B38DF
Computer Architecture and Embedded Systems

Instructions. A Simple Processor. Memory Address Decoding.

Alexander Belyaev

Heriot-Watt University
School of Engineering & Physical Sciences
Electrical, Electronic and Computer Engineering

E-mail: a.belyaev@hw.ac.uk
Office: EM2.29

Based on materials prepared by Dr. Mustafa Suphi Erden and Dr. Senthil Muthukumaraswamy

Assembly coding 1

Code the following in assembler (machine mnemonics)

D4 = D2 + D1 - D0

Load instruction—**MOV Ra, d**

- specifies the operation *RF[a]=D[d]*.

Store instruction—**MOV d, Ra**

- specifies the operation *D[d]=RF[a]*

Add instruction—**ADD Ra, Rb, Rc**

- specifies the operation *RF[a]=RF[b]+RF[c]*

Load-constant instruction—**0011 $r_3r_2r_1r_0$ $c_7c_6c_5c_4c_3c_2c_1c_0$**

- **MOV Ra, #c**—specifies the operation *RF[a]=c*

Subtract instruction—**0100 $ra_3ra_2ra_1ra_0$ $rb_3rb_2rb_1rb_0$ $rc_3rc_2rc_1rc_0$**

- **SUB Ra, Rb, Rc**—specifies the operation *RF[a]=RF[b] – RF[c]*

Jump-if-zero instruction—**0101 $ra_3ra_2ra_1ra_0$ $o_7o_6o_5o_4o_3o_2o_1o_0$**

- **JMPZ Ra, offset**—specifies the operation *PC = PC + offset* if *RF[a]* is 0

Assembly coding 1

Code the following in assembler (machine mnemonics)

D4 = D2 + D1 - D0

```
MOV R2, D2
MOV R1, D1
MOV R0, D0
MOV R4,#0
ADD R4,R1,R2
SUB R4,R4,R0
MOV D4,R4
```

Load instruction—**MOV Ra, d**

- specifies the operation *RF[a]=D[d]*.

Store instruction—**MOV d, Ra**

- specifies the operation *D[d]=RF[a]*

Add instruction—**ADD Ra, Rb, Rc**

- specifies the operation *RF[a]=RF[b]+RF[c]*

Load-constant instruction—**0011** $r_3r_2r_1r_0$ $c_7c_6c_5c_4c_3c_2c_1c_0$

- **MOV Ra, #c**—specifies the operation *RF[a]=c*

Subtract instruction—**0100** $ra_3ra_2ra_1ra_0$ $rb_3rb_2rb_1rb_0$ $rc_3rc_2rc_1rc_0$

- **SUB Ra, Rb, Rc**—specifies the operation *RF[a]=RF[b] – RF[c]*

Jump-if-zero instruction—**0101** $ra_3ra_2ra_1ra_0$ $o_7o_6o_5o_4o_3o_2o_1o_0$

- **JMPZ Ra, offset**—specifies the operation *PC = PC + offset* if *RF[a]* is 0

Assembly coding 2

Code the following in assembler (machine mnemonics) .

N is stored in D[9]

```
i = 0;
sum = 0;
while ( i != N ){
   sum = sum + i;
   i = i + 2;
}
```

Load instruction—**MOV Ra, d**

- specifies the operation *RF[a]=D[d]*.

Store instruction—**MOV d, Ra**

- specifies the operation *D[d]=RF[a]*

Add instruction—**ADD Ra, Rb, Rc**

- specifies the operation *RF[a]=RF[b]+RF[c]*

Load-constant instruction—**0011** $\mathbf{r_3r_2r_1r_0}$ $\mathbf{c_7c_6c_5c_4c_3c_2c_1c_0}$

- **MOV Ra, #c**—specifies the operation *RF[a]=c*

Subtract instruction—**0100** $\mathbf{ra_3ra_2ra_1ra_0}$ $\mathbf{rb_3rb_2rb_1rb_0}$ $\mathbf{rc_3rc_2rc_1rc_0}$

- **SUB Ra, Rb, Rc**—specifies the operation *RF[a]=RF[b] – RF[c]*

Jump-if-zero instruction—**0101** $\mathbf{ra_3ra_2ra_1ra_0}$ $\mathbf{o_7o_6o_5o_4o_3o_2o_1o_0}$

- **JMPZ Ra, offset**—specifies the operation *PC* = *PC* + *offset* if *RF[a]* is 0

Assembly coding 2

Code the following in assembler (machine mnemonics) .

N is stored in D[9]

```
i = 0;
sum = 0;
while ( i != N ){
   sum = sum + i;
   i = i + 2;
}
```

```
       MOV R0, #0     // R0 is "i"
       MOV R1, #0     // R1 is "sum"
       MOV R2, #2
       MOV R3, D[9]
       ADD R4, #0   // for looping
loop:   SUB R4,R3,R0 // R4 = N - i
        JMPZ R4, done
       ADD R1, R1, R0 // sum= sum+i
       ADD R0 R0, R2 //   i  =  i + 2
       JMPZ R4, loop
done:
```

Load instruction—**MOV Ra, d**

- specifies the operation *RF[a]=D[d]*.

Store instruction—**MOV d, Ra**

- specifies the operation *D[d]=RF[a]*

Add instruction—**ADD Ra, Rb, Rc**

- specifies the operation *RF[a]=RF[b]+RF[c]*

Load-constant instruction—**0011 $r_3r_2r_1r_0$ $c_7c_6c_5c_4c_3c_2c_1c_0$**

- **MOV Ra, #c**—specifies the operation *RF[a]=c*

Subtract instruction—**0100 $ra_3ra_2ra_1ra_0$ $rb_3rb_2rb_1rb_0$ $rc_3rc_2rc_1rc_0$**

- **SUB Ra, Rb, Rc**—specifies the operation *RF[a]=RF[b] – RF[c]*

Jump-if-zero instruction—**0101 $ra_3ra_2ra_1ra_0$ $o_7o_6o_5o_4o_3o_2o_1o_0$**

- **JMPZ Ra, offset**—specifies the operation *PC = PC + offset* if *RF[a]* is 0

Assembly coding 3

A programme is required to output the Fibonacci series. Using the formula y(n) = y(n-2) + y(n-1), where y(n) is the current number and y(n-1) and y(n-2) are previous two numbers of the series. Write a programme in machine mnemonics to output the first 10 numbers. Start from y(0) = 0 and y(1) = 1.

Load instruction—**MOV Ra, d**

- specifies the operation *RF[a]=D[d]*.

Store instruction—**MOV d, Ra**

- specifies the operation *D[d]=RF[a]*

Add instruction—**ADD Ra, Rb, Rc**

- specifies the operation *RF[a]=RF[b]+RF[c]*

Load-constant instruction—**0011** $\mathbf{r_3r_2r_1r_0\ c_7c_6c_5c_4c_3c_2c_1c_0}$

- **MOV Ra, #c**—specifies the operation *RF[a]=c*

Subtract instruction—**0100** $\mathbf{ra_3ra_2ra_1ra_0\ rb_3rb_2rb_1rb_0\ rc_3rc_2rc_1rc_0}$

- **SUB Ra, Rb, Rc**—specifies the operation *RF[a]=RF[b] – RF[c]*

Jump-if-zero instruction—**0101** $\mathbf{ra_3ra_2ra_1ra_0\ o_7o_6o_5o_4o_3o_2o_1o_0}$

- **JMPZ Ra, offset**—specifies the operation *PC* = *PC* + *offset* if *RF[a]* is 0

Memory Address Decoding 1

Design a **fully-decoded** positive-logic address decoder for *Your Device* in the following system. (Design just the decoder for *Your Device*, not all of them) The inputs are A15, A14, …A0 and the output is **Select**. Positive logic means **Select**=1 when *Your Device* should be activated, and **Select**=0 when *Your Device* is deactivated. Memory addresses are specified in hexadecimal numbers ($ is used to denote that we deal with hexadecimal numbers).

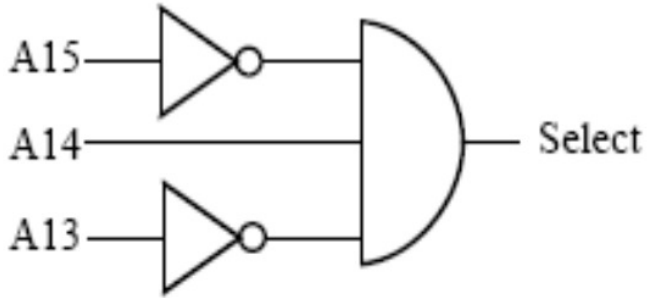
RAM \$0000-\$0FFF
Your Device \$4000-\$5FFF
I/O ports \$8010-\$801F
ROM \$C000-\$FFFF

Memory Address Decoding 1: Solution

Design a **fully-decoded** positive-logic address decoder for *Your Device* in the following system. (Design just the decoder for *Your Device*, not all of them) The inputs are A15, A14, …A0 and the output is **Select**. Positive logic means **Select**=1 when *Your Device* should be activated, and **Select**=0 when *Your Device* is deactivated. Memory addresses are specified in hexadecimal numbers ($ is used to denote that we deal with hexadecimal numbers).

RAM	$0000-$0FFF
Your Device	$4000-$5FFF
I/O ports	$8010-$801F
ROM	$C000-$FFFF

Fully-decoded means **Select** if and only if the address is $4000 to $5FFF.
Write $4000-$5FFF as 010X, XXXX, XXXX, XXXX
Give logic equation directly using all 0's and 1's: **Select** = not(A15)*A14*not(A13)



Memory Address Decoding 2

Using logic gates, design a **minimal cost**, positive-logic address decoder for a component *Your Device* in a small embedded microprocessor system. You do not have to design all four address decoders, just the one for *Your Device*.

The system address bus has 16 address inputs A15 →A0 and an output signal **Select**. The decoder inputs are connected to appropriate address lines to map *Your Device* into the required address range in the memory map. Positive logic means **Select** = 1 when *Your Device* is activated, and **Select** = 0 when *Your Device* is deactivated. Memory addresses are specified in hexadecimal numbers.

RAM $D000-$D3FF
Your Device $D800-$DFFF
ROM $E000-$FFFF
I/O Ports $5800-$58FF

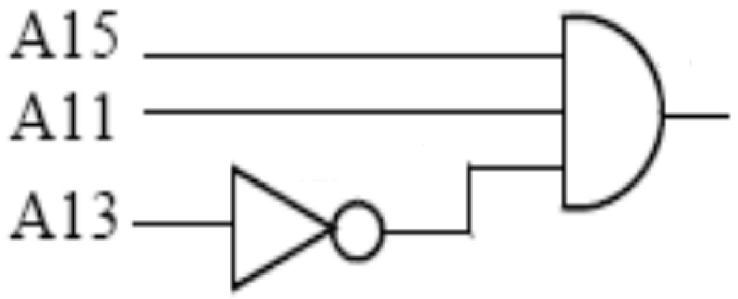
Memory Address Decoding 2: Solution

Using logic gates, design a **minimal cost**, positive-logic address decoder for a component *Your Device* in a small embedded microprocessor system. You do not have to design all four address decoders, just the one for *Your Device*.

The system address bus has 16 address inputs A15 →A0 and an output signal **Select**. The decoder inputs are connected to appropriate address lines to map *Your Device* into the required address range in the memory map. Positive logic means **Select** = 1 when *Your Device* is activated, and **Select** = 0 when *Your Device* is deactivated. Memory addresses are specified in hexadecimal numbers.

RAM $D000-$D3FF	1101 00XX XXXX XXXX
Your Device $D800-$DFFF	1101 1XXX XXXX XXXX
ROM $E000-$FFFF	111X XXXX XXXX XXXX
I/O Ports $5800-$58FF	0101 1000 XXXX XXXX

Your Device Select = A15 • not(A13) • A11



Memory Address Decoding 3

Using logic gates, design a **minimal cost**, positive-logic address decoder for a component *Your Device in* a small embedded microprocessor system. The system address bus has 16 address inputs A15 $\rightarrow$A0 and an output signal **Select**. The decoder inputs are connected to appropriate address lines to map *Your Device* into the required address range in the memory map. Positive logic means **Select** = 1 when *Your Device* is activated, and **Select** = 0 when *Your Device* is deactivated. Memory addresses are specified in hexadecimal numbers.

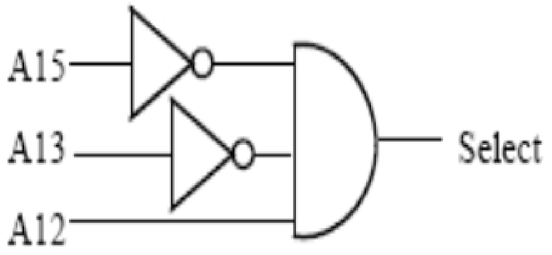
RAM $4000-$47FF
Your Device $5000-$57FF
ROM $6000-$7FFF
I/O $C000-$FFFF

Memory Address Decoding 3: Solution

Using logic gates, design a **minimal cost**, positive-logic address decoder for a component *Your Device in* a small embedded microprocessor system. The system address bus has 16 address inputs A15 →A0 and an output signal **Select**. The decoder inputs are connected to appropriate address lines to map *Your Device* into the required address range in the memory map. Positive logic means **Select** = 1 when *Your Device* is activated, and **Select** = 0 when *Your Device* is deactivated. Memory addresses are specified in hexadecimal numbers.

RAM $4000-$47FF	**0100, 0XXX, XXXX, XXXX**
YourDevice $5000-$57FF	**0101, 0XXX, XXXX, XXXX**
ROM $6000-$7FFF	**011X, XXXX, XXXX, XXXX**
I/O $C000-$FFFF	**11XX, XXXX, XXXX, XXXX**

Select = not(A15) * not(A13) * A12


A15
A13
A12
Select