Projeto CPU

Visão Geral

Nesse projeto você usará o Logisim Evolution para implementar um processador 16 bits de dois ciclos. Esse projeto deve lhe dar um melhor entendimento sobre o datapath do MIPS.

O Processador

Nós daremos lhe um arquivo base `cpu.risc`. Seu processador irá conter uma instância da ALU e do Register File, desenvolvidos no projeto anterior, bem como uma unidade de memória, já presente no starter kit.

Você será responsável, no entanto, por construir todo o datapath e controle do zero. O processador deve implementar a ISA detalhada abaixo na seção *RPIS ISA* usando uma pipeline de dois ciclos.

Seu processador obterá o programa a ser executado do arquivo de testes `run.circ`. O processador irá dar como saída o endereço da próxima instrução a ser recebida na entrada. Dê uma olhada no `run.circ` para ver exatamente o que está acontecendo (mas note que esse arquivo não é parte da submissão).

Seu processador terá duas entradas que virá do arquivo de testes:

ENTRADA	BITS	DESCRIÇÃO
INSTRUCTION	16	Contem a instrução no endereço de memória identificado por `FETCH_ADDRESS` (ver abaixo).
CLOCK	1	O clock do processador. Este sinal pode, e deve ser passado para subcircuitos (e.g. a entrada `CLK` do register file ou a entrada de clock da únidade de memória). Entretanto, o sinal não deve ser passar por qualquer combinação lógica (i.e. não inverta-o, não coloque-o em um AND, etc.).

Seu processador deverá produzir seis saídas para o arquivo de testes:

SAÍDA	BITS	DESCRIÇÃO
$s0	16	O conteúdo de `$s0` (PARA TESTES).
$s1	16	O conteúdo de `$s1` (PARA TESTES).
$s2	16	O conteúdo de `$s2` (PARA TESTES).
$ra	16	O conteúdo de `$ra` (PARA TESTES).
$sp	16	O conteúdo de `$sp` (PARA TESTES).
FETCH_ADDRESS	16	Essa saída é usada como o endereço em que se encontra a próxima instrução a ser lida e recebida na entrada chamada `INSTRUCTION`.

Lembre-se: Não mova a posição das entradas e saídas ou o arquivo de testes pode quebrar! Tenha certeza de que os testes funcionam antes de realizar a submissão.

Memória

A unidade de memória já está completamente implementada! A seguir está detalhado seu esquema de entrada e saída:

NOME	IN ou OUT	DESCRIÇÃO
A: ADDR	IN 16	Endereço a ser lido/escrito na memória.
D: WRITE DATA	IN 16	Valor a ser escrito na memória.
En: WRITE ENABLE	IN 1	Igual a 1 para fazer uma operação de escrita, e 0 para fazer uma operação de leitura.
Clock	IN 1	Clock (obtido através da entrada de clock para o `cpu.circ`).
D: READ DATA	OUT 16	Dado lido/escrito do endereço especificado.

Pipeline

Seu processador terá um pipeline de dois estágios:

1. **Instruction Fetch:** Uma instrução é obtida da memória de instruções.
2. **Execute:** A instrução é decodificada, executada e commitada (write-back). Esse estágio é uma combinação dos estágios restantes de uma pipeline padrão MIPS.

Você deve notar que dependências de dados NÃO são um problema nesse design, visto que todos os acessos a memória de dados acontecem em apenas um estágio do pipeline. No entanto, ainda é necessário lidar com dependências de controle.

Nossa ISA não expõe branch delay slots ao software! Em outra palavras, se um branch for tomado, a instrução imediatamente a seguir não é executada. Isso torna a sua tarefa um pouco mais complicada. Quando você descobrir que há um branch no estágio de execução, você já acessou a memória de instruções e pegou a próxima instrução, provavelmente errada. Você precisa, portanto, matar a instrução que está no estágio de fetch caso a instrução no execute é um jump ou branch tomado. **O ato de matar instruções nesse projeto deve ser realizada através de uma instrução NOP em um MUX que será dirigido ao estagio de execução, escolhendo-a em vez da instrução lida.** Note que `0x0000` é um `nop`. Use este fato a seu favor.

Você só deverá matar uma instrução se um branch for pego! Não mate-a em caso contrário. Mas claro, sempre mate para os jumps.

Como seu processador fará toda a parte de controle e execução no estágio de execução, ele deve ser quase indistinguível de uma implementação de ciclo único, exceto pela latência de um ciclo para ligar a CPU e os delays dos jumps e branches. Ainda assim, queremos um design de dois estágios. Se está incerto sobre como fazer um pipeline, é perfeitamente aceitável (e até mesmo recomendado) que você implemente uma versão de ciclo único, para que possa validar sua lógica de decodificação, controle e execução. Em seguida, modifique o design para que se torne um pipeline de dois ciclos. Algumas coisas a se levar em consideração:

- Os estágios IF e EX terão PCs iguais ou diferentes?
- Você precisa guardar os PCs entre os estágios?
- Para multiplexar um `nop` no fluxo das instruções, você colocará o MUX antes ou depois do registrador de instrução?
- Qual endereço deve ser obtido em seguida enquanto você executa o estágio EX com um `nop`? É diferente do usual?

Você também notará um problema de bootstrapping: Durante o primeiro ciclo, o registrador de instruções que está entre os estágios do pipeline não terão uma instrução da memória. Como lidar com isso? Felizmente, o comportamento do Logisim é de conter um valor zero no inicio da simulação, ou seja um `nop`. Nós aceitaremos a dependência nesse comportamento. Lembre-se de sempre resetar a simulação em *Simulate -> Reset Simulation (CTRL+R)*.

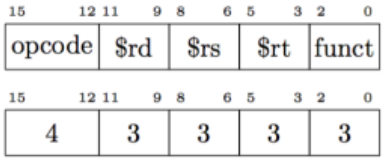
Controle

O Conjunto de Instruções RPIS (RPIS ISA)

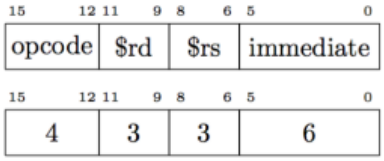
Sua CPU irá suportar o conjunto de instruções listado abaixo. Há diferenças importantes a serem consideradas entre MIPS e RPIS, e essas serão explicadas mais abaixo. Por favor tenha certeza que entendeu o novo formato das instruções antes de começar a implementação!

Formato das Instruções

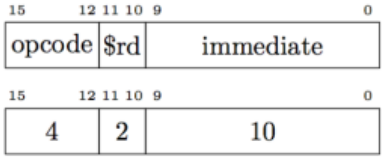
R-Type



I-Type



U-Type



As Instruções

OPCODE/ FUNCT	TIPO	INSTRUÇÃO	FORMATO	COMPORTAMENTO
0x0	R	Shift Left Logical	sll $rd, $rs, $rt	R[rd] = R[rs] << R[rt][4:0]
0x1	R	Shift Right Logical	srl $rd, $rs, $rt	R[rd] = R[rs] >>> R[rt][4:0]

OPCODE/ FUNCT	TIPO	INSTRUÇÃO	FORMATO	COMPORTAMENTO
0x2	R	Add	add $rd, $rs, $rt	R[rd] = R[rs] + R[rt]
0x3	R	And	and $rd, $rs, $rt	R[rd] = R[rs] & R[rt]
0x4	R	Or	or $rd, $rs, $rt	R[rd] = R[rs] | R[rt]
0x5	R	Exclusive Or	xor $rd, $rs, $rt	R[rd] = R[rs] ^ R[rt]
0x6	R	Set Less Than (Signed)	slt $rd, $rs, $rt	R[rd] =(R[rs] < R[rt])? 1 : 0
0x7	R	Multiply (unsigned)	mult $rs, $rt	HI = (R[rs] * R[rt])[31:16]; LO = (R[rs] * R[rt])[15:0];
0x1	U	Load Upper Immediate	lui imm	LO = {imm,0b000000} = imm << 6
0x2	U	Jump (and Link)	j $rd, label	PC = JumpAddr; R[rd] = PC+2; JumpAddr = {PC[15:10],imm};
0x3	I	Jump (and Link) Register	jr $rd, $rs, imm	PC = R[rs] + SignExt(imm); R[rd] = PC+2;
0x4	I	Branch on Equal	beq $rs, $rd label	if(R[rs] == R[rd]) PC = BranchAddr; BranchAddr = PC + SignExt(imm)
0x5	I	Branch on Not Equal	bne $rs, $rd label	if(R[rs] != R[rd]) PC = BranchAddr; BranchAddr = PC + SignExt(imm)
0x6	I	Add Immediate	addi $rd, $rs imm	R[rd] = R[rs] + SignExt(imm)
0x7	I	Set Less Than Immediate	slti $rd, $rs, imm	R[rd] = R[rs] < SignExt(imm)? 1 : 0
0x8	I	And Immediate	andi $rd, $rs imm	R[rd] = R[rs] & ZeroExt(imm)
0x9	I	Or Immediate	ori $rd, $rs imm	R[rd] = R[rs] | ZeroExt(imm)
0xa	I	Load Half Word	lh $rd, offset($rs)	R[rd] = M[addr]; addr = R[rs] + SignExt(offset)
0xb	I	Store Half Word	sh $rd, offset($rs)	M[addr] = R[rd]; addr = R[rs] + SignExt(offset)
0xc	I	Move From High	mfhi $rd	R[rd] = HI
0xd	I	Move From Low	mflo $rd	R[rd] = LO

Diferenças Notáveis Em Relação a MIPS

Jumps e Linking

Em RPIS, há apenas duas instruções de branching: jump e jump register. Para o jump, o endereço de salto é calculadora pegando os 10 bits imediatos na instrução, e então preconcatenando os 6 bits mais significativos de PC. Em outras palavras: Seja `','` um operador de concatenação, então `JumpAddr = {PC[15:10], immediate}`. Por exemplo, se o valor imediato for `0x1a7 = 0b0110100111` e o PC for `0xabcd = 0b1010101111001101`, então `JumpAddr = {101010, 0110100111} = 0b1010100110100111 = 0xa9a7`.

Para o jump register, o imediato será também usado para calculador o endereço de jump, visto que esta é agora uma instrução do tipo I. Mais especificamente, PC será alterado para o valor de `$rs` mais o valor imediato, ou seja `PC = JumpAddr = R[rs] + immediate`. Se você deseja pular para o endereço exato de `$rs`, como o jump register de MIPS, o valor do imediato deverá ser zero.

Em vez de ter mais uma versão dessas duas instruções para linking, o link é feito sempre através do registrador `$rd` (em MIPS o link é sempre feito com `$ra`). Isto é, se `$rd` é, por exemplo, `$s0`, o PC atual será salvo em `$s0` (em MIPS, sempre seria salvo em `$ra`). Entretanto, note que se `$rd` é o registrador zero, então de certa forma o PC atual não é salvo.

Portanto, para obter o mesmo comportamento do `jalr $s0` do MIPS, você faria `jr $s0, $ra, 0`, de tal forma que `R[ra] = PC` e então `PC = R[s0]`. Você deve ter notado que o campo `$rd` das instruções do tipo U tem apenas 2 bits, o que significa que no jump, o `$rd` pode ser apenas os registradores 0-3, ou seja `$0`, `$ra`, `$s0` e `$s1`.

Branches

No RPIS, os alvos das instruções de jump são offsets em bytes em relação ao PC, já em MIPS são offsets em palavras. Por esse motivo, o bit menos significativo do imediato sempre será 0, gastando um bit de espaço. Todavia, a implementação é muito mais fácil dessa forma.

Load Upper Immediate

Já que nossas instruções de tipo I só podem ter 6 bits imediatos, não seriamos capazes de carregar um inteiro de 16 bits. Por esse motivo, em RPIS, `lui` é uma instrução do tipo U, que suporta imediados de 10 bits. Tal valor de 10 bits será carregado no topo do registrador especial `$LO` (e então `mflo` poderá ser usado para carregar o valor de volta para um registrador normal). A sequência de instruções a seguir pode ser usada para carregar um imediato de 16 bits (e.g. `0xABCD = 0b1010,1011,1100,1101`) no registrador `$s0`.

```
lui 0b1010101111
mflo $s0
ori $s0, $s0, 0b001101
```

Para a instrução `lui`, portanto, o valor do campo `$rd` é inutilizado e poderá ser qualquer coisa.

Multiplicação

Como em MIPS, a instrução de multiplicação irá guardar seu resultado em `$HI` e `$LO`. Isto é, os 16 bits mais significativos serão guardados em `$HI`, e os outros 16 menos significativos em `$LO`. Em seguida, `mfhi` e `mflo` podem ser usados para obter os valores de `$HI` e `$LO`, respectivamente.

Set Less Than

Em RPIS, Set Less Than é equivalente a `slt` do MIPS, onde o valor de `$rs` e `$rt` são interpretados como inteiros com sinal. Sua ALU já deve tratar esse caso sem demais problemas.

Observações sobre o logisim

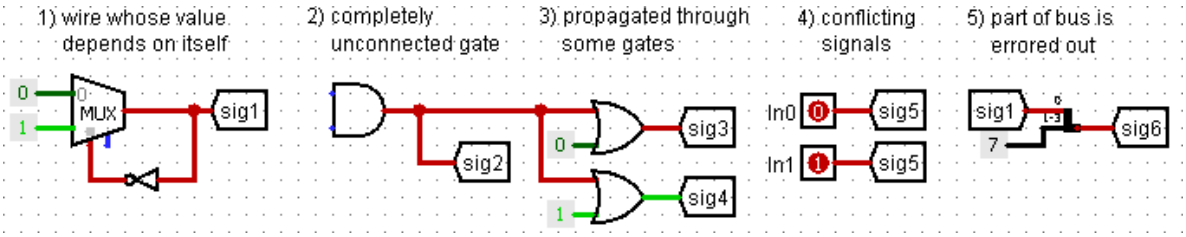
Se você estiver com bugs no Logisin, FECHE e ABRA o circuito novamente! Não perca seu tempo tentando reparar um problema que não é seu. No entando, caso reiniciar não tenha resolvido o problema, é provável que exista algum problema de implementação em seu circuito.

Caso o problema continuar após modificações, tendo certeza que não é um problema seu, entre em contato:

- Denilson Amorim: denimorim@gmail.com
- Thalles Medrado: thallesyann (at) hotmail (dot) com

Ainda sobre o logisim, vale a pena ressaltar os seguintes pontos:

- NÃO ligue o clock em portas lógicas! Esta é uma péssima prática de design na montagem de circuitos, não aconselhamos que faça algo assim e, por isso, iremos penalizar seu projeto.
- Seja CAUTELOSO ao copiar e colar circuitos entre janelas do LOGISIM. O logisim é conhecido por problemas com este tipo de tarefa.
- Quando você importa outro arquivo (Project -> Load Library -> Logisim Library ...), ele aparecerá como uma pasta no painel de visualização da esquerda. Os arquivos de esqueleto já devem ter importado os arquivos necessários.
- Alterar atributos antes de setar um componente altera as configurações padrões do componente. Então, se você for colocar muitos pinos de 32 bits, pode ser útil alterar sua configuração padrão. Se quiser alterar apenas esse componente específico, coloque o componente e depois altere sua configuração.
- Quando você altera as entradas e saídas de um subcircuito que você já colocou no main, o Logisim irá adicionar/remover automaticamente as portas quando retornar ao main e isso, por vezes, muda o próprio bloco. Se haviam fios conectados, o Logisim também fará o deslocamento automático destes, o que pode ser extremamente ruim em alguns casos. Antes de alterar as entradas e saídas de um bloco, às vezes pode ser melhor separar primeiro todos os fios dele no main.
- Os erros de sinal (fios vermelhos) são obviamente ruins, mas tendem a aparecer em tarefas de fiação complicadas, como as que você estará implementando aqui. É bom estar ciente das causas comuns durante a depuração:



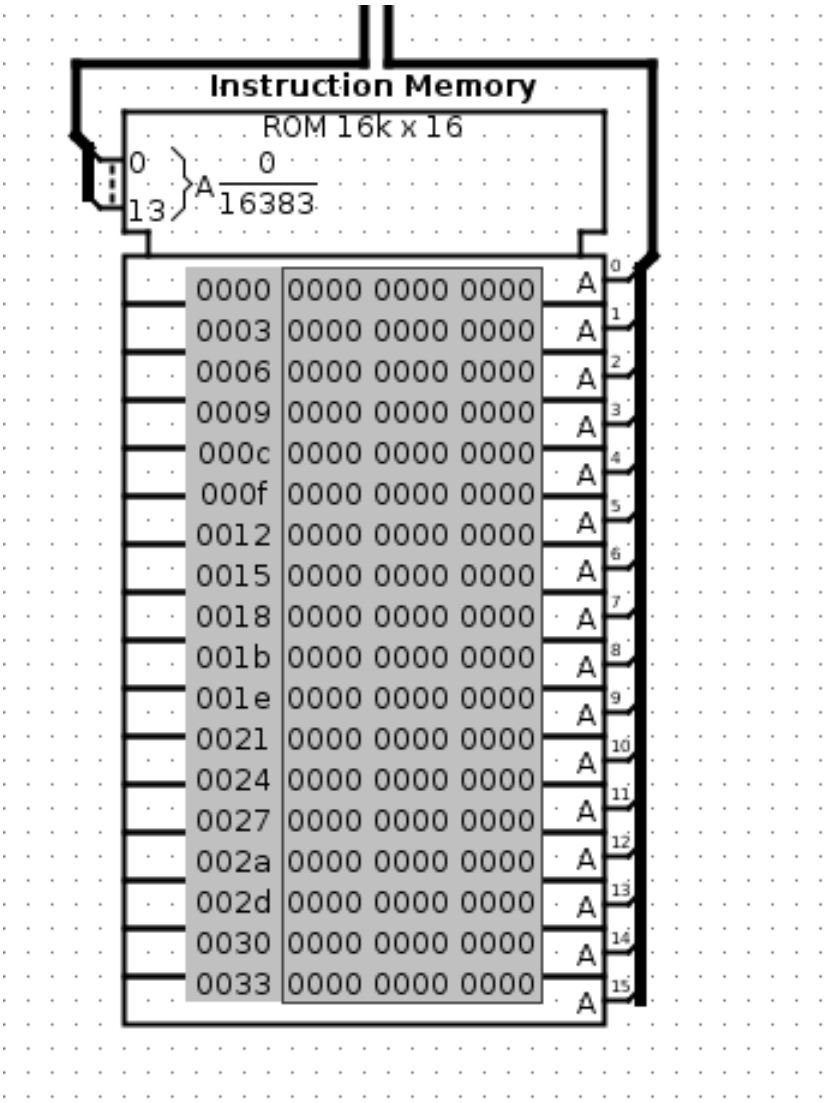
- O Logisim oferece algumas funções para automatizar a implementação do circuito dada uma tabela de verdade, ou vice-versa. Embora permitido, o uso desse recurso é desencorajado. Lembre-se de que você não terá um laptop executando Logisim na prova final.

Testando

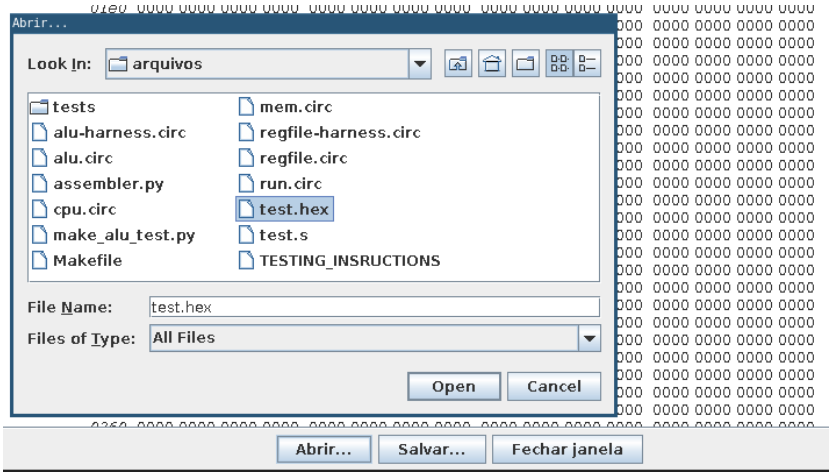
Para esse segundo trabalho, é um pouco mais complicado fazer uso de testes unitários similares ao do primeiro trabalho. Portanto, o melhor a se fazer é escrever pequenos programas em RPIS para exercitar seu datapath. Para isso, incluímos um assembler bastante rudimentar. Para compilar um programa RPIS, faça o seguinte:

```
# Assumindo que o seu programa está no arquivo test.s
python2 assembler.py test.s    # Irá produzir um arquivo test.hex, ou...
python2 assembler.py test.s -v # ...para uma saida mais verbosa na linha de comando
```

Uma vez que você tenha gerado o código de maquina, você precisará carregá-lo na memória de instruções no arquivo `run.circ` e começar a execução. Abra o arquivo `run.circ` e encontre a unidade de memória de instruções:



Clique nela e na sidebar da esquerda, clique na opção *(clicar para editar)* próximo a *Conteudos*. Isso vai abrir um editor hexadecimal com a opção de abrir um arquivo `.hex`. Clique em *Abrir* e então carrega o arquivo `.hex` previamente gerado.



Pronto, agora você pode ir clicando no clock e ver como a sua CPU se comporta no processo. Você pode clicar na CPU para ver o que está acontecendo dentro do sub circuito.

Além disso, você pode executar alguns testes automáticos executando o seguinte código na pasta raiz:

```
make cpu-single-cycle
```

Ou, caso tenha feito pipelined:

```
make cpu-pipelined
```

Submissão

Você deve submeter apenas os circuitos necessários para o funcionamento correto de sua `cpu.circ` (inclusive) dentro de um zip com o nome da equipe.

Ex.: Suponha que você modificou apenas o circuito `cpu.circ`, sua submissão `equipe.zip` deve conter apenas este circuito.

OBS.: Note que todos os arquivos necessários devem estar na raiz do zip, a submissão que não estiver de acordo estará sujeita a punição.

Avaliação

O processador será avaliado em maior parte por um sistema automático. Se algum dos seus testes falharem, tentaremos ver se é um problema simples, e então consertar. Nós então daremos a nota automática, menos um percentual pela gravidade do problema. Por esse motivo, tente deixar o circuito o mais legivel e funcional possível.

Poderá haver penalidade por:

- Circuito em desacordo com especificações do projeto;
- Plágio; Ambas equipes serão penalizadas!
- Submissão fora do formato especificado.

Somente o arquivo `cpu.circ` será considerado como parte da submissão. Nossa própria versão dos outros arquivos será utilizada para fins de avaliação, incluindo ALU e Register File.

Lembre-se de usar o **Logisim Evolution**! Caso contrário, seu circuito não será compatível com o sistema de testes.