PCS3115 - Sistemas Digitais I - Trabalho 4

por Bruno de Carvalho Albertini

15/11/2021

Neste trabalho você montará um fluxo de dados a partir de componentes que já fez e desenvolverá uma unidade de controle para o processador PoliStack.

Introdução

O PoliStack é um processador didático baseado em um processador comercial. Sua arquitetura é de pilha, ou seja, todos os operandos estão na memória e não há banco de registradores. Este tipo de máquina de pilha programável é considerado um processador rudimentar, mas ainda é bastante utilizado pois a síntese pode ser otimizada para que ocupe muito pouca área e gaste muito pouca energia.

A ULA sempre opera através da pilha.

A chave para tal tipo de processador é a memória, que deve ser uma memória de duas portas com acesso simultâneo, o que pode ser feito usando uma memória deste tipo ou emulando este comportamento através de um controlador de memória. A memória de duas portas usada no PoliStack possui duas portas, chamadas de `A` e `B`. Cada porta de memória é independente e possui uma entrada de endereço e uma saída de dados.

A 70T653M da Renesas é um exemplo de memória comercial *dual-port*.

A memória responde assincronamente colocando os valores das posições de memória apontadas pelo endereço na saída da porta correspondente. A escrita é realizada através de uma entrada de dados, porém a escrita **não** é dupla: somente o dado da porta de entrada (única para a memória) é escrito. O endereço de escrita é compartilhado com o a porta `B`, então as operações possíveis em um determinado ciclo são: leitura da porta `A` e leitura ou escrita na porta `B`. Os sinais tradicionais de memória estão presentes: `EN` (*enable*) desabilita a memória toda (nenhuma operação é realizada e as saídas ficam em alta impedância), `WE` (*write enable*) habilita a escrita, e `BSY` (*busy*) indica que a memória está realizando uma operação. A memória pode demorar vários ciclos de *clock* para realizar a operação, então o processador precisa obedecer os sinais. Um exemplo: processador coloca os endereços nas entradas de endereço das portas, se é uma escrita levanta o `WE` e finalmente levanta o `EN`. No máximo após a próxima borda de subida a memória levantará o `BSY` e começará a realizar a operação, quando o processador abaixa o `WE` (se escrita), mas **não** abaixa o `EN`. Quando a memória terminar a operação, os dados lidos estarão nas saídas de dados das portas correspondentes (se for escrita o dado escrito também é colocado na porta de leitura) e a memória abaixa o `BSY`, quando então o processador pode abaixar o `EN`. Se o `EN` for abaixado durante uma operação (ou seja, com `BSY` alto), o comportamento da memória é imprevisível.

Descrição do acesso à memória.

Como a memória é única neste sistema, não é necessário trabalhar com alta impedância.

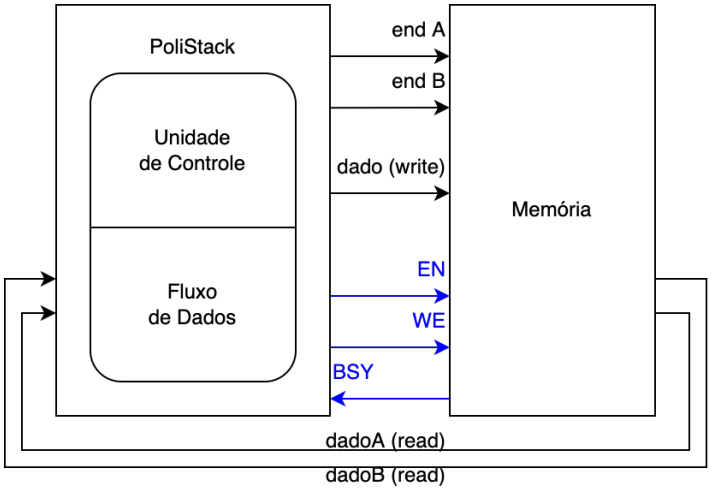

Figura 1: Conexões entre o processador e a memória no PoliStack. Em preto os dados e em azul o controle.

Neste trabalho, a memória é endereçada em bytes, com palavras de 32b (4B) e permite acesso desalinhado. O PoliStack que construiremos possui instruções de 1B (sempre) e largura de dados de 32b (para nossa implementação somente, pois a nossa memória tem palavras de 32b), com alguns registradores refletindo a largura do barramento de endereços (e.g. se a memória possui barramentos de endereços de n bits, os registradores que trabalham com endereçamento também possuem a mesma largura de n bits).

Fluxo de Dados

O Fluxo de Dados do PoliStack é simpplificado pois é uma máquina de pilha: não há bancos de registradores. Os operadores sempre vêm da memória e correspondem ao topo da pilha e ao segundo elemento da pilha para o caso de operações que exigem dois operandos. O resultado é sempre guardado no topo da pilha.

Registradores

Todos os três registradores do PoliStack possuem *enable* e são síncronos na borda de subida, possuem reset assíncrono com valor configurável e possuem entrada e saída paralela (registrador simples). O elemento `d_register` do trabalho 2 pode ser usado para instanciar estes registradores.

Program Counter

O PC (*Program Counter*) reseta em zero e é incrementado um a um a cada ciclo (exceto em instruções de desvio onde pode ser sobrescrito por um valor determinado). Como a memória é organizada em bytes e as instruções são de 1B, o PP aponta exatamente para umaa instrução, ou seja, na palavra de memória de 4B obtida na leitura da memória apontada pelo PC, a instrução sempre está no byte menos significativo. Exemplo: se `mem[PC]=0x0b0b7082`, a instrução a ser executada é `0x82`.

Stack Pointer

O SP (*Stack Pointer*) é um registrador que aponta sempre para o topo da pilha. Como a pilha começa no final da memória RAM

e cresce no sentido do começo da memória, este registrador reseta em `0x1FFF8`. Neste trabalho vamos usar endereçamento de 17 bits (a memória tem 2^{17} posições de 1B). Como os operandos são de 4B, o SP começa com um valor que representa o final de memória mas com espaço para armazenar um operando se necessário, daí o valor de reset. Para empilhar um valor, primeiro subtrai-se 4 do SP, abrindo espaço na pilha para um operando, e depois fazemos `mem[SP]=valor`. O valor estará na posição de memória apontada por SP. A entrada deste registrador é permanentemente ligada na saída da ULA.

Um operando tem exatamente 4B.

Por último, o IR (*Instruction Register*) é sempre de 8b (pois todas as instruções do PoliStack são de 8b) e reseta em zero. Este registrador é necessário pois durante a execução da instrução, pode ser necessário que a unidade de controle consulte a instrução sendo executada (por exemplo para extrair um imediato), mas também precise usar o acesso à memória. É uma maneira de manter a instrução disponível durante toda a sua execução sem precisar ocupar a memória. O valor desse registrador sofre alteração sempre que o processador for executar uma nova instrução e representa a instrução completa sendo executada. A entrada sempre vem da porta A da memória (`memA_rdd`) e somente o byte menos significativo da palavra lida é usado.

Instruction Register

Outros Componentes

Além dos três registradores, o fluxo de dados conta com uma ULA e com vários multiplexadores que escolhem os valores para os demais componentes de acordo com os sinais de controle enviados pela unidade de controle. A ULA é a mesma do Trabalho 2, e sua implementação pode ser usada para montar o fluxo de dados. Os multiplexadores são facilmente implementáveis usando atribuições condicionais do tipo `when-select` ou `when-else`.

Todos os multiplexadores existentes no fluxo de dados estão descritos na Tabela 1.

Atividades

T4A1 Implemente um componente em VHDL correspondente ao fluxo de dados do PoliStack que respeite a entidade da Figura 2.

Trabalho 4, Atividade 1, 20 envios, maior nota, 4 pontos

O fluxo de dados é parametrizável, e os parâmetros definem os tamanhos das entradas e dos componentes internos. As saídas `memA_addr` e `memB_addr` são os endereços para a memória, respectivamente para a porta A e para a porta B. A saída `memB_wrd` é a saída com os dados a serem gravados na memória (no endereço `memB_addr`) e as entradas `memA_rdd` e `memB_rdd` são os dados lidos pela memória nas posições indicadas pelos endereços das respectivas portas.

Os sinais `pc_en`, `ir_en` e `sp_en`, quando em alto na borda de subida, habilitam a carga paralela nos registradores homônimos. Todos os sinais terminados `src` controlam os multiplexadores da Tabela 1.

Saída	Seletor	Descrição
PC	pc_src	Entrada do registrador PC.
	0	saída da ULA
	1	dados lidos da porta A da memória (memA_rdd)
memA_addr	mem_a_addr_src	Saída de endereços para a porta A da memória.
	0	saída do registrador SP
	1	saída do registrador PC
memB_addr	mem_b_addr_src	Saída de endereços para a porta B da memória.
	00	saída do registrador SP
	01	valor lido da memória na porta A (memA_rdd)
	1X	saída da ULA
memB_wrd	mem_b_wrd_src	Saída de dados para serem escritos na memória.
	00	saída dda ULA
	01	valor de saída do MUX memb_mem
	10	saída do registrador SP
	11	valor parcial do IR: signExt(ir[6:0])
memb_mem	mem_b_mem_src	Seletor intermediário para o memB_wrd (acima).
	0	valor lido da memória na porta A (memA_rdd)
	1	valor lido da memória na porta B (memB_rdd)
alu_a	alu_a_src	Valor de entrada A da ULA.
	00	saída do registrador PC
	01	saída do registrador SP
	1X	valor lido da memória na porta A (memA_rdd)
alu_b	alu_b_src	Valor de entrada A da ULA.
	00	valor de saída do MUX imm_shft
	01	valor de saída do MUX alu_mem
	10	valor parcial do IR (ir[4:0]«5)
	11	valor parcial do IR (not(ir[4])&ir[3:0]«2)
imm_shft	alu_shfimm_src	Seletor intermediário para o alu_b (acima).
	0	constante 1 (0x1=000...001)
	1	constante 4 (0x4=000...100)
alu_mem	alu_mem_src	Seletor intermediário para o alu_b (acima).
	0	valor da memória e IR (memA_rdd«7 | IR[6:0])
	1	valor lido da memória na porta B (memB_rdd)

Tabela 1: Multiplexadores do fluxo de dados

O sinal alu_op controla a operação da ULA e é ligado diretamente na instância dentro do fluxo de dados. A saída instruction é uma cópia da saída do IR, para que a unidade de controle saiba qual instrução está sendo executada.

Lembre-se que o fluxo de dados é combinatório, então você deve montá-lo estruturalmente usando os componentes que projetou nos trabalhos anteriores. Obviamente compoenentes sequenciais (e.g. registradores) podem ser incluídos no seu projeto, mas não deve existir nenhum componente sequencial novo.

T4A2 Implemente um componente em VHDL correspondente a unidade de controle do PoliStack que respeite a entidade da Figura 3.

Trabalho 4, Atividade 2, 20 envios, maior nota, 6 pontos

```
entity data_flow is
  generic(
    addr_s : natural := 16; -- address size in bits
    word_s : natural := 32  -- word size in bits
  );
  port (
    clock, reset: in bit;
    -- Memory Interface
    memA_addr, memB_addr : out bit_vector(addr_s-1 downto 0);
               memB_wrd  : out bit_vector(word_s-1 downto 0);
    memA_rdd,  memB_rdd  : in  bit_vector(word_s-1 downto 0);
    -- Control Unit Interface
    pc_en, ir_en, sp_en            : in  bit;
    pc_src, mem_a_addr_src,
    mem_b_mem_src                  : in  bit;
    mem_b_addr_src, mem_b_wrd_src,
    alu_a_src, alu_b_src           : in  bit_vector(1 downto 0);
    alu_shfimm_src, alu_mem_src    : in  bit;
    alu_op                         : in  bit_vector(2 downto 0);
    instruction                    : out bit_vector(7 downto 0)
  );
end entity;
```

Figura 2: Entidade VHDL para o fluxo de dados (T4A1).

Os sinais de `*en`, `*src` e o `alu_op` tem o mesmo significado semântico que no fluxo de dados, mas note que são saídas pois a unidade de controle é quem gera estes sinais para o fluxo de dados. De fato, a unidade de controle e o fluxo de dados são diretamente conectados para compor o processador. O sinal `instruction` é uma cópia do conteúdo do `IR` vinda do fluxo de dados.

Os sinais que ainda não foram descritos no fluxo de dados são relativos ao controle da memória. O sinal `mem_we`, quando alto, habilita a escrita na memória na próxima borada de subida se e somente se o sinal `mem_enable` estiver habilitado (em alto), caso contrário a memória não realizará nenhuma operação. Caso o `mem_we` estiver baixo mas o `mem_enable` alto, a memória fará somete as leituras nas portas correspondentes. O sinal `mem_busy` é o retorno da memória: enquanto estiver alto a memória está trabalhando e o resultado não é garantido.

Por último, o sinal `halted` é alto quando o processador encontra uma instrução `BREAK`, que efetivamente trava o processador para sempre. As demais instruções possíveis e seus efeitos sobre o fluxo de dados pode ser vista na Tabela 2.

Os pseudocódigos da tabela de instruções assumem que as ações acontecem em sequência, ou seja, um ponto e vírgula separando as ações fazem com que a segunda ação já veja os efeitos da primeira.

A instrução `IM` é especial. Note que há duas versões da mesma instrução com o mesmo *opcode*. Quando o processador encontra esta instrução, ele executa o `IM1`. Se logo em seguida houver uma segunda instrução `IM`, ele executará uma `IM*`. De fato, para qualquer quantidade de instruções `IM` em sequencia, o processador executará

```
entity control_unit is
  port (
    clock, reset: in bit;
    pc_en, ir_en, sp_en,
    pc_src, mem_a_addr_src, mem_b_mem_src, alu_shfimm_src, alu_mem_src,
    mem_we, mem_enable: out bit;
    mem_b_addr_src, mem_b_wrd_src, alu_a_src, alu_b_src: out bit_vector(1 downto 0);
    alu_op: out bit_vector(2 downto 0);
    mem_busy: in bit;
    instruction: in bit_vector(7 downto 0);
    halted: out bit
  );
end entity;
```

Figura 3: Entidade VHDL para a unidade de controle (T4A2).

`IM1` da primeira vez e `IM*` para as demais. Qualquer outra instrução quebrará este ciclo e a próxima instrução `IM` será tratada como o início de uma nova sequencia. Esta instrução é útil para construir imediatos por partes pois, como as instruções são de 8b e o imediato de 7b, são necessárias várias instruções `IM` para construir um imediato de 32b.

Se você entendeu este trabalho até aqui, já deve ter percebido que a unidade de controle na realidade é uma grande máquina de estados. Como dica, é muito útil ter os estados clássicos da arquitetura de computadores, como o *fetch*, *decode*, *execute* e *write*. O único estado obrigatório neste trabalho é o de *fetch*, onde a instrução apontada pelo `PC` deve ser buscada da memória e colocada em `IR`, e o `PC` incrementado. No entanto, sugerimos que trabalhe com o *decode*, que observa o `IR` e decide que rumo a máquina de estados tomará. Opcionalmente adote o *execute*, que é um ou mais estados que variam de acordo com a instrução sendo executada, e o *write*, onde caso aplicável a instrução escreve algo na memória.

Como no *decode* não há acesso à meória, aproveite para buscar o topo da pilha e o elemento seguinte pois eles são usados pela maioria das instruçõees.

Na implementação de referência, apenas a instrução STORE precisou de mais de um ciclo de execução.

Instruções para Entrega

Para este trabalho a biblioteca `ieee.numeric_bit` do pacote `ieee` é a única permitida para todas as atividades. O uso de `process` só está permitido na atividade A2, mas as funções podem ser usadas em todas as atividades. A violação destas restrições acarreta nota zero automaticamente, sem direito a revisão.

Restrições, preste atenção!

Para a atividade A2, você ainda pode considerar que os componentes do Trabalho 2 (`d_register`) e `alu`) estarão disponíveis.

Se você não tirou 10 no T2, o juiz usará o registrador e a ULA do professor.

Como boa prática de engenharia, faça seus *testbenches* e utilize o GHDL para validar suas soluções antes de postá-las no juiz.

Há um *link* específico no e-Disciplinas para cada atividade deste trabalho. Acesse-o somente quando estiver confortável para enviar sua solução. Você pode enviar apenas um único arquivo com sua

descrição VHDL em UTF-8 para cada atividade. O nome do arquivo não importa, mas sim a descrição que está dentro. As entidades devem ser como as especificadas ou o juiz te atribuirá nota zero.

Quando acessar o *link* no e-Disciplinas, o navegador abrirá uma janela para envio do arquivo. Selecione-o e envie para o juiz. Jamais recarregue a página de submissão pois seu navegador pode enviar o arquivo novamente, o que vai ser considerado pelo juiz como um novo envio e pode prejudicar sua nota final. Caso desista do envio, simplesmente feche a janela antes do envio.

Quando você clicar no *link* tem 1 minuto para enviar o arquivo ou fechar a janela, caso contrário uma submissão será contabilizada.

Depois do envio, a página carregará automaticamente o resultado do juiz, quando você poderá fechar a janela. Se não quiser esperar o resultado, feche a janela após o envio e verifique sua nota no e-Disciplinas posteriormente. A nota dada pelo juiz é somente para a submissão que acabou de fazer. Sua nota na atividade poderá ser vista no e-Disciplinas e pode diferir da nota dada pelo juiz dependendo da estratégia de atribuição de notas utilizada pelo professor que montou o problema.

Pode demorar alguns segundos até o juiz processar seu arquivo.

Atenção: não atualize a página de envio e não envie a partir de conexões instáveis (e.g. móveis) para evitar que seu arquivo chegue corrompido no juiz.

OPCode	Instrução	Descrição
0000_0000	BREAK	Levanta o `halt` e trava o processador.
0000_0010	PUSHSP	Empilha o conteúdo de `SP`. `mem[sp-4]=sp; sp=sp-4`
0000_0100	POPPC	Desempilha para o `PC`. `pc=mem[sp]; sp=sp+4`
0000_0101	ADD	Empilha a soma do topo com o segundo elemento da pilha. `mem[sp+4]=mem[sp+4]+mem[sp]; sp=sp+4`
0000_0110	AND	Empilha o AND do topo com o segundo elemento da pilha. `mem[sp+4]=mem[sp+4]&mem[sp]; sp=sp+4`
0000_0111	OR	Empilha o OR do topo com o segundo elemento da pilha. `mem[sp+4]=mem[sp+4]|mem[sp]; sp=sp+4`
0000_1000	LOAD	Substitui o topo da pilha pelo conteúdo endereçado pelo topo. `mem[sp]=mem[mem[sp]]`
0000_1001	NOT	Empilha o NOT do topo da pilha. `mem[sp]=not(mem[sp])`
0000_1010	FLIP	Empilha o reverso (MSB<->LSB) do topo da pilha. `mem[sp]=flip(mem[sp])`
0000_1011	NOP	Não faz nada por um ciclo de *clock*.
0000_1100	STORE	Guarda o segundo elemento da pilha no endereço apontado pelo topo. Desempilha ambos. `mem[mem[sp]]=mem[sp+4]; sp=sp+8`
0000_1101	POPSP	Desempilha para o `SP`. `sp=mem[sp]`
0001_nnnn	ADDSP	Soma o topo da pilha com o conteúdo no endereço calculado. `mem[sp]=mem[sp]+mem[sp+ir[3:0]«2]`
001_nnnnn	CALL	Empilha o `PC` e o sobrescreve com `ir[4:0]«5`n, causando um salto. `sp=sp-4; mem[sp]=pc; pc=ir[4:0]«5`
010_nnnnn	STORESP	Desempilha e guarda o valor desempilhado no endereço calculado. `mem[sp+(not(ir[4])&ir[3:0])«2]=mem[sp]; sp=sp+4`
011_nnnnn	LOADSP	Busca o valor no endereço calculado e empilha. `mem[sp-4]=mem[sp+(not(ir[4]&ir[3:0]))«2]; sp=sp-4`
1_nnnnnnn	IM1	Empilha um imediato vindo da instrução. `sp=sp-4; mem[sp]=signExt(ir[6:0])`
1_nnnnnnn	IM*	Desloca e adiciona um imediato no topo da pilha. `mem[sp]=memB[sp]«7 | ir[6:0]`

Tabela 2: Instruções do PoliStack