگیرید و بعد سراغ پیاده­سازی بانک رجیستر بروید.
3. ALU را طراحی کنید.
4. ابتدا واحد مسیر داده را با حل تمرین یا استاد نهایی کنید. سپس، آنرا پیاده کنید.
5. واحد کنترل را طراحی کنید.
6. واحد کنترل و مسیر داده را به هم وصل کنید.
7. هر دستورالعمل را جداگانه آزمایش کنید.
8. چند برنامه­ی ساده بنویسید.