

The Von Neumann Model A Programmer's Perspective –Computer Organization –

Lionel Morel

Computer Science and Information Technologies - INSA Lyon

Fall-Winter 2023-24

Outline

Overview

Instruction Set

Micro-Machine

Basic Instructions

Addressing Modes

Control-Flow

Outline

Overview

Instruction Set

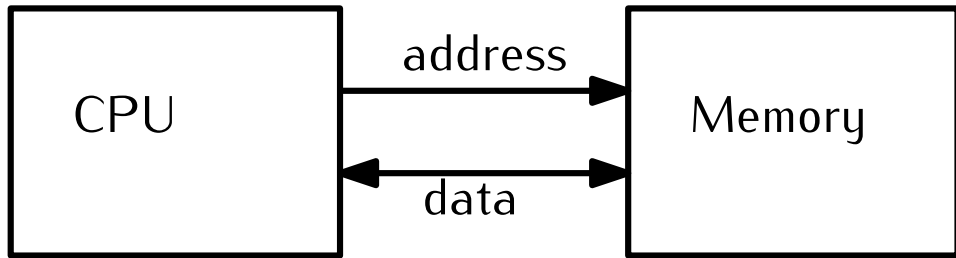
Micro-Machine

Basic Instructions

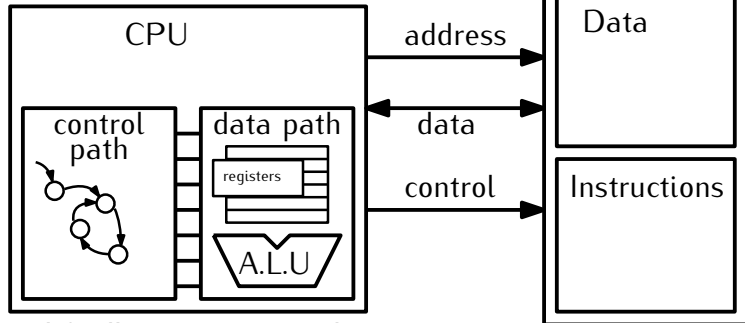
Addressing Modes

Control-Flow

General View



Von Neumann Architecture

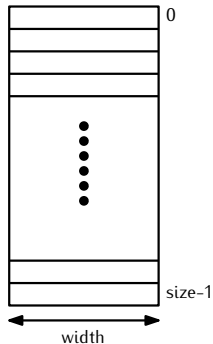


- ▶ At the heart of (nearly) all computers today.
- ▶ Contains:
 - ▶ A unique storage where both instructions and data are stored, called **central memory**
 - ▶ A **processor** made of:
 - ▶ An **data-path** including an Arithmetic and Logical Unit
 - ▶ A **control-path**
 - ▶ An **input/output** system that interconnects peripherals (not depicted for now)

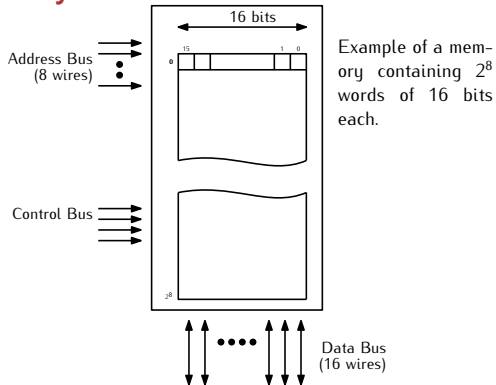
Central Memory

- ▶ Memory contains a finite number of information
- ▶ All these elements are **encoded in binary**
- ▶ An item in memory is accessed through its **address**

Logical View



Physical View



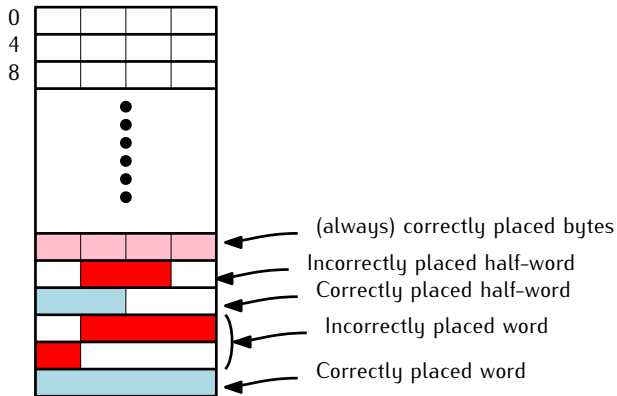
Central Memory - Data and addresses

- ▶ Data is organized in packets of bytes:
 - ▶ 1 **byte** = 8 bits
 - ▶ 2 bytes (16 bits) = **half-word**
 - ▶ 4 bytes (32 bits) = **word**
 - ▶ 8 bytes (64 bits) = **double word**
- ▶ Each byte has an address.
 - ▶ Example: if A is the address of a **word**, then the **next word** is at address $A + 4$

Central Memory - Memory Alignment

In most systems a data piece can be stored at specific addresses only:

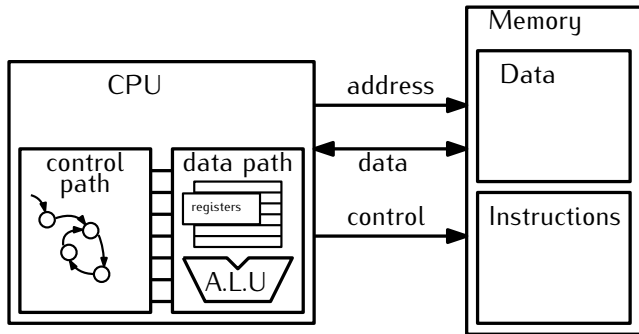
Size	Address should be ...
byte	... whatever
half-word	... even
word	... multiple of 4
double	... multiple of 8



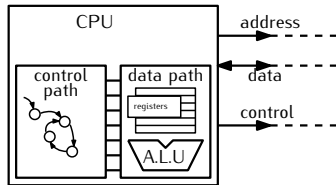
Central Memory - Interaction with CPU

The CPU:

- ▶ Sends address it wants to READ/WRITE on the **address bus**
- ▶ Reads (writes) data it wants to READ (WRITE) from (to) memory on the **data bus**
- ▶ Manages memory (including the above signals) through **control bus**.



The CPU - A Von Neumann Machine



The Von Neumann Cycle

```
forever do{  
    Fetch Instruction from Memory  
    Decode Instruction  
    Execute Instruction  
}
```

The Processor

Externally

Application Binary Interface describes:

- ▶ what are the **width of data and address buses**
- ▶ what **types of data** are available
- ▶ what **registers** are available to programmers
- ▶ what **instructions** can be used

Internally

- ▶ Dedicated **Registers**
- ▶ a **DataPath** interconnecting these registers with combinatorial logic
 1. perform calculations
 2. select data to update registers with
- ▶ a **Control Unit** (or path), ie an Algorithmic State Machine, to ... control the rest

Today

- ▶ We look at the **external vision** of the CPU: ABI, instructions, assembler, etc.

Next time

- ▶ We will look at the **internal vision** of the CPU: registers, datapath, control unit, VN cycle, etc

Programming Languages

- ▶ Architecture-Independent
- ▶ Machine details are abstracted away
- ▶ Programming concepts may be elaborate (object, data structures, patterns, etc)
- ▶ Examples: C, C++, Java, Python, Ruby, etc.

Machine Language vs Assembly Language

Machine Language

- ▶ A **Program** is a sequence of **binary instructions**
- ▶ ie **Instructions** are “just” **sequences of bits**, 0s and 1s.
- ▶ Each instruction is **interpreted by the CPU** and triggers internal changes to it so that the corresponding **behavior** is actually applied
- ▶ Each model of processor has its own machine language.

Assembly Language

- ▶ Very close to the Machine Language
- ▶ Instructions are “human-readable”
- ▶ 1 instruction in ASM → 1 instruction in machine language
- ▶ Some humans write/read them

Rmk: Both are specific to a processor family!

Outline

Overview

Instruction Set

Micro-Machine

Basic Instructions

Addressing Modes

Control-Flow

Instruction Set

We want to look at the code actually executed by the processor

⇒ Binary/Assembly

Each processor implements a specific **Instruction Set**, ie a set of instructions that programs can use to make the processor perform the required actions.

ISA (Instruction Set Architecture)

Examples:

- ▶ IA32 or IA64: Intel's 32-bits or 64 bits Instruction Set
- ▶ ARM
- ▶ MSP430
- ▶ RISC-V

Instruction Set Architecture

The ISA defines:

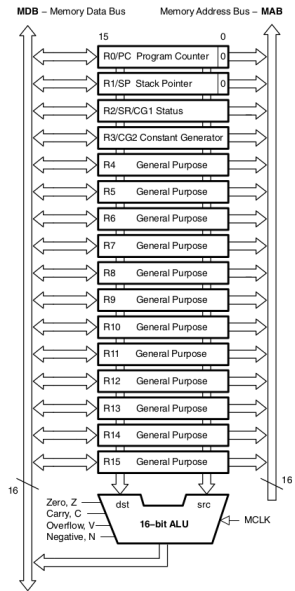
- ▶ The **types of data** that can be manipulated (typically ints of various sizes, Boolean fields, floats)
- ▶ **Instructions** to ...
- ▶ ... **manipulate** this **data**
- ▶ ... **access memory**
- ▶ ... **control the flow of execution**
- ▶ ... help **synchronize** different execution threads (see Operating Systems)
- ▶ ... handle **Input/Output devices**

An Example ISA - the msp430

- ▶ It's a 16-bits machine, ie:
 - ▶ Addresses are on 16 bits
 - ▶ You have 16 registers R_0 to R_{15}
 - ▶ NB: Registers R_0 to R_3 are reserved, don't use them!
- ▶ We'll look at the core ISA:
 - ▶ Computational instructions
 - ▶ Control-flow instructions
 - ▶ Memory-access instructions
- ▶ We will look at:
 - ▶ Syntax (how do you write the instruction?)
 - ▶ Semantics (what does the instruction do?)
 - ▶ Encoding (how is the instruction seen by the processor?)

msp430 - Available registers

- ▶ Basic local memory for the CPU to compute with
- ▶ ie they are **INSIDE** the CPU!
- ▶ Noted R_0 to R_{15}
- ▶ The ABI says:
 - ▶ R_0 , R_1 , R_2 and R_3 have dedicated functions.
 - ▶ R_4 to R_{15} are working registers for general use.



Instructions

An instruction can be characterized by:

- ▶ the **type of operations** it implements: computational, control-flow, memory
- ▶ its format: one, two or three operands, addressing modes allowed

Addressing Modes

Generally speaking, defines how the instructions identify their operands.

Two major styles of ISA

Register-memory Architectures

- ▶ Allows **all operations to be performed on memory as well as registers.**
- ▶ **msp430**, Motorola 68000, x86

Load-Store Architectures

- ▶ **explicit memory acces** instructions (typically *load from-* and *store to-* memory)
- ▶ **register-only ALU** instructions
- ▶ eg ARM, MIPS, RISC-V, the **micro-machine**

Outline

Overview

Instruction Set

Micro-Machine

Basic Instructions

Addressing Modes

Control-Flow

The micro-machine (1/3)

In the first lab sessions, we'll look at a home-made processor.

- ▶ 8-bits instructions
- ▶ 8-bits signed integers only
- ▶ 2 8-bits registers, named A and B

Computation instructions with 1 or 2 operands

```
B -> A          21 -> B
B + A -> A       B xor -42 ->
                  A
not B -> A       lsr A -> A
A xor 12 -> A    B - A -> A;
```

WARNING: some instructions that would seem “intuitive” are actually forbidden... eg: $A+B \rightarrow B$ is incorrect ... $B+A \rightarrow B$ is correct.

The micro-machine (2/3)

Memory reads and writes

$*A \rightarrow A$	$*A \rightarrow B$
$A \rightarrow *A$	$B \rightarrow *A$
$*cst \rightarrow A$	$*cst \rightarrow B$
$A \rightarrow *cst$	$B \rightarrow *cst$

***A means: “the content of memory at the address contained in register A”.**

The micro-machine (3/3)

Unconditional absolute branch

JA 42

continues execution at address 42.

Conditional relative branch

JR offset

JR offset IFZ

(executed if Z=1)

JR offset IFC

JR offset IFN

(executed if C=1)

(executed if N=1)

The micro-machine how-to

- ▶ During the first lab, you will explore the syntax and semantics and **Assemble instructions yourself**, ie write machine-language from a given assembly language program.
- ▶ Later on, you will have a tool to do that for you. This tool is called an **Assembler**.

Outline

Overview

Instruction Set

Micro-Machine

Basic Instructions

Addressing Modes

Control-Flow

Basic Instructions

- ▶ Perform a **logical** or **integer** operation on its arguments
- ▶ These operations include:
 - ▶ ADD(-ition), SUB(-straction)
 - ▶ SLL, SLT, SRL, SRA: Shift Left/Right Logical/Arithmetic
 - ▶ XOR, OR, AND, NOT: Boolean operations
 - ▶ Double or single-operand
 - ▶ Some instructions modify the Status Bits, aka FLAGS.

Micro-machine - logical and integer operations

Computation instructions with 1 or 2 operands

B -> A	21 -> B
B + A -> A	B xor -42 ->
	A
not B -> A	lsr A -> A
A xor 12 -> A	B - A -> A;

msp430 - Double-Operand instructions

Syntax and Semantics

operation X_s, X_d

implements

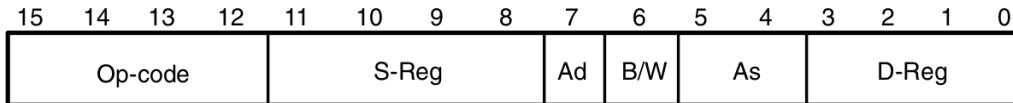
src operation dst \rightarrow dst

or *operation src \rightarrow dst*

or *src operation dst*

Mnemonic	S-Reg, D-Reg	Operation	Status Bits			
			V	N	Z	C
MOV(.B)	src, dst	src \rightarrow dst	-	-	-	-
ADD(.B)	src, dst	src + dst \rightarrow dst	*	*	*	*
ADDC(.B)	src, dst	src + dst + C \rightarrow dst	*	*	*	*
SUB(.B)	src, dst	dst + .not.src + 1 \rightarrow dst	*	*	*	*
SUBC(.B)	src, dst	dst + .not.src + C \rightarrow dst	*	*	*	*
CMP(.B)	src, dst	dst - src	*	*	*	*
DADD(.B)	src, dst	src + dst + C \rightarrow dst (decimally)	*	*	*	*
BIT(.B)	src, dst	src .and. dst	0	*	*	*
BIC(.B)	src, dst	.not.src .and. dst \rightarrow dst	-	-	-	-
BIS(.B)	src, dst	src .or. dst \rightarrow dst	-	-	-	-
XOR(.B)	src, dst	src .xor. dst \rightarrow dst	*	*	*	*
AND(.B)	src, dst	src .and. dst \rightarrow dst	0	*	*	*

Encoding



msp430 - Single-Operand

Syntax and Semantics

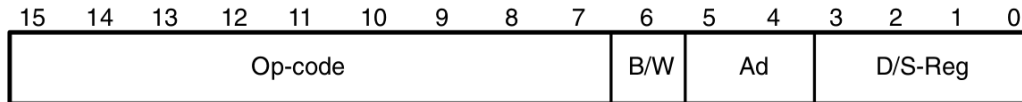
operation $X_{s/d}$

implements

operation src/dst

Mnemonic	S-Reg, D-Reg	Operation	Status Bits			
			V	N	Z	C
RRC (.B)	dst	C → MSB →LSB → C	*	*	*	*
RRA (.B)	dst	MSB → MSB →LSB → C	0	*	*	*
PUSH (.B)	src	SP - 2 → SP, src → @SP	-	-	-	-
SWPB	dst	Swap bytes	-	-	-	-
CALL	dst	SP - 2 → SP, PC+2 → @SP dst → PC	-	-	-	-
RETI		TOS → SR, SP + 2 → SP TOS → PC, SP + 2 → SP	*	*	*	*
SXT	dst	Bit 7 → Bit 8.....Bit 15	0	*	*	*

Encoding



Outline

Overview

Instruction Set

Micro-Machine

Basic Instructions

Addressing Modes

Control-Flow

Addressing Modes

- ▶ So far, we used only registers in computation. But we need to get data from memory as well.
- ▶ in **register-memory** architectures, arithmetic and logical instructions can work indifferently on registers and memory
- ▶ in **load-store** architectures, arithmetic and logical instructions only work on register. Dedicated instructions are available to copy data from memory to registers (LOAD) and reversly (STORE)

msp430 - addressing modes

An example of a **register-memory** architecture.

As/Ad	Addressing Mode	Syntax	Description
00/0	Register mode	Rn	Register contents are operand
01/1	Indexed mode	X(Rn)	(Rn + X) points to the operand. X is stored in the next word.
01/1	Symbolic mode	ADDR	(PC + X) points to the operand. X is stored in the next word. Indexed mode X(PC) is used.
01/1	Absolute mode	&ADDR	The word following the instruction contains the absolute address. X is stored in the next word. Indexed mode X(SR) is used.
10/-	Indirect register mode	@Rn	Rn is used as a pointer to the operand.
11/-	Indirect autoincrement	@Rn+	Rn is used as a pointer to the operand. Rn is incremented afterwards by 1 for .B instructions and by 2 for .W instructions.
11/-	Immediate mode	#N	The word following the instruction contains the immediate constant N. Indirect autoincrement mode @PC+ is used.

Micro-machine - memory instructions

An example of a **load-store** architecture.

Memory reads and writes

$*A \rightarrow A$	$*A \rightarrow B$
$A \rightarrow *A$	$B \rightarrow *A$
$*cst \rightarrow A$	$*cst \rightarrow B$
$A \rightarrow *cst$	$B \rightarrow *cst$

***A means: “the content of memory at the address contained in register A”.**

Outline

Overview

Instruction Set

Micro-Machine

Basic Instructions

Addressing Modes

Control-Flow

Control-Flow Instructions

- ▶ The general execution model is **sequential**
- ▶ Instructions are executed by the processor **one after the other, in the order they are written in the binary program.**
- ▶ Algorithms are usually more complex than that.
They use **control structures** such as if-then-else, for and while loops, etc.
- ▶ At the machine level, these control structures are built using:
 - ▶ **basic blocks:** *“A sequence of contiguous instructions that contains no jumps or labels.”*
 - ▶ **control-flow instructions:** instructions that allow to **jump** or **branch** to any address in the code. Essentially, jumping from the end of a block to start of another one.

Labels

- ▶ In assembly¹, we use labels to have named references on memory cells
- ▶ A label is a string ending with the ":" character
- ▶ It can be used to reference:
 - ▶ a line of program

```
int main(){  
    int a,b;  
    a = 0;  
    b = 10;  
    while (a<b){  
        a++;  
    }  
}
```

- ▶ a data item

```
int X;  
  
int main(){  
    int a,b,c;  
    X = 42;  
}
```

```
main:  
    SUB.W #4, R1  
    MOV.W #0, 2(R1)  
    MOV.W #10, @R1  
    BR    #.L2  
.L3:  
    ADD.W #1, 2(R1)  
.L2:  
    CMP.W @R1, 2(R1) [ JL    .L3  
    MOV.B #0, R12  
    ADD.W #4, R1  
    RET
```

```
31 X:  
32    .word 0
```

```
main:  
    MOV.W #42, &X  
    MOV.B #0, R12  
    RET
```

¹you can observe these on <https://gcc.gdbolt.org/>

Jump instructions

- ▶ “*jump*” and “*branch*” are (almost) interchangeable;
- ▶ **unconditionnal jumps**: “goto” some place, whenever we execute the branch;
- ▶ **conditionnal jumps**: test a condition to decide whether to jump or not.

Unconditionnal Jumps

- ▶ Forces a **new address** `addr` to PC
- ▶ The next instruction executed is the one located at *Mem[addr]*

Conditionnal Jumps

- ▶ Decide whether to branch or not, based on a condition
- ▶ Condition can rely on the value of CPU **flags**
 - ▶ JC = “Jump if Carry [flag is set]”
 - ▶ JZ = “Jump if Zero [flag is set]”
 - ▶ That's the case for our msp430 (see later on)
- ▶ Condition can rely on the value of registers, given as parameter to the instruction:
 - ▶ `beq r1, r2, label` = “branch to *label* if $r1 == r2$ ”
 - ▶ `bge r1, r2, label` = “branch to *label* if $r1 \geq r2$ ”
 - ▶ eg risc-v processor

The micro-machine - Jump Instructions

Unconditional absolute branch

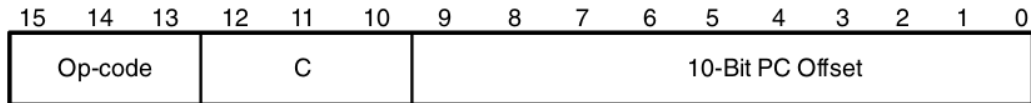
JA 42

continues execution at address 42.

Conditional relative branch

JR offset	JR offset IFZ <i>(executed if Z=1)</i>
JR offset IFC <i>(executed if C=1)</i>	JR offset IFN <i>(executed if N=1)</i>

Jumps on msp430



Mnemonic	S-Reg, D-Reg	Operation
JEQ/JZ	Label	Jump to label if zero bit is set
JNE/JNZ	Label	Jump to label if zero bit is reset
JC	Label	Jump to label if carry bit is set
JNC	Label	Jump to label if carry bit is reset
JN	Label	Jump to label if negative bit is set
JGE	Label	Jump to label if $(N \text{ .XOR. } V) = 0$
JL	Label	Jump to label if $(N \text{ .XOR. } V) = 1$
JMP	Label	Jump to label unconditionally

Control structures: if-then-else

if(*condition*){

*Ins*₁
...
*Ins*_{*n*}

}else{

*Ins*_{*k*}
...
*Ins*_{*m*}

}

Control structures: if-then-else

if(*condition*){

*Ins*₁
...
*Ins*_{*n*}

}else{

*Ins*_{*k*}
...
*Ins*_{*m*}

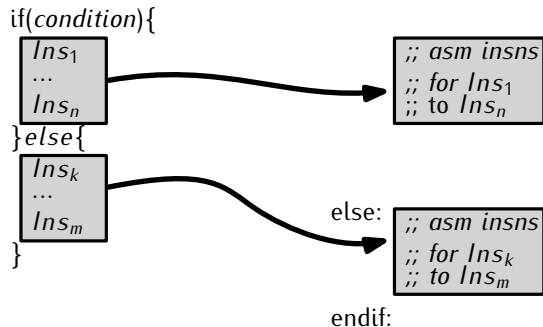
}

First, associate to each
"jump locations" a label

else:

endif:

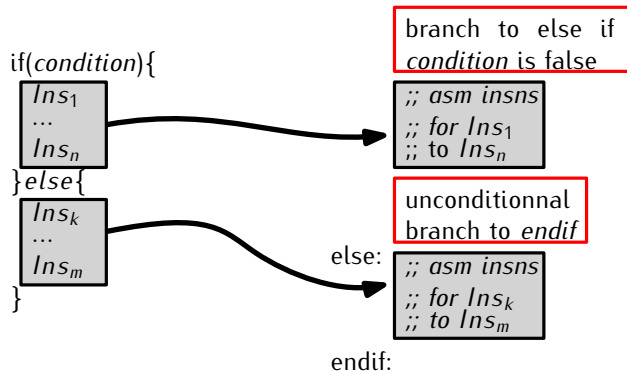
Control structures: if-then-else



First, associate to each "jump locations" a label

Then generate the asm code for each of the "then" and "else" block.

Control structures: if-then-else




First, associate to each "jump locations" a label

Then generate the asm code for each of the "then" and "else" block.

Finally, add jump instructions to evaluate the condition of the IF and decide which region of code the CPU should jump to.

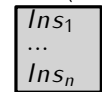
Control Structures: while

Proceed the same way for while loops....

```
while(condition){  
    
}
```


Control Structures: while

```
while(condition) {
```



```
  Ins1  
  ...  
  Insn
```

```
}
```

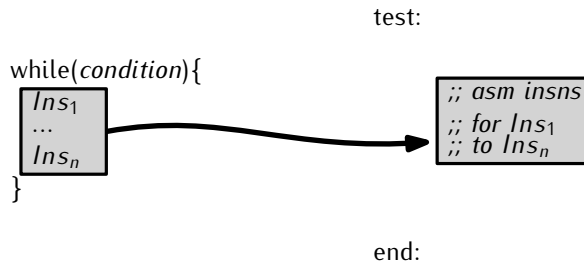
test:

end:

Proceed the same way for while loops....

First, associate a label to each "jump locations" (the test of the condition and the region "after the loop is finished").

Control Structures: while

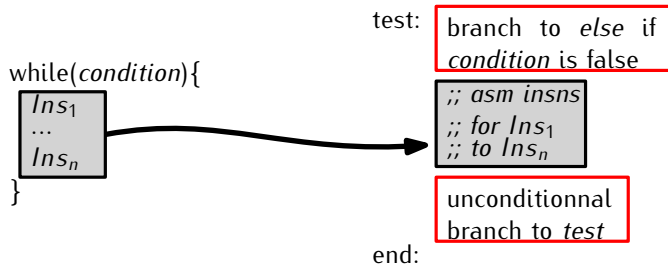


Proceed the same way for while loops....

First, associate a label to each "jump locations" (the test of the condition and the region "after the loop is finished").

Then generate the asm code for the body of the loop.

Control Structures: while



Note, that at the end of the loop body, you should always go back to evaluate the condition to decide if you should end the loop...

Proceed the same way for while loops....

First, associate a label to each "jump locations" (the test of the condition and the region "after the loop is finished").

Then generate the asm code for the body of the loop.

Finally, add jump instructions to evaluate the condition of the WHILE and decide which region of code the CPU should jump to.

Next week...

... we'll look at the internals of the Micro-machine and talk about the Von Neumann Cycle:

