Quarto Exercício-Programa (EP 4)

MAC329 – Álgebra Booleana e Aplicações

Marcelo S. Reis [1]
Junior Barrera [1]
Gabriela E.F. Sanada [1]

[1] Instituto de Matemática e Estatística, Universidade de São Paulo.
msreis@ime.usp.br

11 de junho de 2013

1 Introdução

Neste Exercício-Programa (EP) faremos o acabamento de nosso computador HIPO em Logisim ([1]), deixando-o pronto para executar qualquer código escrito em linguagem HIPO.

Para atingirmos os objetivos deste EP, deveremos fazer o acoplamento do código do ciclo de instrução implementado no EP 3 com o da Unidade Lógica e Aritmética (ALU) desenvolvida no EP 2. Além disso, implementaremos as operações de entrada/saída (I/O, *input/output*) do computador, assim como as instruções remanescentes do conjunto de instruções. Por fim, faremos uma pequena modificação no montador HIPO entregue no EP 1, de modo a permitir que um código escrito em alto nível seja testado no computador HIPO escrito em Logisim.

Após esta introdução, na seção 2 detalharemos como deverá ser feito o acabamento do computador HIPO em Logisim. Na seção 3 será descrita a modificação a ser feita no montador entregue no EP 1. Finalmente, encerraremos este enunciado com a data de entrega deste EP e outras observações importantes.

2 Acabamento do computador HIPO

Para o acabamento de nosso computador HIPO, basicamente precisaremos realizar três tarefas:

1. acoplar a ALU na arquitetura de ciclo de instrução;
2. implementar os desvios condicionais;
3. adicionar funcionalidades de entrada/saída (I/O).

A seguir, detalharemos cada uma dessas tarefas.

Código		Instrução	Descrição
Decimal	Hexadecimal		
21	0x15	{`ADD`} `XY`	Soma ACC com conteúdo do endereço `XY`
22	0x16	{`SUB`} `XY`	Subtrai conteúdo do endereço `XY` de ACC
23	0x17	{`MUL`} `XY`	Multiplica ACC pelo conteúdo do endereço `XY`
24	0x18	{`DIV`} `XY`	Divide ACC pelo conteúdo do endereço `XY`
25	0x19	{`REM`} `XY`	Resto da divisão de ACC pelo conteúdo do endereço `XY`

Tabela 1: descrição das instruções aritméticas que serão implementadas neste EP. Observe que o acumulador (ACC) é ligado na entrada A da ALU, que o conteúdo do endereço `XY` é ligado na entrada B da ALU, e que o resultado da operação é guardado no acumulador através da saída R da ALU.

2.1 Acoplamento da ALU no ciclo de instrução

Para acoplarmos a Unidade Lógica e Aritmética (ALU) desenvolvida no EP 2 com a arquitetura de ciclo de instrução do EP 3, duas modificações serão necessárias:

- As entradas e a saída da ALU passarão a ser de 16 bits, e não de 8 bits como no EP 2;
- O valor da saída da ALU somente será alterado se a mesma estiver habilitada pelo controlador. Isso permitirá, por exemplo, que após uma soma de dois números (operação que exige o acumulador em modo de leitura como uma das entradas da ALU), o valor resultante possa ser guardado no acumulador (operação que exige o acumulador em modo de escrita, ligado na saída da ALU).

Além da indicação de transbordamento (*overflow*), passaremos também a indicar a ocorrência de divisão por zero. Na ocorrência de transbordamento e/ou de divisão por zero, a execução do programa deverá ser imediatamente interrompida. Para as modificações da ALU, será permitido o uso de todos os módulos aritméticos do Logisim (e.g. *Adder*, *Divider*, etc.).

Além das modificações da ALU, para facilitar o acoplamento da mesma com o acumulador utilizaremos um multiplexador para definir se esse registrador lê um valor da saída da ALU ou do barramento da memória. Como uma das entradas da ALU é sempre o acumulador, a saída do mesmo deverá ser ligada diretamente na ALU, utilizando para isso um desmultiplexador. Na figura 1 mostramos um diagrama de como fazer o acoplamento da ALU com a arquitetura do ciclo de instrução.

O acoplamento da ALU com o ciclo de instrução nos permitirá a implementação das instruções correspondentes às cinco operações aritméticas. Na tabela 1 mostramos quais são essas operações.

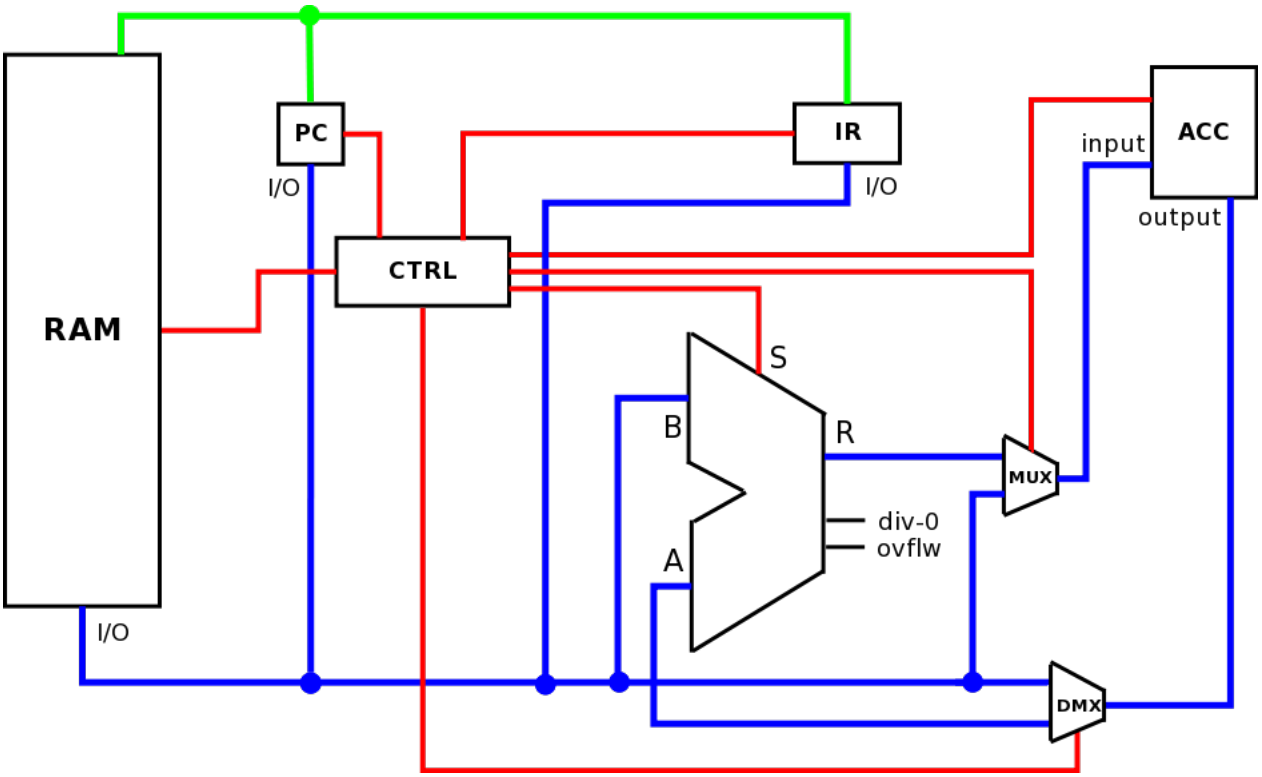

Figura 1: diagrama de acoplamento da ALU com a arquitetura do ciclo de instrução. Os barramentos de dados, de endereço e de controle são indicados, respectivamente, em azul, verde e vermelho. A operação aritmética a ser executada (`S`) é determinada pelo controlador (`CTRL`). Uma das entradas da ALU é sempre o conteúdo do acumulador (`ACC`; portanto, o desmultiplexador na saída do acumulador deverá ter setado como saída a entrada `A` da ALU) e a outra é sempre o conteúdo de uma determinada posição de memória que esteja no momento no barramento de dados. Observe que no caso de transbordamento (`ovflw`) e/ou de divisão por zero (`div-0`) a execução do computador deve ser imediatamente interrompida.

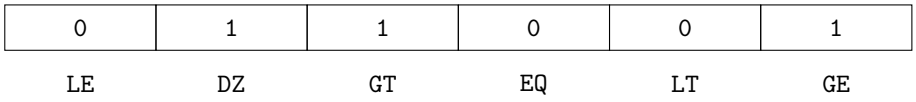

Figura 2: exemplo do estado que o registrador auxiliar assume, para o caso em que o acumulador é atualizado com o valor `+3217`. Observe que esse valor é diferente de zero (DZ), maior que zero (GT) e maior ou igual a zero (GE), o que implica que os respectivos bits dessas condições são setados.

Código		Instrução	Descrição
Decimal	Hexadecimal		
52	0x34	{`JLE`} `XY`	Desvia para a instrução apontada por `XY` se ACC contém um valor menor ou igual a zero
53	0x35	{`JDZ`} `XY`	Idem, diferente de zero
54	0x36	{`JGT`} `XY`	Idem, maior que zero
55	0x37	{`JEQ`} `XY`	Idem, igual a zero
56	0x38	{`JLT`} `XY`	Idem, menor que zero
57	0x39	{`JGE`} `XY`	Idem, maior ou igual a zero

Tabela 2: descrição das instruções de desvio condicional que serão implementadas neste EP.

2.2 Desvios condicionais

As instruções de desvios condicionais serão implementadas com o auxílio de um registrador auxiliar, de 6 bits. Cada bit é uma *flag*, e indica se uma dada condição de salto é satisfeita ou não (veja figura 2). O registrador auxiliar deverá ser inicializado supondo que o acumulador foi carregado com o valor zero, e a partir daí deverá ser atualizado sempre que o valor do acumulador mudar. Cada *flag* deverá ser atualizada empregando um circuito combinatório próprio (e.g., a atualização da flag EQ, na prática, é uma operação NOR sobre todos os 16 bits que compõem o valor do acumulador).

Dessa forma, cada desvio condicional opera como se fosse um desvio não-condicional que ocorre somente se a sua respectiva *flag* estiver setada. Na tabela 2 mostramos todos os desvios condicionais que serão implementados neste EP.

2.3 Entrada / Saída

Para a entrada/saída, a utilização de pinos de 16 bits já será suficiente. Para implementar a instrução de leitura (escrita), proceda de forma similar ao que foi feito para implementar a instrução `STA` (`LDA`), com a diferença de que estaremos lidando com o pino de entrada (saída)

Código		Instrução	Descrição
Decimal	Hexadecimal		
31	0x1F	{`INN`} `XY`	Lê da entrada e armazena na posição `XY`
41	0x29	{`PRN`} `XY`	Escreve na saída o conteúdo da posição `XY`

Tabela 3: descrição das instruções de entrada/saída que serão implementadas neste EP.

ao invés do acumulador. Poderemos sempre supor que o valor correto a ser inserido já esteja nos pinos de entrada imediatamente antes do início do ciclo de instrução de seu respectivo `INN`.

Tanto na entrada quanto na saída os valores deverão ser fornecidos em hexadecimal com complemento de dois. Na tabela 3 são detalhadas as operações de I/O.

Opcional (vale bônus na nota). Implemente formas mais sofisticadas de I/O, utilizando teclado na entrada, display na saída, ou ainda fazendo a conversão de base (i.e., lê em decimal na entrada, converte para hexadecimal para processamento interno e converte de hexadecimal para decimal para exibir no display de saída) e/ou o processamento de sinal (i.e., o número é inserido/exibido de forma sinalizada e uma (des)conversão de complemento de dois faz-se necessária).

2.4 Recomendações e dicas

- Por questões de simplicidade, poderemos supor que todos os endereços de memória são válidos (i.e., embora as posições de memória do computador HIPO vão de 0 até 99, poderemos trabalhar com uma memória RAM cujas posições de memória vão de 0 até 255). Da mesma forma, poderemos supor que os valores de dados sempre são números inteiros de 16 bits sinalizados, o que generaliza o intervalo $[-9999, +9999]$. Portanto, a nossa arquitetura em Logisim é uma generalização da especificação original do computador HIPO.
- Da mesma forma que no item anterior, poderemos supor que todas as instruções de um programa armazenado na memória RAM sejam válidas (i.e., não será preciso tratar o caso de instrução inválida).

3 Refatoração do montador

Para testarmos o nosso computador HIPO, faremos uma pequena modificação no montador implementado no EP 1: ao invés da saída produzir um código em linguagem de máquina HIPO, a mesma passará a gerar um código hexadecimal compatível com a nossa implementação em Logisim. Por exemplo, dado o seguinte arquivo de entrada:

```
; apos o ponto e virgula temos comentarios
;
00 {LDA} 10    ; soma valores
01 {ADD} 11
02 {STA} 12    ; prepara a impressao
03 {PRN} 12
04 {STP}       ; fim do programa
;
10 -7770       ; termos da soma
11 +7777
```

O montador deverá produzir a seguinte saída:

```
0B0A 150B 0C0C 290C 4600 0000 0000 0000 0000 0000 E19F 1E61
```

Observe que todos os endereços e dados são convertidos para hexadecimal, e que os dados também utilizam complemento de dois. Dessa forma, para testarmos um programa HIPO, bastará escrevermos o mesmo em alto nível, utilizarmos o montador para produzirmos um código hexadecimal e colarmos este último no módulo RAM de nosso computador[1].

4 Entrega do EP e observações

4.1 Observações importantes

- Este EP deverá ser feito em grupos de 4 pessoas, de preferência mantendo a mesma equipe do EP anterior; em casos excepcionais e com autorização do professor / monitor, pode-se aceitar grupos com menos integrantes.
- Organização do código é muito importante; para isso divida o seu circuito principal em subcircuitos que serão utilizados pelo circuito principal (e/ou por outros subcircuitos). Utilize *splitters* para diminuir a poluição visual dos barramentos e encapsule subcircuitos em contextos que exigem visualizá-los em um nível de abstração mais elevado.
- O montador deve ser compilável sem erros ou avisos, quando utilizado o compilador `gcc` com as opções `-ansi -pedantic -Wall`. Portanto, façam uso dessas opções para testar a corretude do código.

[1] A propósito, o pequeno programa HIPO que desenvolvemos no EP 1 é uma boa opção para testar tanto o montador modificado quanto o computador HIPO em Logisim.

- Documente adequadamente o seu circuito, através de comentários em um arquivo texto README que deverá acompanhar o código-fonte. Faça também (breves) comentários no próprio circuito, para facilitar o entendimento do mesmo.

- Recomenda-se que as dúvidas sejam discutidas no fórum da disciplina no Moodle.

4.2 Data de entrega

Este EP deverá ser entregue até o dia **30 de junho (domingo)**, através do fórum da disciplina no Moodle. Após esse dia, o sistema não permitirá a entrega deste EP.

Deverá ser entregue um zip ou tar.gz, contendo:

- o arquivo `.circ` do código do computador HIPO descrito na seção 2;

- o arquivo `.c` do código do montador HIPO refatorado de acordo com a seção 3;

- um arquivo README, contendo comentários que acharem pertinentes para o entendimento da implementação;

- um arquivo FEEDBACK, relatando a experiência do grupo na execução dos EPs: pontos positivos, negativos e sugestões de melhoramentos para versões futuras do curso.

Apenas um membro do grupo deverá subir o arquivo; portanto, lembrem-se de colocar no arquivo README o nome completo e o número USP de todos os integrantes do grupo.

Referências

[1] Carl Burch. Logisim – a graphical tool for designing and simulating logic circuits. `http://ozark.hendrix.edu/~burch/logisim/index.html`, 2001.

[2] Valdemar W. Setzer. The HIPO computer – a tool for teaching basic computer principles through machine language. `http://www.ime.usp.br/~vwsetzer/hipo/hipo-descr.html`, 2000.