2ºTeste AC (30/jan/2018)

[1]

Instrução	Descrição	Codificação											
		b11	b10	b9	b8	b7	b6	b5	b4	b3	b2	b1	b0
ldi rx, #const3	rx = const5	1	0	0	0	const5					rx	rx	rx
ld rx, [ry]	rx = mem[ry]	0	0	1	0	0	0	ry	ry	ry	rx	rx	rx
st rx, [ry]	mem[ry] = rx	0	0	0	rx	rx	rx	ry	ry	ry	0	0	0
addc rx, ry, rz	rx = ry + rz + cin	0	1	0	ry	ry	ry	rz	rz	rz	rx	rx	rx
sbb rx, ry, rz	rx = ry - rz - cin	0	1	1	ry	ry	ry	rz	rz	rz	rx	rx	rx
jz address8	(Z==1) ? pc = address8 : pc = pc + 1	1	0	1	0	address8							
jmp offset6	pc = pc + offset6	1	1	0	0	0	0	offset6					
			OP_ALU		AA			AB			AD		
		OPCODE											

Foi discutido nas aulas as opções relativamente à posição de const5, address8 e offset6, bem como relativamente à codificação das instruções. A única instrução que tem um bit fixo, o b10, é a ld rx, [ry], porque que ry chegue à saída da ALU (para definir o endereço da RAM) é necessário que OP_ALU seja 0-.

b)

Considere que o PC = 0x40. Indique a gama de endereços possíveis de alcançar com a instrução JMP. A constante offset6 representa um número relativo (inteiro com sinal). O maior positivo a 6 bit é 01111b (0x0F), o menor negativo é 10000b (0x10).

A gama de endereços possíveis de alcançar é de [0x40 + 0xF0 = 0x30 até 0x40 + 0x0F = 0x4F]

Quando é realizada a extensão do bit de sinal do offset6 para 8 bit da constante 0x10, obtemos 0xF0.

c)

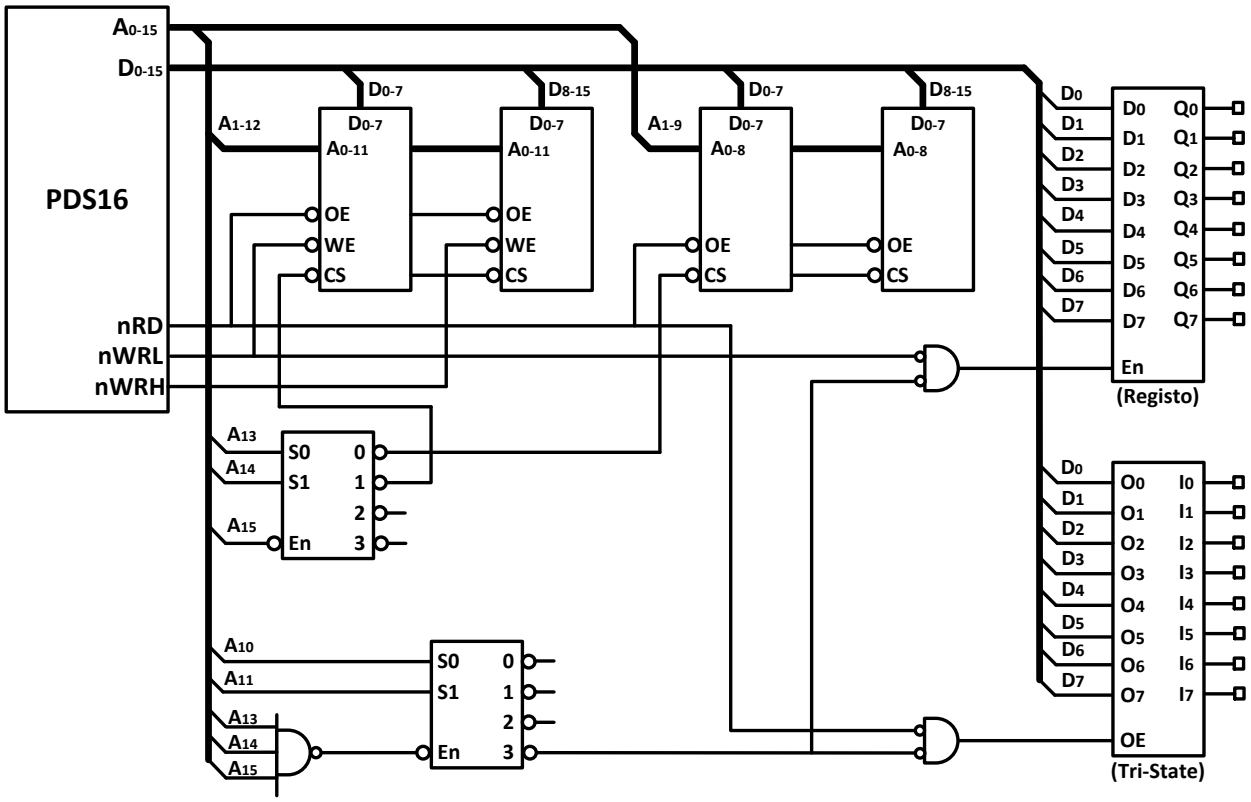
b2	b1	b0	Z	SO	SI	SD	ED	EC	EP	WR	RD
1	0	0	0	0	0	00	1	0	0	0	-
1	0	0	1	0	0	00	1	0	0	0	-
0	0	1	0	0	0	01	1	0	0	0	1
0	0	1	1	0	0	01	1	0	0	0	1
0	0	0	0	0	0	-	0	0	0	1	0
0	0	0	1	0	0	-	0	0	0	1	0
0	1	0	0	0	0	10	1	1	1	0	-
0	1	0	1	0	0	10	1	1	1	0	-
0	1	1	0	0	0	10	1	1	1	0	-
0	1	1	1	0	0	10	1	1	1	0	-
1	0	1	0	0	0	-	0	0	0	0	-
1	0	1	1	-	1	-	0	0	0	0	-
1	1	0	0	1	0	-	0	0	0	0	-
1	1	0	1	1	0	-	0	0	0	0	-
1	1	1	0	0	0	-	0	0	0	0	-
1	1	1	1	0	0	-	0	0	0	0	-

d)
Dimensão em bits da memória de código = 256 * 12 = 3072 bits
256 posições de memória * 12 bits que é a dimensão de cada instrução.

Dimensão em bits da ROM do módulo Descodificador = 8 * 2 * 9 = 144.
Posições da ROM são 16 = 8 instruções * 2 (porque tem de considerar para cada posição a flag de zero a 0 e a 1). Cada posição da ROM tem 9 bits para definir cada uma das saídas SO,SI,SD(2),ED,EC,EP,WR,RD.

e)
Como a const5 representa um número natural a extensão para 8 bit faz-se acrescentando 0.
const5_0 – constExt_0
const5_1 – constExt_1
const5_2 – constExt_2
const5_3 – constExt_3
const5_4 – constExt_4
0 – constExt_5
0 – constExt_6
0 – constExt_7

[2]



a) **Módulo ROM**
A0..8 = 512 Byte * 2 = 1 KByte ou 512 Word

O endereçamento interno do módulo ROM é de 0000h .. 03FFh [1 KByte] (10bits).

Como o DECODER que gera CS para a RAM e ROM utiliza o bit de Address A15 ligado ao En que é ativo com 0, isso significa que o DECODER só está ativo na gama de endereços de 0000h a 7FFFh (A15=0). Aos seletores do mesmo DECODER estão ligados A14 e A13, isso significa que o espaço de endereçamento está dividido em 4 blocos de 8KByte cada.

A_{15}	A_{14}	A_{13}	A_{12}	A_{11}	A_{10}	A_9	A_8	A_7	A_6	A_5	A_4	A_3	A_2	A_1	A_0	
0	0	0	X	X	X	X	X	X	X	X	X	X	X	X	X	0000h .. 1FFFh (8KByte)
0	0	1	X	X	X	X	X	X	X	X	X	X	X	X	X	0200h..3FFFh (8KByte)
0	1	0	X	X	X	X	X	X	X	X	X	X	X	X	X	0400h..5FFFh (8KByte)
0	1	1	X	X	X	X	X	X	X	X	X	X	X	X	X	0600h..7FFFh (8KByte)

Módulo RAM

A0..11 = 4 KByte * 2 = 8 KByte ou 4 KWord

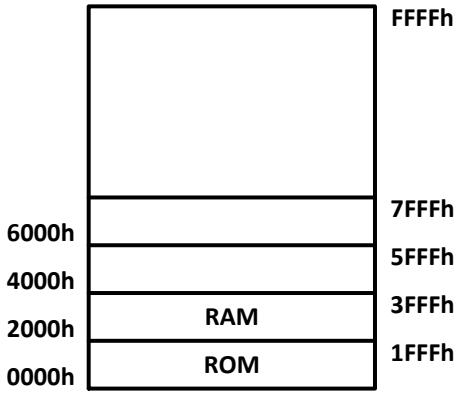
O endereçamento interno do módulo RAM é de 0000h .. 1FFFh [8 KByte] (13bits).

Como se pretende os endereços e dimensões que os módulos de ROM e RAM ocupam no espaço de endereçamento, a resposta seria:

ROM 8KByte de 0000h a 1FFFh e

RAM 8KByte de 2000h a 3FFFh

A ROM tem 7KByte de *foldback*, porque o dispositivo físico tem 1 KByte mas ocupa no espaço de endereçamento 8 KByte, logo temos 8KByte – 1KByte = 7KByte.



Mapa de Memória com os módulos de RAM e ROM

Portos de entrada/saída

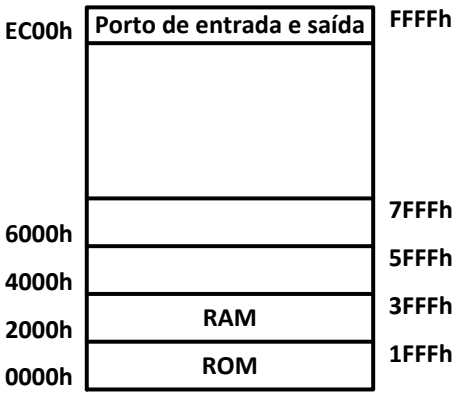
Como o DECODER que gera CS para os portos de entrada e saída utiliza os bit de Address A15, A14 e A13 ligado ao En que é ativo com 0, isso significa que o DECODER só está ativo na gama de endereços de E000h a FFFFh (A15, A14, A13 = 1). Aos seletores do mesmo DECODER estão ligados A10 e A11 e como a saída 3 é que faz CS, a gama de endereços é de EC00h a FFFFh.

A_{15}	A_{14}	A_{13}	A_{12}	A_{11}	A_{10}	A_9	A_8	A_7	A_6	A_5	A_4	A_3	A_2	A_1	A_0	
1	1	1	X	1	1	X	X	X	X	X	X	X	X	X	X	EC00h .. FFFFh

Como se pretende os endereços e dimensões que os portos de entrada e saída ocupam no espaço de endereçamento, a resposta seria:

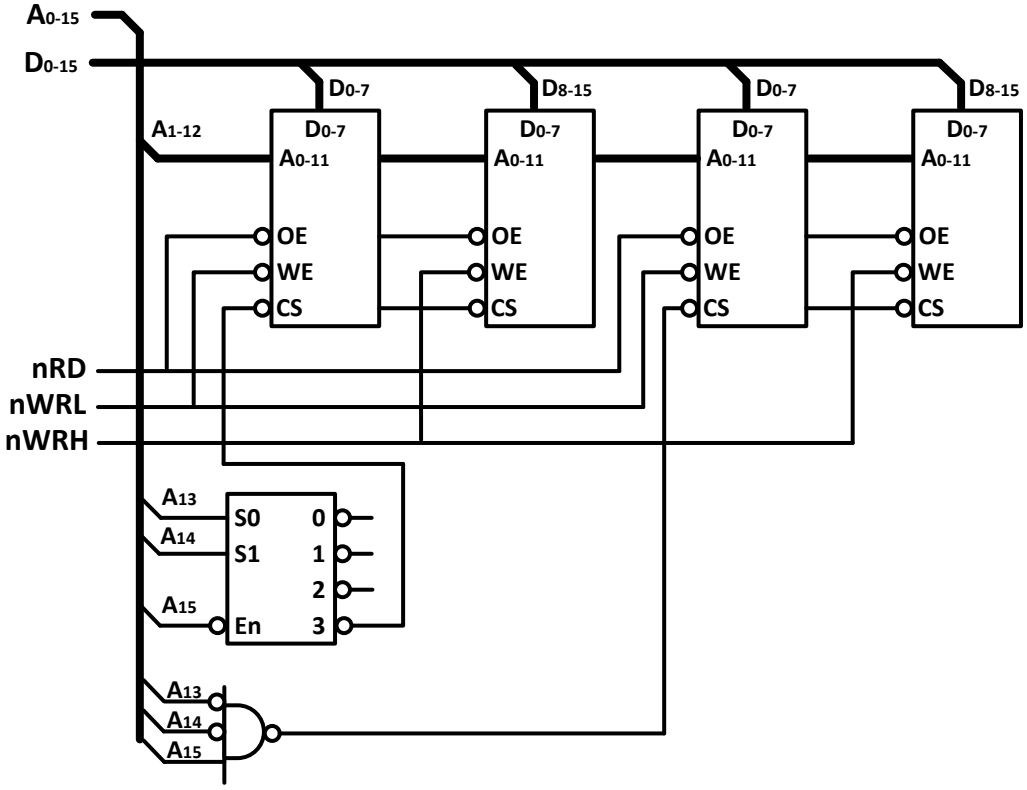
Portos de entrada e saída FFFFh - EC00h = 13FFh = 5KByte.

Os Portos de entrada e saída tem 5 KByte-1 de *foldback*, porque o dispositivo físico tem 1 Byte mas ocupa no espaço de endereçamento 5 KByte, logo temos 5 KByte – 1 Byte.

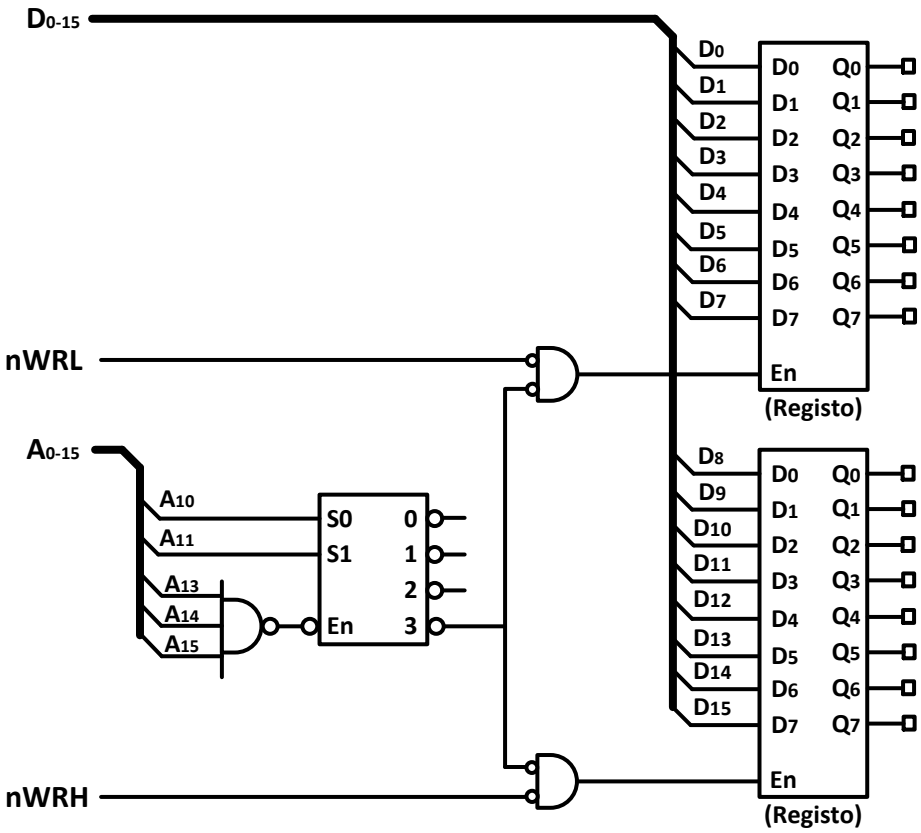


Mapa de Memória com os módulos de RAM e ROM e Portos de entrada saída

b)



c)


D0-15
nWRL
A0-15
A10
A11
A13
A14
A15
S0
S1
En
0
1
2
3
nWRH
D0
D1
D2
D3
D4
D5
D6
D7
D0
D1
D2
D3
D4
D5
D6
D7
Q0
Q1
Q2
Q3
Q4
Q5
Q6
Q7
En
(Registo)
D8
D9
D10
D11
D12
D13
D14
D15
D0
D1
D2
D3
D4
D5
D6
D7
Q0
Q1
Q2
Q3
Q4
Q5
Q6
Q7
En
(Registo)

[3]

```
        .section startup
        jmp     main

        .section directdata
        .org 4
values:
        .word   2, -3, 5, -9, -1
absvalues:
        .space  10

        .text
        .org 0x0080
main:
        ldi     r0, #low(absvalues)
        ldih    r0, #high(absvalues)
        ldi     r1, #low(values)
        ldih    r1, #high(values)
        ldi     r2, #5
        jmpl    copyabs
        jmp     $
;----------------------------------------
;uint8 valabs(int16 val) {
;   if (val<0)
;      return -val;
;   return val;
;}
;----------------------------------------
valabs:
        shl     r6, r0, #1, 0
        jnc     valabs_end
        not     r0, r0
        add     r0, r0, #1
valabs_end:
        ret
```

```
;----------------------------------------
;void copyabs(uint16 d[], int16 s[], uint16 n) {
;           while(n!=0) {
;             --n;
;             d[n] = valabs(s[n]);
;           }
;}
;----------------------------------------
copyabs:
            st          r3, copyabs_r3
            st          r5, copyabs_r5

            mov                 r3, r0
while:
            sub         r0, r2, #0
            jz          fim_copyabs
            sub         r2, r2, #1
            ld          r0, [r1, r2]
            jmpl        valabs
            st          r0, [r3, r2]
            jmp         while
fim_copyabs:
            ld          r3, copyabs_r3
            ld          r5, copyabs_r5
            ret

            .section directdata
copyabs_r3:
            .space   2
copyabs_r5:
            .space   2

            .end
```

[4]

```
        .EQU IO_PORT_ADDR,0xff00

        .section startup
        jmp     main

        .section directdata
        .org 4

counter:
        .word   0
state:
        .word   0

        .text
        .org 0x0080
main:
                                        ;       |D7|D6|D5|D4|D3|D2|D1|D0|
        jmpl    PORT_Read               ; r0 =  |S2 | X | X | X | X | X |S1 |
        ld      r1, state
        sub     r6, r1, #0
        jz      case0
        sub     r6, r1, #1
        jz      case1
        sub     r6, r1, #2
        jz      case2
        jmp     main
        jmp     $
case0:
        mov     r1, r0
        ld      r0, counter
        jmpl    PORT_Write
        shr     r6, r1, #1, 0
        jnc     end_case0
        ldi     r1, #1
        st      r1, state               ; estado = 1
        ldi     r1, #0
        st      r1, counter             ; counter = 0
end_case0:
        jmp     main

case1:
        shr     r6, r0, #1, 0           ; S1==0
        jnc     end_case1_0
        shr     r6, r0, #8, 0           ; S2==0
        jnc     end_case1
```

```
        ldi     r1, #2
        st      r1, state               ; estado = 2
        jmp     end_case1
end_case1_0:
        ldi     r1, #0
        st      r1, state
end_case1:
        jmp     main

case2:
        shr     r6, r0, #1, 0     ;S1==0
        jnc     end_case2_0
        shr     r6, r0, #8, 0     ;S2==0
        jc      end_case2
        ldi     r1, #1
        st      r1, state
        ld      r1, counter
        add     r1, r1, #1
        st      r1, counter
        jmp     end_case2
end_case2_0:
        ldi     r1, #0
        st      r1, state
end_case2:
        jmp     main

;---------------------------------------
;uint8 PORT_Read()
;---------------------------------------
PORT_Read:
        ldi     r0, #low(IO_PORT_ADDR)
        ldih    r0, #high(IO_PORT_ADDR)
        ldb     r0, [r0, #1]
        ret


;---------------------------------------
;void PORT_Write(uint8 val)
;---------------------------------------
PORT_Write:
        st      r1, PORT_Write_r1

        ldi     r1, #low(IO_PORT_ADDR)
        ldih    r1, #high(IO_PORT_ADDR)
        stb     r0, [r1, #1]
```

```
        ld      r1, PORT_Write_r1
        ret

        .section directdata
PORT_Write_r1:
        .space  2

        .end
```