Computação Digital

Introdução ao projeto

Wouter Caarls
wouter@ele.puc-rio.br

Período 2024.2: 12 Agosto 2024 - 16 Dezembro 2024

Projeto

O projeto final da disciplina é o desenho de um processador. As especificações são

- Arquitetura *Von Neumann*;
- Arquitetura *load/store*;
- Instruções e palavras de 16 bits;
- Memória de 8192 palavras;
- 16 registradores de 16 bits;
- PC (r15) e SP (r14) no banco de registradores;
- Pilha;
- E/S mapeada na memória.

Entrega

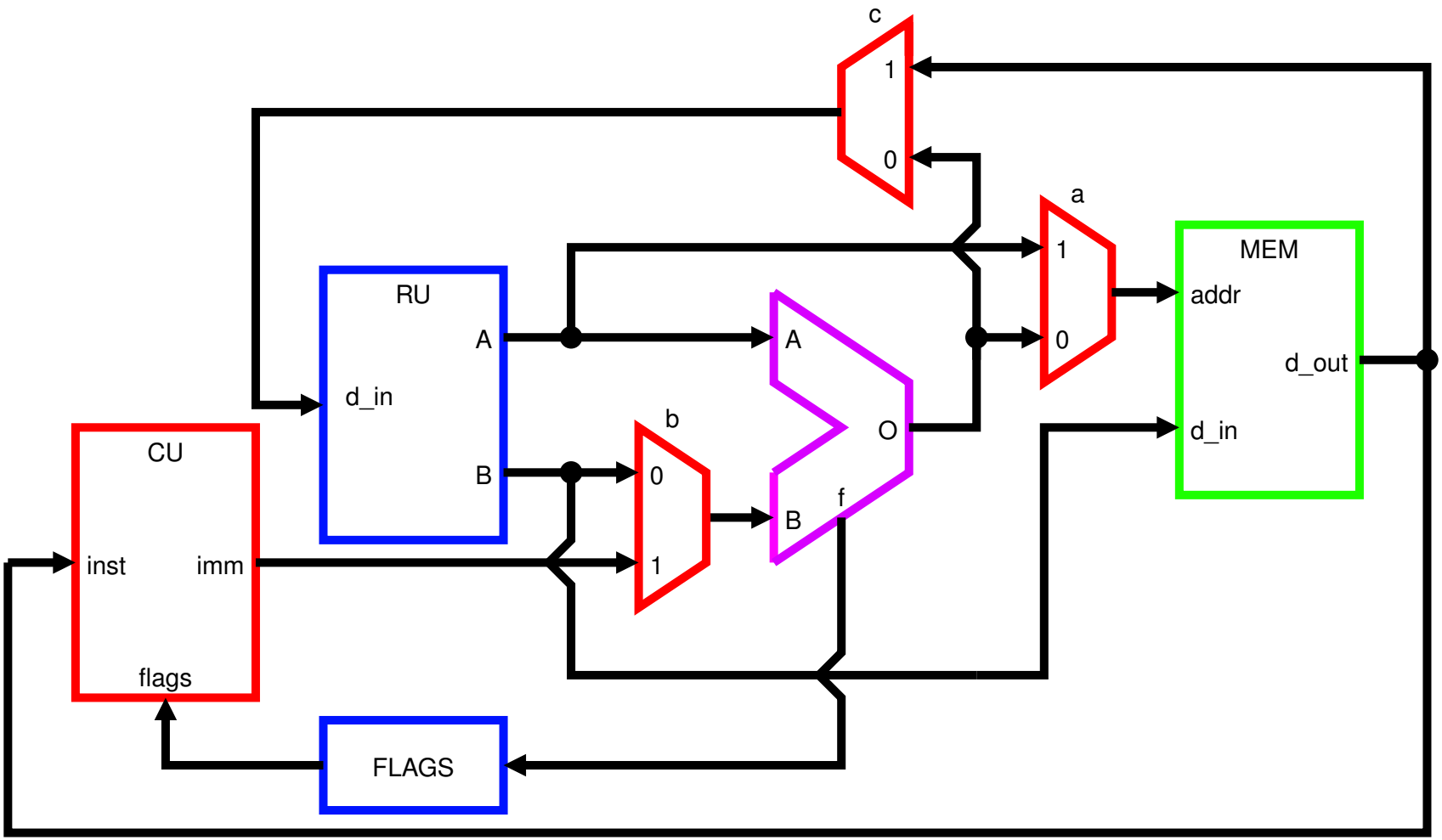
A data de entrega do projeto é 9-12-2024 às 09:00. Precisa entregar

- Todo o projeto num arquivo .zip, pronto para simular e sintetizar;
- Um relatório sobre o projeto em arquivo .pdf, contendo:
  - Desenho do caminho de dados;
  - Descrições das unidades, incluindo a máquina de estados da unidade de controle;
  - Programas em *assembly* e C que escreveu e rodou;
  - Resultados produzidos pelos programas;
  - Fotos da placa real rodando os programas;
  - Observações feitas ao longo da implementação.

O tamanho do relatório deve ser por volta de 10 páginas.

O projeto pode ser feito em duplas.

Caminho de dados



Conjunto de instruções

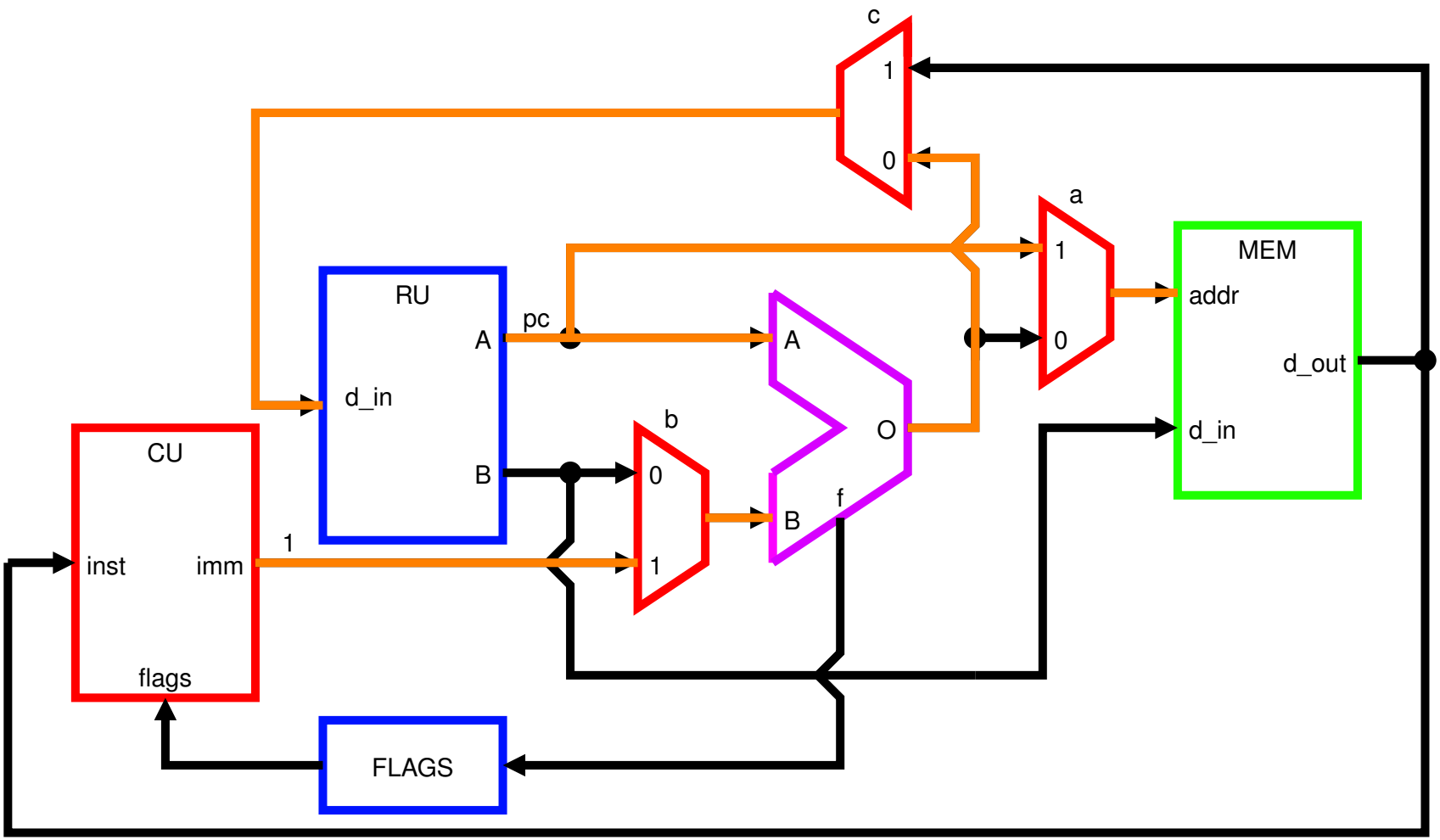
O conjunto de instruções tem 4 grupos:

- Valores imediatos e saltos;
- Acesso de memória;
- Aritmética;
- Lógica.

Só as instruções aritméticas e lógicas devem mudar o estado dos *flags*. O processador tem quatro: o *carry flag* (a operação gerou *carry*), o *zero flag* (o resultado da operação foi 0), o *negative flag* (o resultado da operação com sinal é negativo) e o *overflow flag* (o resultado da operação com sinal sobreescreveu o sinal).

Fetch (1)

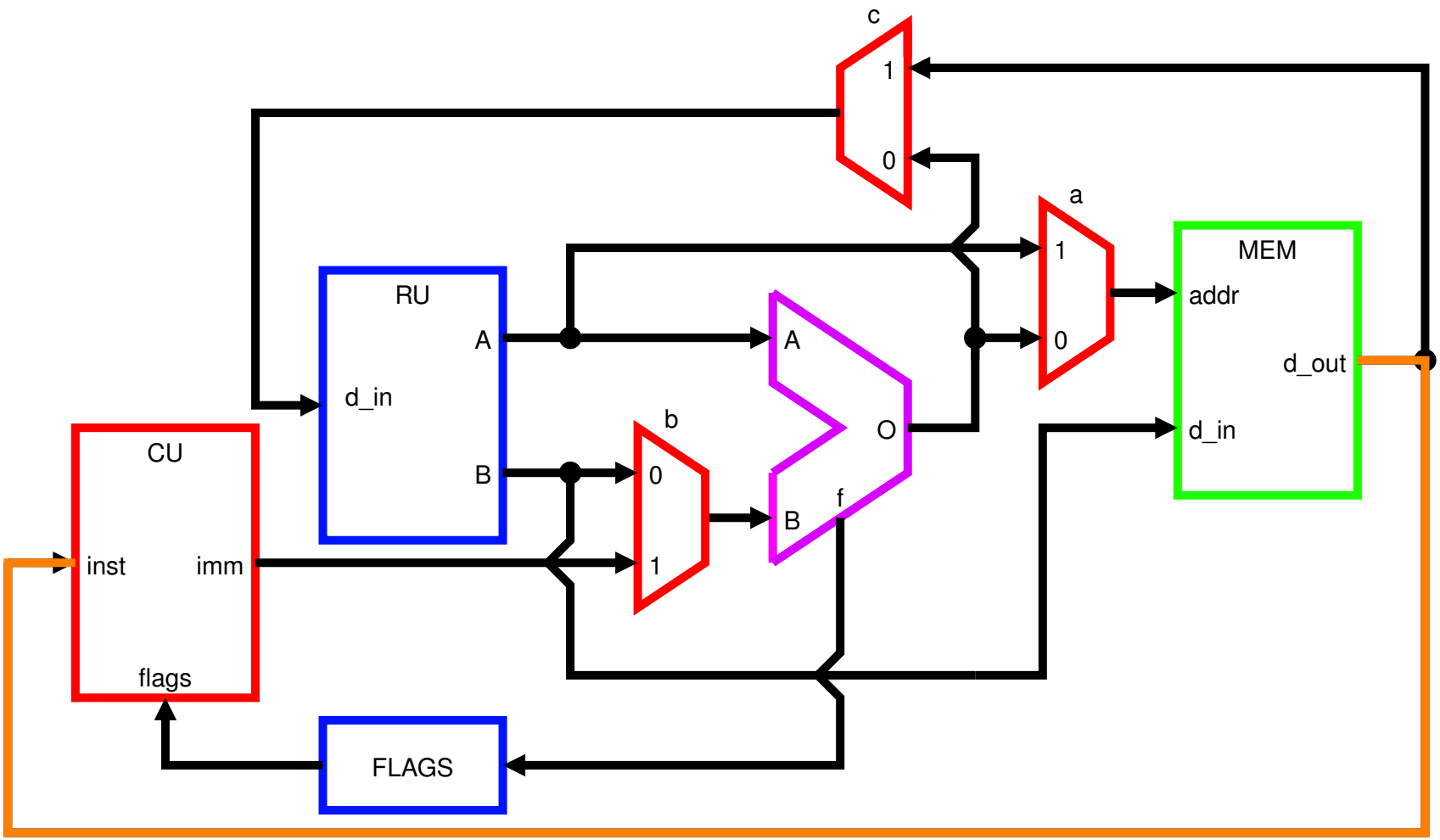
Fetch (1)



$$MAR \leftarrow PC, PC \leftarrow PC + 1$$

Fetch (2)

Fetch (2)



$CIR \leftarrow MEM[MAR]$

Valores imediatos

```
mov  rd, c8u ; rd ← c8u
movt rd, c8u ; rd ← (rd&255)|(c8u<<8)
```

Instrução	Op	N1	N2	N3
mov	0000	rd	c8u	...
movt	0001	rd	c8u	...

```
mov  r9, 254 ; 0000 1001 11111110
movt r5, 4   ; 0001 0101 00000100
```

Saltos

```
b<cond> c8i  ; if cond then pc ← pc + c8i + 1 else pc ← pc +
    1
jmp     c12u ; pc ← c12u
```

Instrução	Op	N1	N2	N3
b<cond>	0010	cond	c8i	...
jmp	0011	c12u	...	...

Condição	Codificação	Descrição
	0000	Não condicional
z	0001	zero = True
nz	0010	zero = False
cs	0011	carry = True
cc	0100	carry = False
lt	0101	overflow != negative
ge	0110	overflow = negative

```
bcc -5   ; 0010 0100 11101000
jmp 1948 ; 0011 011110011100
```

Acesso de memória

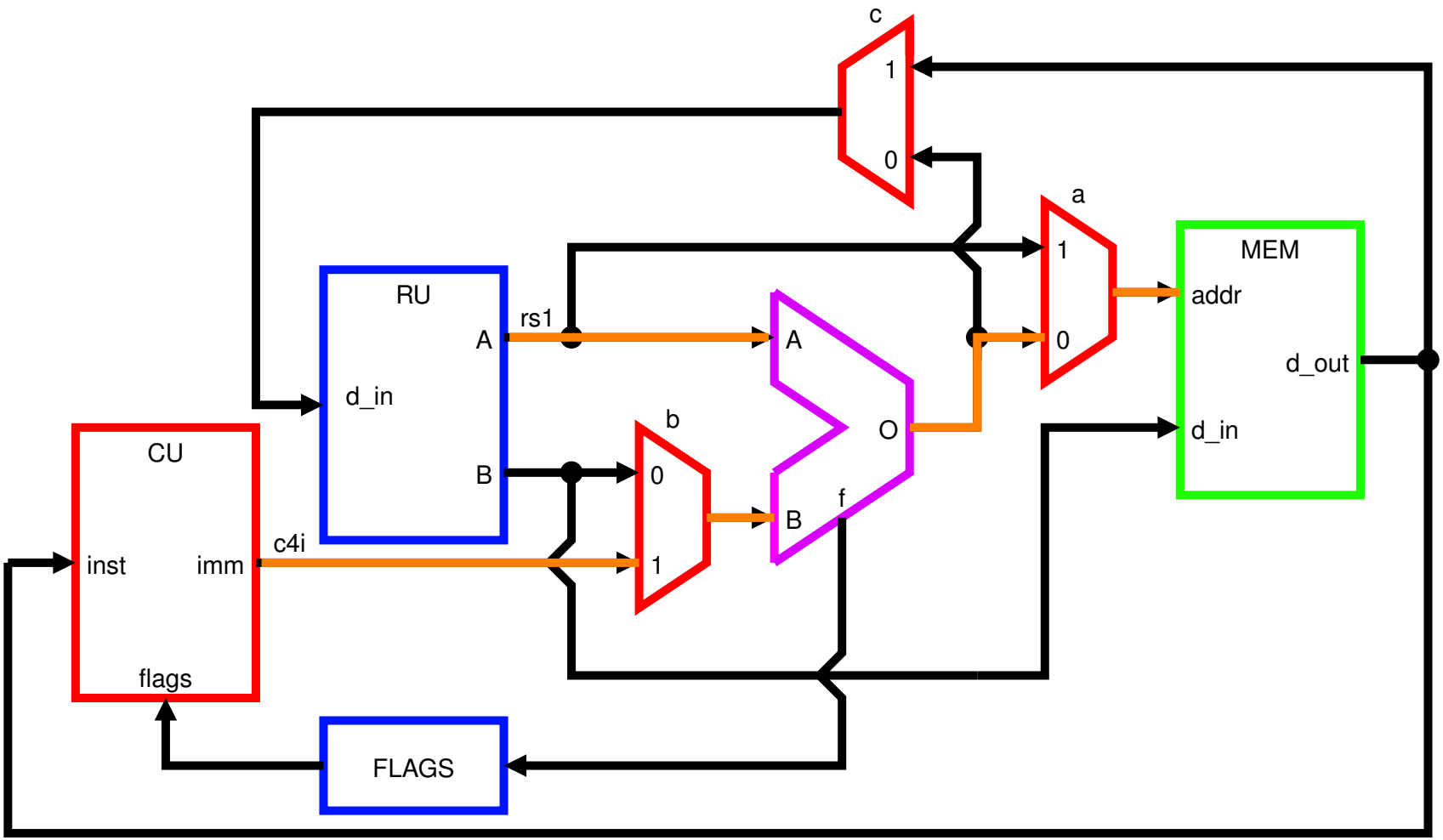
```
ldr  rd,  [rs,  c4i] ; rd ← MEM[rs + c4i],   pc ← pc + 1
str  rs1, [rs2, c4i] ; MEM[rs2 + c4i] ← rs1, pc ← pc + 1
push rs ; [sp] ← rs,     sp ← sp - 1, pc ← pc + 1
pop  rd ; rd ← [sp + 1], sp ← sp + 1, pc ← pc + 1
```

Instrução	Op	N1	N2	N3
ldr	0100	rd	rs	c4i
str	0101	rs1	rs2	c4i
push	0110	0000	1110	rs
pop	0111	rd	1110	0000

```
ldr r0, [r3, 5] ; 0100 0000 0011 0101
pop r6          ; 0111 0110 1110 0000
```

LDR rd, [rs, c4i]

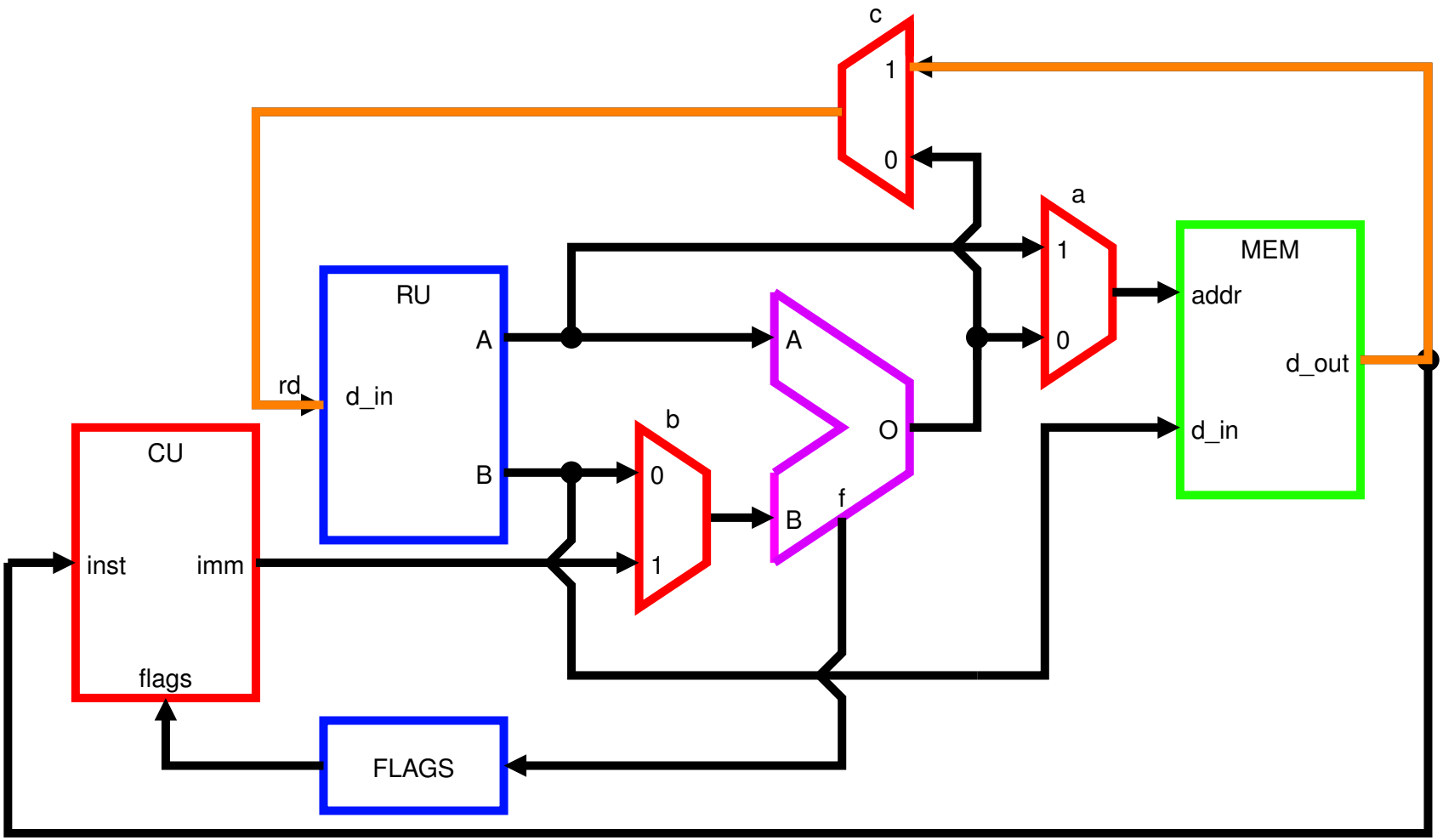
Execute



$$MAR \leftarrow rs + c4i$$

LDR rd, [rs, c4i]

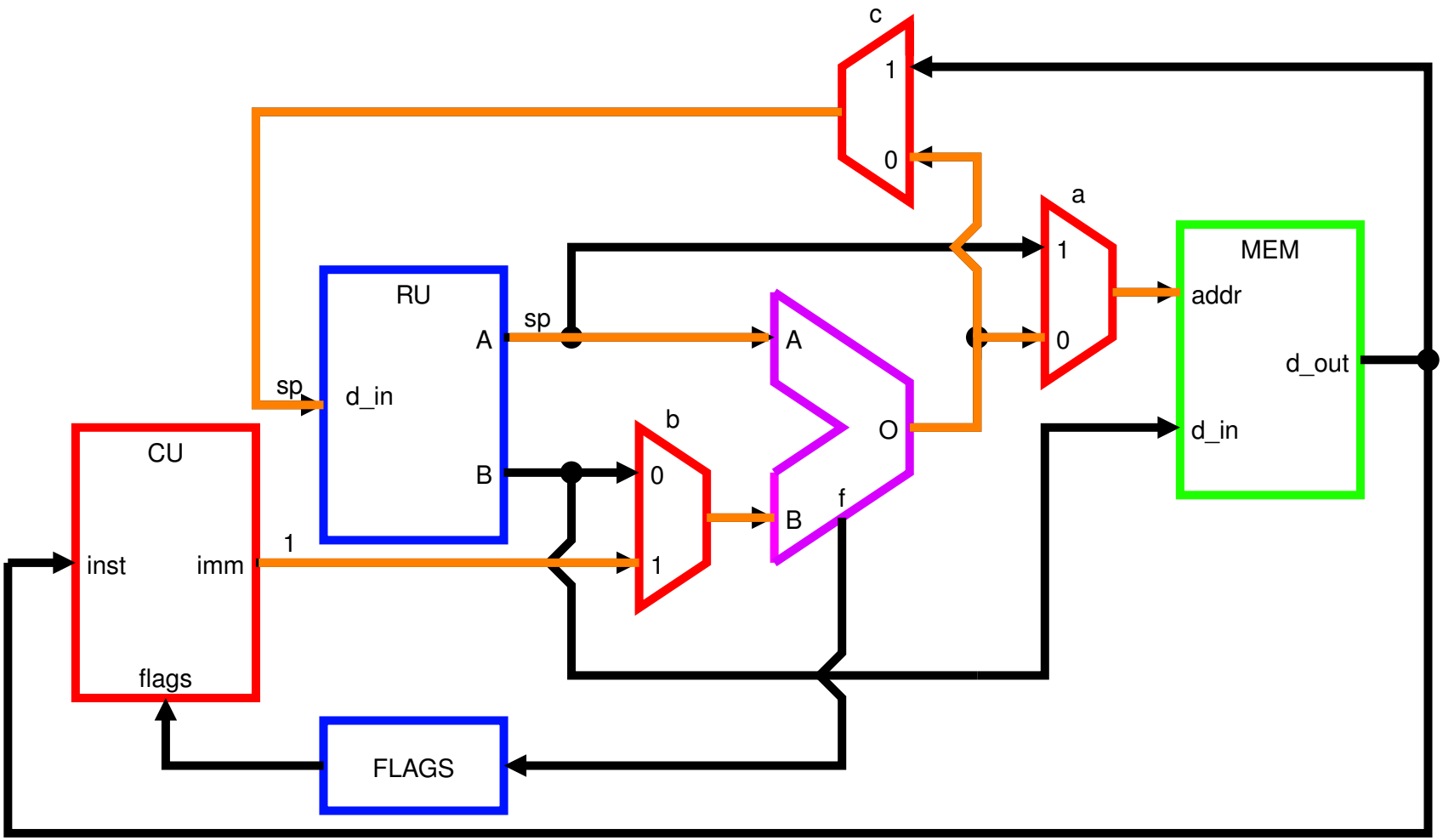
Write back



$$rd \leftarrow MEM[MAR]$$

POP rd

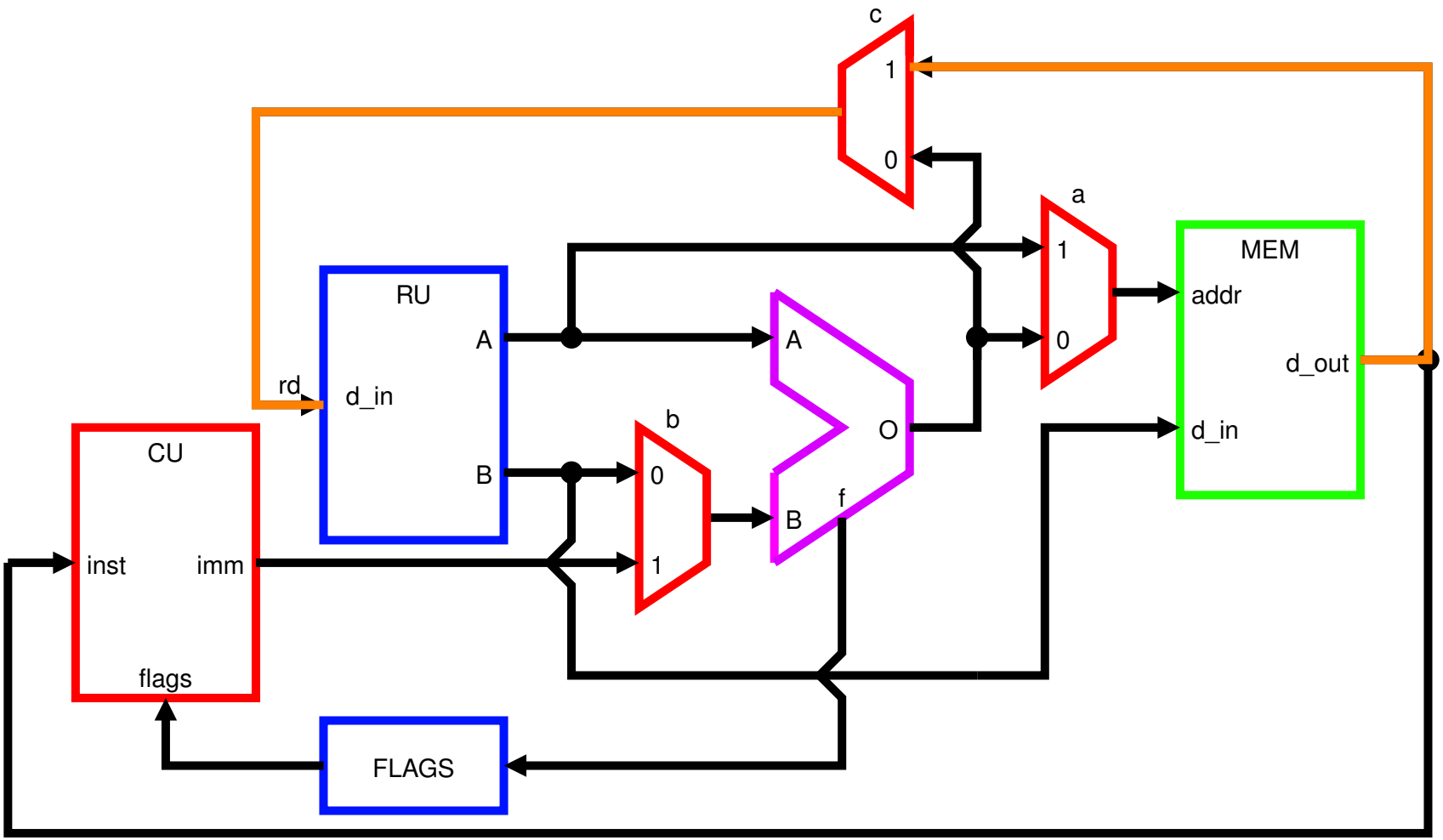
Execute



$$MAR \leftarrow sp + 1, sp \leftarrow sp + 1$$

POP rd

Write back



$rd \leftarrow MEM[MAR]$

Aritmética

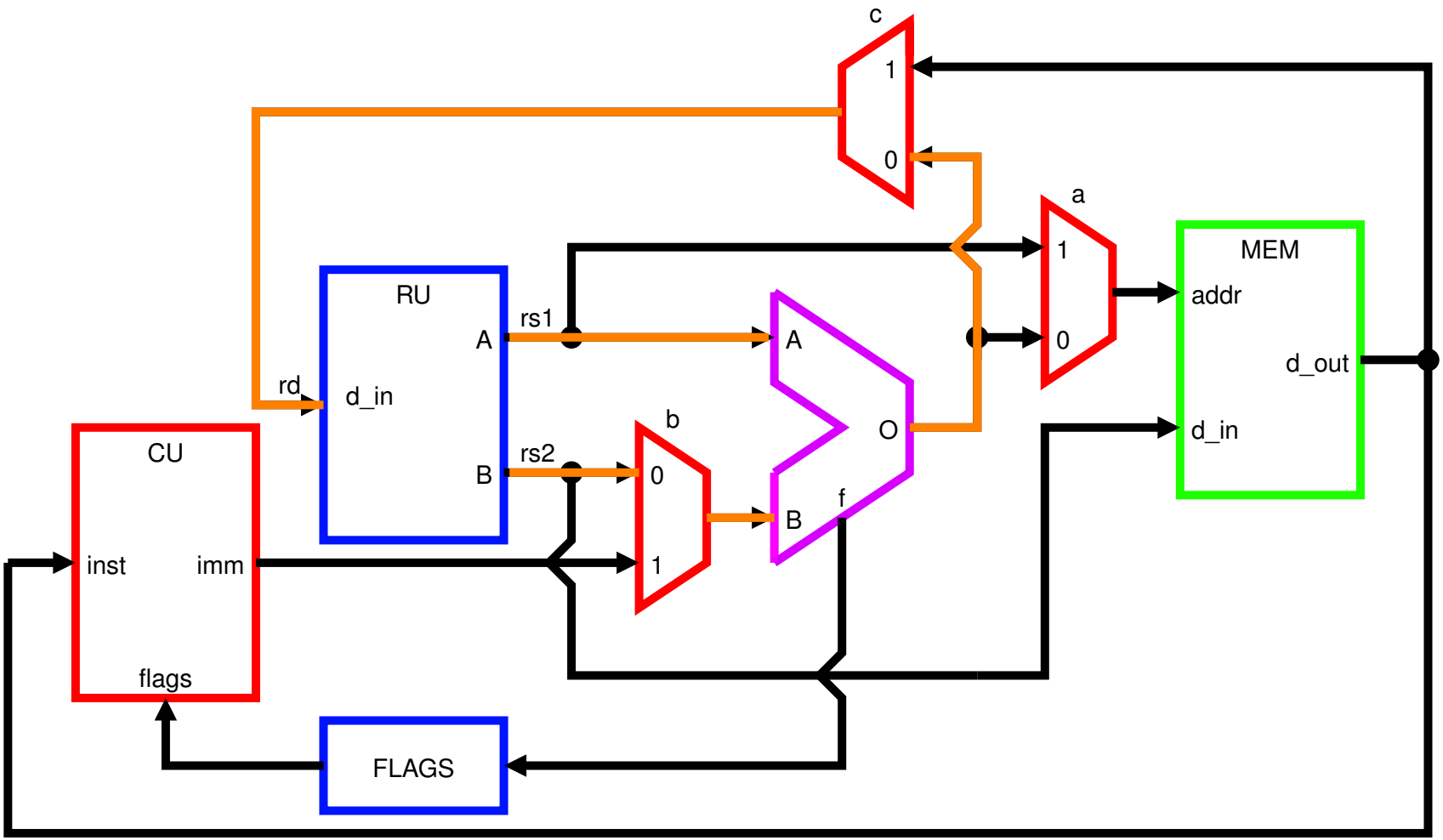
```
add  rd, rs1, rs2 ; rd ← rs1 + rs2,  pc ← pc + 1
add  rd, rs,  c4u ; rd ← rs  + c4u,  pc ← pc + 1
sub  rd, rs1, rs2 ; rd ← rs1 - rs2,  pc ← pc + 1
sub  rd, rs,  c4u ; rd ← rs  - c4u,  pc ← pc + 1
```

Instrução	Op	N1	N2	N3
add	1000	rd	rs1	rs2
add	1001	rd	rs	c4u
sub	1010	rd	rs1	rs2
sub	1011	rd	rs	c4u

```
add  r9, r10, r14 ; 1000 1001 1010 1110
sub  r9, r10, 14  ; 1011 1001 1010 1110
```

ADD rd, rs1, rs2

Execute



$$rd \leftarrow rs1 + rs2$$

Lógica

```
shft rd, rs , 1   ; rd ← rs1 << 1 (d = 0)
shft rd, rs , -1  ; rd ← rs1 >> 1 (d = 1)
and  rd, rs1, rs2 ; rd ← rs1 & rs2
or   rd, rs1, rs2 ; rd ← rs1 | rs2
xor  rd, rs1, rs2 ; rd ← rs1 ^ rs2
```

Instrução	Op	N1	N2	N3
shft	1100	rd	rs	d000
and	1101	rd	rs1	rs2
or	1110	rd	rs1	rs2
xor	1111	rd	rs1	rs2

```
shft r5, r3, -1  ; 1100 0101 0011 1000
xor  r2, r8, r12 ; 1111 0010 1000 1100
```

Organização da memória

O processador começa a executar a instrução no endereço `0x0010` da memória. A memória começa com as entradas/saídas mapeadas. A pilha começa em `0x1FFF`.

Endereço	Nome	Uso
`0x0000`	`btn`	Estado dos botões
`0x0001`	`enc`	Contagem do *encoder*
`0x0002`	`kdr`	Último caractere lido do teclado PS/2
`0x0003`	`udr`	Dados da parta serial
`0x0004`	`usr`	Estado da porta serial
`0x0005`	`led`	Valor mostrado nos LEDs
`0x0006`	`ssd`	Valor mostrado no *display* de 7 segmentos
`0x0007`	`ldr`	Caractere a ser mandado para o LCD
`0x0008`	`lcr`	Comando a ser mandado para o LCD

O registrador do teclado volta para 0 ao ler o valor. Os registradores do LCD voltam para 0 ao terminar de mandar.

Assembler

A infraestrutura de *software* do projeto fica em um pacote de Python chamado `puc16`[1]. Pode instalar usando Thonny ou `pip`. Contém um *assembler*, que pode usar para converter um programa em *assembly* para código binário no formato de um *array* de VHDL:

```
$ cat simple.asm
main: mov r0, 0
loop: add r0, r0, 1
      b   @loop

$ as-puc16 simple.asm
signal ram: ram_t := (
16 => "0000000000000000", --simple.asm:1: main: mov r0, 0
17 => "1001000000000001", --simple.asm:2: loop: add r0, r0, 1
18 => "0010000011111110", --simple.asm:3:       b @loop
others => (others => '0'));
```

[1] https://pypi.org/project/puc16/

Preprocessador

O *assembler* vem com preprocessador que reconhece os comandos

.include inclui o conteúdo de um outro arquivo:

```
.include "def.asm"
```

.macro define um *macro* que pode ser chamado no resto do código:

```
        .macro waitkb
_wait:  ldr   $0, [@kdr] ; Read keyboard character
        mov   $0, $0     ; Set flags
        bz    @_wait     ; Wait until nonzero
        .endmacro
```

Na hora de chamar usando

```
waitkb r0
```

o argumento `$0` é substituido por `r0`. Rótulos começando em _ são individualizados.

Exemplo

```
        .include "def.asm"
        .include "macros.asm"

; Write welcome message
main:   mov  r0, 7
        mov  r1, low(@msg1)
        movt r1, high(@msg1)
        call @writemsg ; Write 7 characters from @msg1 to lcd
halt:   b    @halt
; Write message to LCD (r0 = length, r1 = address)
writemsg:
        add  r0, r0, r1; Set r0 to one past final character
wloop:  ldr  r2, [r1]  ; Get character to write
        writelcd r2    ; Write character to LCD
        add  r1, r1, 1 ; Advance
        sub  r2, r1, r0;
        bnz  @wloop    ; Loop until last character was sent
        ret            ; Return from subroutine

.section data
msg1:   .dw "welcome"
```

Compilador

O pacote `puc16` também contém um compilador para C. Porém, é limitado a valores inteiros de 16 bits. O compilador pode emitir *assembly*, ou VHDL diretamente.

```
$ cat hello.c
#include "puc16.h"

unsigned char buf[] = "Hello, world!";

void main(void)
{
  for (int ii=0; buf[ii]; ++ii)
  {
    while (inp(LDR));
    outp(buf[ii], LDR);
  }
}
```

Exemplo (startup)

```
$ cc-puc16 hello.c
signal ram: ram_t := (
16 => "1001110011110010", -- 15:          add r12, r15, 2
17 => "0110000011101100", -- 16:          push r12
18 => "0011000000010100", -- 17:          jmp @main
19 => "0010000011111111", -- 18: loop:    b @loop
```

Exemplo (main)

```
20 => "0000000000000000", -- 37: main_0: mov r0, 0
21 => "0000000100000000", -- 38: main_2: mov r1, low(@buf)
22 => "0001000100010000", -- 39:         movt r1, high(@buf)
23 => "1000000100010000", -- 40:         add r1, r1, r0
24 => "0100000100010000", -- 41:         ldr r1, [r1, 0]
25 => "1001000100010000", -- 42:         add r1, r1, 0
26 => "0010000100000001", -- 43:         bz @main_4
27 => "0011000000011101", -- 44:         jmp @main_6
28 => "0011000000101010", -- 45: main_4: jmp @main_epilog
29 => "0000000100000111", -- 46: main_6: mov r1, 7
30 => "0100000100010000", -- 47:         ldr r1, [r1, 0]
31 => "1001000100010000", -- 48:         add r1, r1, 0
32 => "0010000100000001", -- 49:         bz @main_8
33 => "0011000000011101", -- 50:         jmp @main_6
34 => "0000000100000000", -- 51: main_8: mov r1, low(@buf)
35 => "0001000100010000", -- 52:         movt r1, high(@buf)
36 => "1000000100010000", -- 53:         add r1, r1, r0
37 => "0100000100010000", -- 54:         ldr r1, [r1, 0]
38 => "0000001000000111", -- 55:         mov r2, 7
39 => "0101000100100000", -- 56:         str r1, [r2, 0]
40 => "1001000000000001", -- 57:         add r0, r0, 1
41 => "0011000000010101", -- 58:         jmp @main_2
42 => "0111111111100000", -- 59: main_epilog: pop r15
```

Exemplo (data)

```
4096 => "0000000001001000", -- 21: buf:          .dw 72
4097 => "0000000001100101", -- 22:               .dw 101
4098 => "0000000001101100", -- 23:               .dw 108
4099 => "0000000001101100", -- 24:               .dw 108
4100 => "0000000001101111", -- 25:               .dw 111
4101 => "0000000000101100", -- 26:               .dw 44
4102 => "0000000000100000", -- 27:               .dw 32
4103 => "0000000001110111", -- 28:               .dw 119
4104 => "0000000001101111", -- 29:               .dw 111
4105 => "0000000001110010", -- 30:               .dw 114
4106 => "0000000001101100", -- 31:               .dw 108
4107 => "0000000001100100", -- 32:               .dw 100
4108 => "0000000000100001", -- 33:               .dw 33
4109 => "0000000000000000", -- 34:               .dw 0
others => (others => '0'));
```

Projeto

No projeto, tem que implementar

- Um processador capaz de executar o conjunto de instruções definido nesse documento.
- Periféricos para
  - Display de 7 segmentos
  - Teclado PS/2
  - LCD

Para testar, precisa

- Verificar o funcionamento usando `unittest.asm`[2]
- Escrever uma calculadora de valores com 1 dígito, usando o teclado e o LCD. Ao digitar `1=2<Enter>` Deve aparecer `1+2=3` (não precisa tratar o SHIFT do teclado, então = é interpretado como +). A calculadora deve tratar só adição e subtração.

[2] `https://github.com/wcaarls/puc16/blob/master/examples/asm/unittest.asm`

Dicas

- A RAM é síncrona, com tamanho de 8192 palavras de 16 bits:

```
process(clk) is
begin
  if rising_edge(clk) then
    if we = '1' then
      ram(to_integer(unsigned(addr))) <= d_in;
    end if;
    d_out <= ram(to_integer(unsigned(addr)));
  end if;
end process;
```

- Ao invés de implementar um decodificador de endereços separadamente, faz dentro da RAM.

Dicas (ctd)

- Na CU, define valores padrões para os sinais de saída, e sobreescreve quando necessário. Evita definir o valor de todos os sinas para todas as instruções.
- Use *aliases* para partes do CIR

```
alias gr:   crumb  is cir(15 downto 14);
alias op:   crumb  is cir(13 downto 12);
alias grop: nibble is cir(15 downto 12);
alias r1:   nibble is cir(11 downto 8);
alias r2:   nibble is cir(7  downto 4);
alias r3:   nibble is cir(3  downto 0);
alias f:    nibble is cir(11 downto 8);
alias c4:   nibble is cir(3  downto 0);
alias c8:   byte   is cir(7  downto 0);
alias c12:  std_logic_vector(11 downto 0) is
    cir(11 downto 0);
```

- Implementa as funções de transição e de saída da CU em lógica combinacional. A CU tem só 2 registradores: `state` e `cir`.

Dicas (ctd)

- Para converter um valor com sinal de 4 bits em 16 bits, precisa repetir o bit 3 para os novos bits mais significativos (`3210` -> `3333333333333210`). Para valores sem sinal basta concatenar `"0000000000000"`.
- Use as definições dadas em `libcpu.vhd` ao invés de vetores diretos (`bBNZ` ao invés de `"0010"`).
- Implemente as operações da ALU definidas em `libcpu.vhd`.
- Overflow =

```
    add  not (A(15) xor B(15)) and (A(15) xor Z(15))
    sub      (A(15) xor B(15)) and (A(15) xor Z(15))
others  '0'
```