Lecture 8

First Instructions III

Assembly Language is Simple

A few simple instructions, a few simple addressing modes, simple syntax

Yet very powerful

But assembly does not provide the constructs of higher level languages that programmers rely on: no *variables*, no *arrays*, no loops, no if-else etc.

We will have to do the work ourselves: use labels, keep track of the nature of the data we store in memory locations (e.g. signed/unsigned, byte/word) etc.

Today: More on arrays, more instructions, more addressing modes

Also more hands on coding

Next week: Conditional jumps and how to do loops, ifs etc.

Assembly Instructions

All instructions have one, two or no operands

These operands are either

- an **immediate value** or
- a **memory location** (register) where we can read/write the value

  the memory location can be **memory mapped** specified by an address

  a **core register** specified by R0 - R15

e.g.,

```
mov.w   #0x2400, SP
mov.w   R4, &0x1C20
add.w   &0x1C00, &0x1C08
```

copy the content of memory location with address `0x1C00` to memory location with address `0x1C08`

or

```
jmp     0x441A
```

execute the instruction at memory location with `0x441A`

Very difficult to keep track of all the addresses, so we use **labels**

Labels in Assembly

A **label** is simply a name that we give to the address of a memory location

e.g.,

```
        .data
array1: .word  2, 3, 5, 7          array1 = 0x1C00
array2: .space 8                   array2 = 0x1C08
```


Then instead of


```
        add.w  &0x1C00, &0x1C08
```


we write

```
        add.w  &array1, &array2
```

absolute address mode

We have even more tricks such as **symbolic address mode**

= indexed address w.r.t. PC

```
        add.w array1, array2
```

Labels in Assembly

A **label** is simply a name that we give to the address of a memory location

Applies to memory locations that hold data – we treat these as **variables**

but also to memory locations that hold **instructions**

e.g.


```
loop:    jmp   loop
```

```
loop = 0x441A
```

Then instead of

```
         jmp   0x441A
```

where `0x441A` is the address in FRAM where this instruction is

note that we would need to compile first with placeholders, figure out all addresses and then enter them in our code

we write

```
loop:    jmp   loop
```

So Far

Five instructions

```
mov.w    src, dst
add.w    src, dst
rra.w    dst
jmp      label
nop
```

These instructions also have a byte version

Hint: For Quiz #3 use

```
rla.w    dst
```

Plus a few emulated instructions

`clr.w dst`	same as	`mov.w`	`#0, dst`
`inc.w dst`	same as	`add.w`	`#1, dst`
`incd.w dst`	same as	`add.w`	`#2, dst`

So Far

Five addressing modes

e.g.,

```
mov.w    src, dst
```


- **Immediate data #N:** `src` is the value given after #
- **Absolute address &ADDR:** the memory address of `src` or `dst` is given after &

- **Register mode Rn:** `src` or `dst` is one of the core registers R0 - R15

- **Symbolic mode X:** where X is simply a label
  (the memory address of `src` or `dst` is X + PC)

- **Indexed mode X(Rn):** the memory address of `src` or `dst` is X + Rn
  i.e., (X + Rn) points to the `src` or `dst`

Arrays in Assembly

When emulating arrays in assembly we must be careful with the *index*

We increment the index by 2 for words but only by 1 for bytes !

```
                array1   array1+2   array1+4
array1: .word   0x0100,  0x0200,   0x0300

                array2   array2+1   array2+2
array2: .byte   0x10,    0x20,     0x30
```


Task: Find the sum of all numbers in the array

We will do this – using **indexed mode** of addressing
and **indirect register mode**
and **indirect autoincrement register mode**

Indexed Mode and **Word** Arrays

```
array1: .word  0x0100,  0x0200,  0x0300

        mov.w  &array1, R5        array1(R4)   where R4 = 0
        add.w  &array1+2, R5      array1(R4)   where R4 = 2
        add.w  &array1+4, R5      array1(R4)   where R4 = 4
```


Alternatively


```
        mov.w  #0, R4             ; R4 = 0 will be the index
        mov.w  array2(R4), R5     ; R5 = array2[R4]
        inc.w  R4                 ; R4++
        inc.w  R4                 ; R4++
        add.w  array2(R4), R5     ; R5 += array2[R4]
        incd.w  R4                ; R4 = R4 + 2
        add.w  array2(R4), R5     ; R5 += array2[R4]
```

Indexed Mode and **Byte** Arrays

Rewrite our previous example using indexed mode

```
array2: .byte  0x10,   0x20,   0x30
```

```
        mov.b  &array2, R5          array1(R4)   where R4 = 0
        add.b  &array2+1, R5        array1(R4)   where R4 = 1
        add.b  &array2+2, R5        array1(R4)   where R4 = 2
```


Hence

```
        mov.w  #0, R4               ; index R4 = 0       - 16-bit
        mov.b  array2(R4), R5       ; R5 = array2[R4]    - 8-bit
        inc.w  R4                   ; R4++               - 16-bit
        add.b  array2(R4), R5       ; R5 += array2[R4]   - 8-bit
        inc.w  R4                   ; R4++               - 16-bit
        add.b  array2(R4), R5       ; R5 += array2[R4]   - 8-bit
```

Indirect Register Mode

Indirect Register Mode of addressing works **only for the source!**

Syntax

```
mov.w   @R4, R5
```


Copy word from address in R4 to the destination

e.g.:


```
        .data
var1:   .word 0x2357            var1 = 0x1C00
```



```
        mov.w   #var1, R4       ; R4 contains the address of the src
        mov.w   @R4, R5         ; R5 <- 0x2357
```

same *effect* as

```
        mov.w   &var1, R4       ; address is hardcoded
```

Indirect Autoincrement Register Mode

Indirect Autoincrement Register Mode works **only for the source!**

Syntax

```
mov.w   @R4+, R5
```


Copy word from address in R4 to R5 then increment R4 so it points to next word in array!!


```
array1: .word ...
```



```
        mov.w   #var1, R4     ; R4 contains the address of the src
        mov.w   @R4+, R5
```

same *effect* as

```
        mov.w   @var1, R4
        incd.w  R4            ; because a word was fetched
```