Computação Digital
Projeto Final

Guilherme Dantas
gdantas@aluno.puc-rio.br

3 de dezembro de 2020

1 Introdução

Este projeto final tem como objetivo implementar um processador em VHDL que atende às especificações dadas pelo professor do curso.

2 Metodologia

A arquitetura do processador se baseará no exemplo dado no slide do módulo 9, "Organização de processadores"(veja a figura 1). No entanto, antes de implementar as entidades em VHDL, é perspicaz primeiramente formalizar o comportamento do processador ao decodificar e executar cada instrução especificada. Assim, caso necessário, serão feitas modificações nessa arquitetura.

3 Análise das instruções

Serão listadas as instruções suportadas pelo processador e como serão interpretadas pelo processador em *regsiter transfer notation*.

A etapa de *fetch* é idêntica para todas as instruções, obtendo 1 palavra da memória e incrementando o *program counter*.

Toda operação aritmética e lógica entre os operadores `A` e `B` se dá na ALU.

```
fetch 1:
    A <- pc
    B <- 1
fetch 2:
    cir <- MEM[A]
    pc <- A + B
```

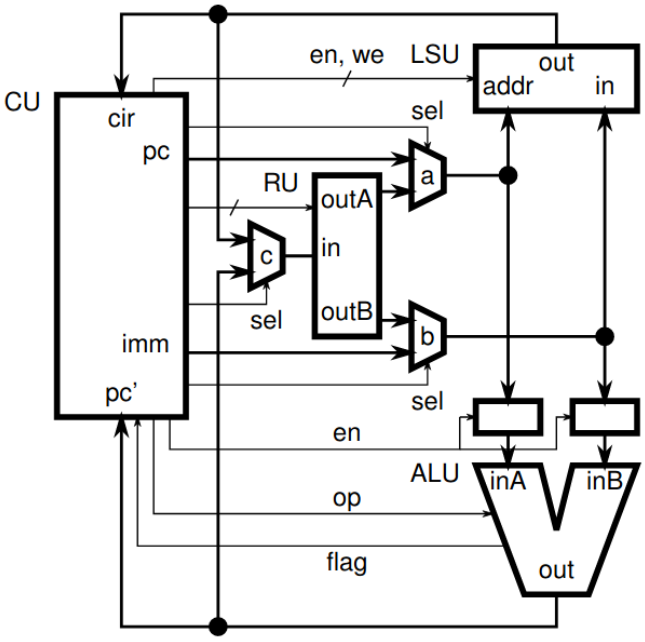

Figura 1: Arquitetura-base do processador, obtido dos slides "Organização de processadores"do módulo 9.

3.1 `ldi Rd, N`

```
decode:
    A <- pc
    B <- 1
execute:
    Rd <- MEM[A]
    pc <- A + B
```

3.2 `ldr Rd, [Rr]`

```
decode:
    A <- Rr
execute:
    Rd <- MEM[A]
```

3.3 `str Rd, [Rr]`

```
decode:
    A <- Rr
    B <- Rd
```

```
execute:
    MEM[A] <- B
```

3.4 push Rd

Precisamos guardar o *stack pointer* em um registrador. No entanto, se armazenássemos no banco de registros, teríamos os seguintes problemas:

- a instrução `pop` demoraria 2 ciclos de clocks a mais
- teríamos 5 registros no banco de registros (não é potência de 2)

Portanto, armazenamos o *stack pointer* num registrador a parte do banco de registros. É até interessante ter um *hardware* específico para esse ponteiro pois sobre ele são executadas apenas duas operações aritméticas - incremento e decremento - e armazená-lo no banco de registros seria desperdício.

Nota 1. O *stack pointer* (`sp`) é armazenado em um registrador a parte.

De modo a não perdermos ciclos preciosos de *clock*, incrementos e decrementos do *stack pointer* ocorrem localmente, não passando pela ALU.

```
decode:
    A <- sp
    B <- Rd
execute:
    MEM[A] <- B
    sp <- sp - 1
```

3.5 pop Rd

A operação de *pop* é simétrica à operação de *push*. O valor de `sp + 1` é calculado pela própria *control unit* e não pela ALU.

```
decode:
    A <- sp + 1
execute:
    Rd <- MEM[A]
    sp <- sp + 1
```

3.6 mov Rd, Rr

```
decode:
    A <- Rr
execute:
    Rd <- A
```

3.7 `inc Rd`

Não é necessário criar uma operação de incremento. Basta somar 1 (valor imediato) a `Rd`. Para operações lógico-aritmética como essa, o bit de *carry* gerado pelo incremento é armazenado no registrador `carry` dentro do processador.

Nota 2. A ALU sinaliza por uma *flag* se a operação gerou carry ou não.

Nota 3. As operações lógico-aritméticas geram *carry* e essa flag é armazenada no registro `carry` do *control unit* na etapa *execute*. No entanto, de modo a deixar o código mais limpo, isto será omitido do código RTN.

```
decode:
    A <- Rd
    B <- 1
execute:
    Rd <- A + B
```

3.8 `dec Rd`

```
decode:
    A <- Rd
    B <- 1
execute:
    Rd <- A - B
```

3.9 `incc Rd`

Aqui é necessário passar o valor do bit *carry* para a ALU como operando.

```
decode:
    A <- Rd
    B <- "0000000" # carry
execute:
    Rd <- A + B
```

3.10 `decb Rd`

Aqui é passado o valor do bit *borrow*, o inverso lógico do bit *carry*.

```
decode:
    A <- Rd
    B <- "0000000" # not carry
execute:
    Rd <- A - B
```

3.11 add Rd, Rr

```
decode:
    A <- Rd
    B <- Rr
execute:
    Rd <- A + B
```

3.12 sub Rd, Rr

```
decode:
    A <- Rd
    B <- Rr
execute:
    Rd <- A - B
```

3.13 and Rd, Rr

```
decode:
    A <- Rd
    B <- Rr
execute:
    Rd <- A & B
```

3.14 or Rd, Rr

```
decode:
    A <- Rd
    B <- Rr
execute:
    Rd <- A | B
```

3.15 xor Rd, Rr

```
decode:
    A <- Rd
    B <- Rr
execute:
    Rd <- A ^ B
```

3.16 lsl Rd

Aqui é importante perceber que não precisamos implementar operações para shifts lógicos e aritméticos. É fácil perceber que o que difere nessa operações é o bit que é concatenado no início ou no fim do resultado. Podemos passar esse bit como segundo operando para a ALU.

```
decode:
    A <- Rd
    B <- 0
execute:
    Rd <- A(7..0) # B(0)
```

3.17 `lsr Rd`

```
decode:
    A <- Rd
    B <- 0
execute:
    Rd <- B(0) # A(7..0)
```

3.18 `rol Rd`

Aqui passamos o bit *carry* (valor imediato) ao invés de zero.

```
decode:
    op <- OpLShift
    A <- Rd
    B <- "0000000" # carry
execute:
    Rd <- A(7..0) # B(0)
```

3.19 `ror Rd`

```
decode:
    op <- OpRShift
    A <- Rd
    B <- "0000000" # carry
execute:
    Rd <- B(0) # A(7..0)
```

3.20 `jmp Rd`

```
decode:
    A <- Rd
execute:
    pc <- A
```

3.21 `bz Rd`

Aqui checa-se se o resultado da última operação foi zero. Para isso, o *control unit* deve também armazenar em um registrador a *flag* indicada pela ALU.

Nota 4. A ALU sinaliza por uma *flag* se o resultado foi zero.

Nota 5. As operações lógico-aritméticas implicam no armazenamento da flag *zero* no registro `zero` do *control unit* na etapa *execute*. No entanto, de modo a deixar o código mais limpo, isto será omitido do código RTN.

```
decode:
    A <- Rd
execute:
    if zero then
        pc <- A
    end if
```

3.22 bnz Rd

```
decode:
    A <- Rd
execute:
    if not zero then
        pc <- A
    end if
```

3.23 bcc Rd

```
decode:
    A <- Rd
execute:
    if not carry then
        pc <- A
    end if
```

3.24 jmpi N

A função `resize` apenas ajusta o valor com sinal de 4 bits `N` para 8 bits, utilizando complemento a 2.

```
decode:
    A <- pc
    B <- resize(N, 8)
execute:
    pc <- A + B
```

3.25 bzi N

```
decode:
    A <- pc
    B <- resize(N, 8)
execute:
```

```
    if zero then
        pc <- A + B
    end if
```

3.26 bnzi N

```
decode:
    A <- pc
    B <- resize(N, 8)
execute:
    if not zero then
        pc <- A + B
    end if
```

3.27 bcci N

```
decode:
    A <- pc
    B <- resize(N, 8)
execute:
    if not carry then
        pc <- A + B
    end if
```

4 Unidades

4.1 RU - register unit

A unidade de registros contém 4 registros de 8 bits cada. As saídas A e B são selecionadas pelas entradas `SEL_A` e `SEL_B`, respectivamente. A operação de escrita em um registro é realizada da seguinte forma (veja a figura 2 como referência):

- Põe-se a palavra a ser escrita na entrada `C`
- Põe-se o endereço do registrador a ser escrito na entrada `SEL_C`
- Durante pelo menos um ciclo de *clock*, ligar a entrada `EN_C`

4.2 ALU - arithmetic logic unit

A unidade lógica e aritmética possui duas entradas para os operandos e uma saída para o resultado. A operação desejada é selecionada através da entrada `op` (veja a figura 3), que possui 8 valores possíveis, listados todos a seguir. As flags `carry` e `zero` são sinalizadas nas respectivas portas de saídas.

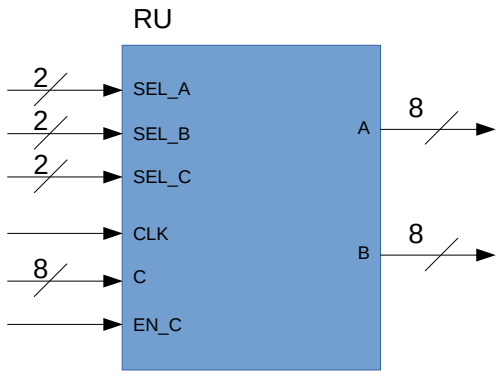

Figura 2: Diagrama da unidade de registros

0. `OpSel`: $C \leftarrow A, carry \leftarrow 0$
1. `OpAdd`: $D \leftarrow A + B, C \leftarrow D(7..0), carry \leftarrow D\ lsr\ 8$
2. `OpSub`: $D \leftarrow A + \overline{B} + 1, C \leftarrow D(7..0), carry \leftarrow D\ lsr\ 8$
3. `OpAnd`: $C \leftarrow A\ \&\ B, carry \leftarrow 0$
4. `OpOr`: $C \leftarrow A \mid B, carry \leftarrow 0$
5. `OpXor`: $C \leftarrow A \wedge B, carry \leftarrow 0$
6. `OpLShift`: $C \leftarrow A(6..0)\ \#\ B(0), carry \leftarrow A(7)$
7. `OpRShift`: $C \leftarrow B(0)\ \#\ A(6..0), carry \leftarrow A(7)$

O registro interno `D` possui 9 bits de modo a acomodar o bit de *carry*. A *flag* `zero` é ligada se e somente se todos os bits de `C` forem zero.

4.3 LSU - load store unit

A unidade de *load/store* possui uma memória interna de 256 palavras de 8 bits cada. Os endereços (entrada `addr`) têm, portanto, 8 bits de largura. A unidade opera em um dos dois modos, dependendo do valor da entrada *write enable* (`we`): leitura (`we = '0'`) ou escrita (`we = '1'`). No modo de leitura, a saída `dataout` sinaliza a palavra no endereço apontado. Enquanto que no modo de escrita, o dado na entrada `datain` é escrito no endereço apontado.

Além disso, os dispositivos de entrada e saída estão mapeados na memória. De forma a se comunicar com eles, o componente possui também a entrada `devicein` para escrever num endereço fixo um byte e a saída `deviceout` para ler de um endereço fixo um byte. Essas portas tem seus valores atualizados a cada ciclo de *clock*. Veja a figura 4.

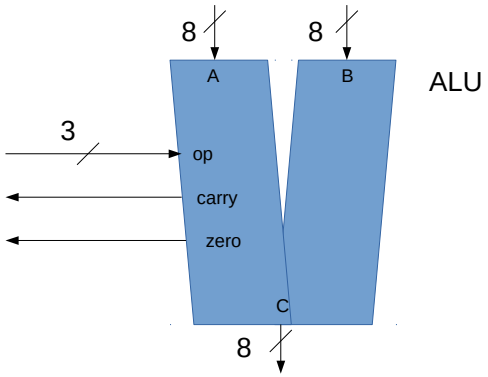

Figura 3: Diagrama da unidade lógica e aritmética

4.4 CU - control unit

A unidade de controle é a mais complexa pois armazena a lógica da máquina de estados, tem o maior número de portas e se conecta com o maior número de unidades. A máquina de estados possui os seguintes estados.

1. Fetch [1 e 2] - busca a instrução atual da memória e aponta para a seguinte
2. Decode - decodifica a instrução, configurando o caminho de dados
3. Execute - executa a instrução, efetivando resultados em registradores

Um diagrama simplificado da máquina de estados do *control unit* está representada na figura 5. Perceba que as transições de saída dos estados decode e execute dependem da instrução lida da memória. É importante lembrar que todas as transições foram detalhadas na seção anterior.

O diagrama dessa unidade está ilustrado na figura 6. Para torná-lo mais claro, os nomes das portas possuem como prefixo o componente de origem ou de destino, exceto o *clock*.

4.5 Outros componentes

A arquitetura estrutural do processador conta também com outros componentes periféricos mais simples para o seu funcionamento, listados a seguir.

- Multiplexadores 2-1
- Detector de borda positiva
- Contador (incrementa e decrementa)
- *Driver* de *display* de 7 segmentos

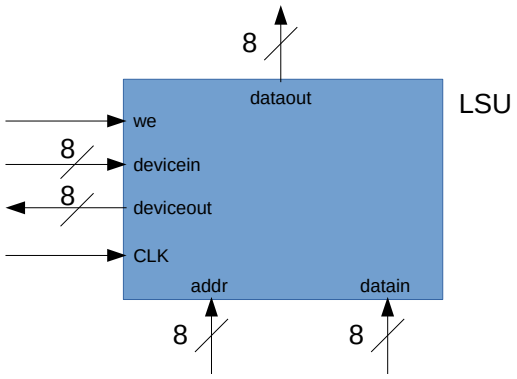

Figura 4: Diagrama da unidade *load/store*

5 Controle do caminho de dados

É preciso agora converter o código em RTN para sinais de controles dados pela *control unit*. Vale lembrar que é preciso colocar na saída `alu_op` na etapa de decode o código da instrução a ser executada pela ALU.

```
A <- var (valor de registrador interno no barramento A):
    a_mux_sel <- '0' (cu)
    a_mux_in <- var

A <- Rx (valor do registro x no barramento A):
    a_mux_sel <- '1' (ru)
    ru_sel_a <- x

B <- var (valor de registrador interno no barramento B):
    b_mux_sel <- '0' (cu)
    b_mux_in <- var

B <- Rx (valor do registro x no barramento B):
    b_mux_sel <- '1' (ru)
    ru_sel_b <- x

Rx <- MEM[A] (valor da memória no registro x):
    c_mux_sel <- '0' (lsu)
    ru_c_en <- '1'
    ru_sel_c <- x
    lsu_we <- '0'
```

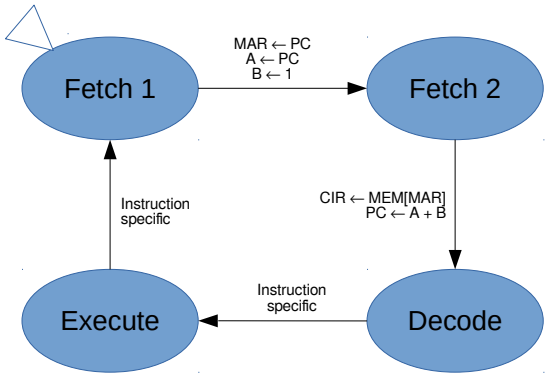

Figura 5: Máquina de estados da unidade de controle

```
Rx <- alu (resultado de operação no registro x):
    c_mux_sel <- '1' (alu)
    ru_c_en <- '1'
    ru_sel_c <- x

MEM[A] <- Rx (valor do registro x na memória):
    b_mux_sel <- '1' (ru)
    ru_sel_b <- x
    lsu_we <- '1'

var <- MEM[A] (valor da memória num registrador interno):
    lsu_we <- '0'
    var <- lsu_out

MEM[A] <- B (escrita à memória):
    lsu_we <- '1'

sp <- sp + 1 (incremento do stack pointer):
    sp_inc <- '1'

sp <- sp - 1 (decremento do stack pointer):
    sp_dec <- '1'
```

6 Testes

De forma a testar o processador, foram rodados dois programas compilados pelo *assembler* em Python.

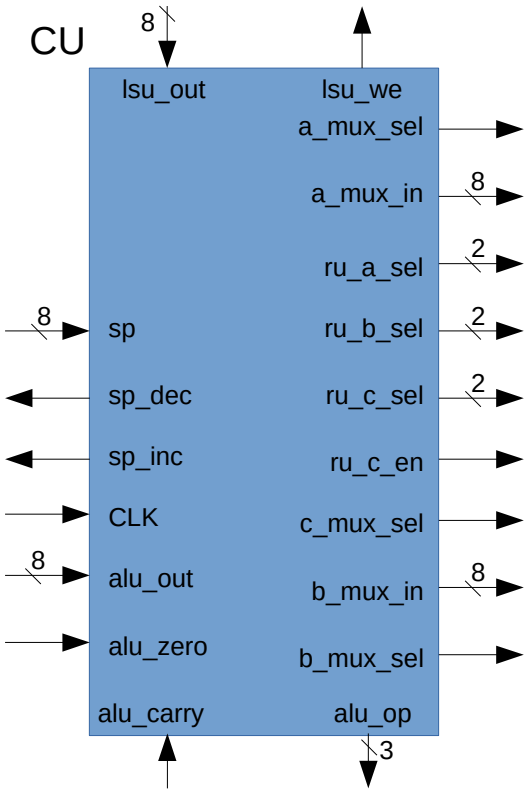

Figura 6: Diagrama da unidade de controle

6.1 Contador de bordas positivas

O código em *assembly* está nos slides 14-15 do enunciado do projeto. Para testá-lo, foi criado um *testbench* em que a entrada `BTN_EAST` era pressionada a cada 500 ciclos de *clock*, para que houvesse tempo de o programa detectar a borda positiva desse sinal e fosse possível observar o *delay*. Ao simular o circuito, pode-se observar o resultado esperado, ilustrado na figura 7.

6.2 Gerador da sequência de Fibonacci

O código em *assembly* está anexado junto com este relatório com o nome `fib.asm`. Usando o mesmo código de *testbench* do teste anterior, e com o programa carregado na memória, simulamos novamente o circuito. O resultado da simulação está ilustrado na figura 8.

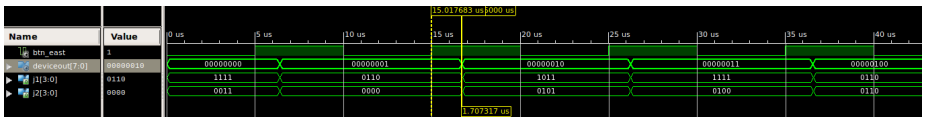

Figura 7: Simulação do detector de borda positiva. O sinal `deviceout` indica o valor no endereço de memória `0x0e` e `j1` e `j2` são os sinais enviados ao *display* de 7 segmentos. Há um delay de $1.7\mu s$ entre a borda positiva do sinal e o incremento do valor no *display* com o *clock* oscilando a 50 MHz

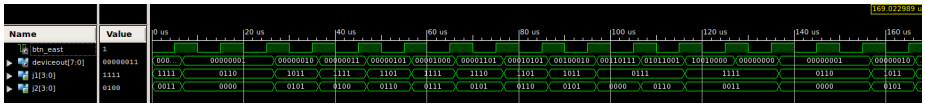

Figura 8: Simulação do gerador de sequência de Fibonacci. É possível perceber a sequência truncada sendo gerada no sinal `deviceout` (em binário): 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 0, 1, 1, 2, ...