50.002 COMPUTATIONAL STRUCTURES

INFORMATION SYSTEMS TECHNOLOGY AND DESIGN

Designing an Instruction Set

Natalie Agus (Fall 2018)

1 The Von Neumann Model

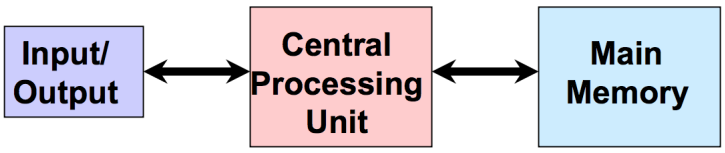


Figure 1

The von Neumann architecture is shown above, which is the structure of general purpose computer nowadays and it is said that it was inspired by the concept of the Universal Turing Machine. It has **expandable storage** (memory), just as what the infinite tape of the Turing Machine signifies.
The four components of the model are:

1. Central Processing Unit (CPU): containing several registers, as well as logic for performing a specified set of operations on their contents.
2. Memory: storage of N words of W bits each, where W is a fixed architectural parameter, and N can be expanded to meet needs.
3. Input/ Output: Devices for communicating with the outside world.
4. Connection bus that connects all the three components together.

Meanwhile, the anatomy of the CPU is shown below:

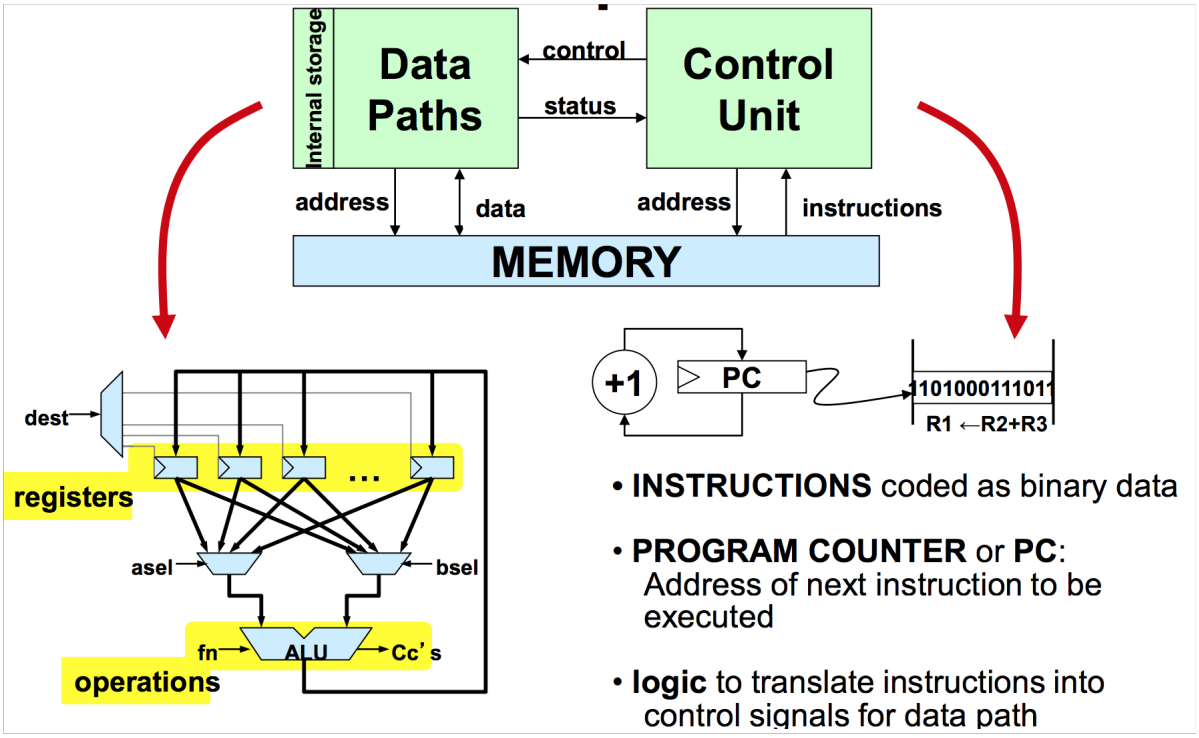


Figure 2

2 Programmability

Here's the list of the parts of von Neumann computer broken down that shows how this computer is programmable:

1. A memory (single, and expandable resource pool) doesn't only store data but also **stores instructions** that make up a program. Then, we need an interpreter called the CPU to fetch, execute, and interprets instructions from the memory.

2. The CPU is consisted of data paths and control unit (in green, Figure 1). They are sort of the brain of the system

3. Control unit supplies all the direction and figure out what to do

4. Data Paths manipulate data values and load / unload data from the memory to the registers (flip flop)

5. The Data Path is made up of 32 registers, see bottom left of the figure (temporarily hold data values), and muxes to decide what operation must be done on the ALU (arithmetic logic unit)

6. The Control Unit is made of the program counter, which tells us which line of the memory to read instructions from, and logic to translate instructions into control signals for data path

3 The β Architecture

This β architecture is created by MIT to explain the concept of von Neumann computers that are used nowadays. The Figure 3 in the next page is the complete architecture. Over the course, you will understand more of the entire workings of the architecture, so do not worry if you do not understand it yet.
Below are the important details of each component:

1. See Figure 2
2. **The Main Memory**: It can read and write a 32 bit data per operation (one row). The 32 bit data is also called as **word**.
3. **The Registers**: There are 32 (R1 to R31), each of 32-bit registers. The last register (R31) always has a value of 0 (in 32 bits).
4. **The Processor State (PC)**: An address of 32 bit **but by convention, the last two least significant bits is always 00 for a PC**. The PC determines which line of the memory it reads
5. About byte: 1 byte is 8 bits, 1 word has 4 bytes. So to increment a PC from one row of the main memory to the next row, we write : PC = PC + 4, because by convention some machine uses byte memory address. So to go to the next row of words in the main memory, we practically increase the PC by 4 bytes.
6. Note that Von Neumann Architecture is a concept, and the β architecture is an **example** of it. There are also Von Neumann computers that are different from β, for example, 64-bit computers nowadays.

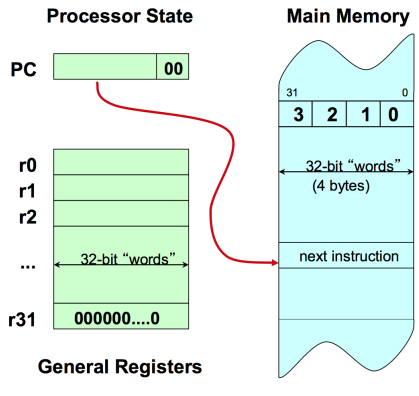


Figure 3

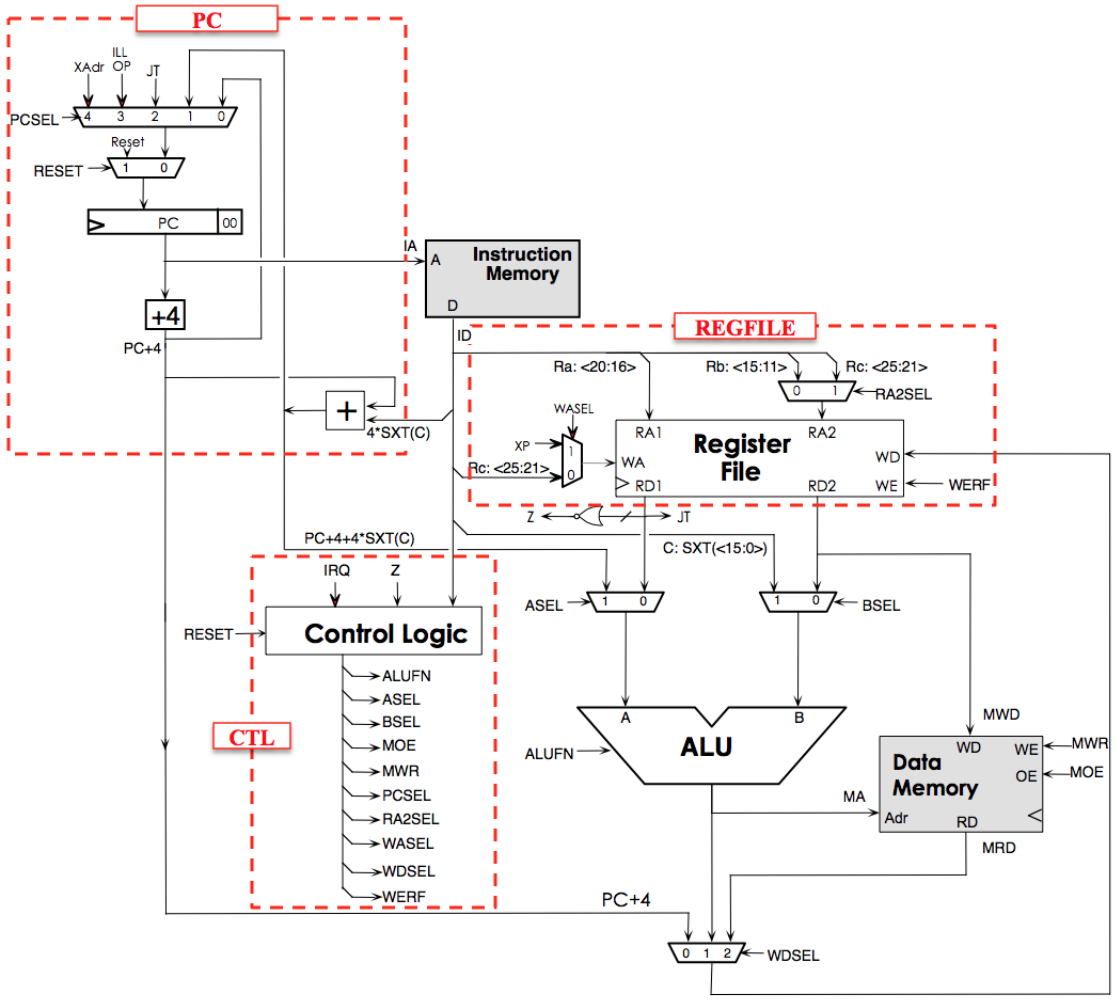


Figure 4

4 Beta Instruction Formats

The β instruction is 32-bits long, and its purpose is to tell the computer what task that has to be done, for example, to ADD, COMPARE, or to do boolean operations such as AND, OR, XOR, etc. Our high-level languages, like Java, C/C++, Python, etc will all be translated into this binary formats because the machine can only understand binary level languages (called machine languages). In the next note, we will learn how these high-level languages is transformed into this 32-bit β machine language. For now, lets take a look on the 32-bit β instruction format.

The figure below shows two types of β instruction formats:

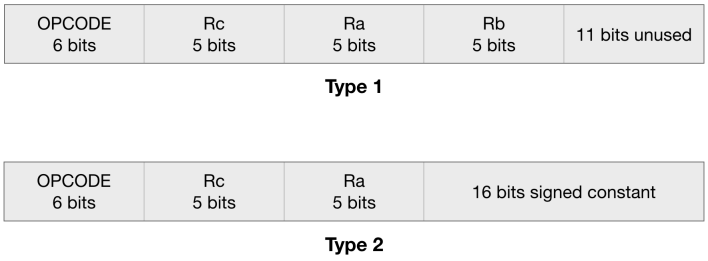


Figure 5

The meaning of each section is as follows:

1. OPCODE is 6 bits, they each signify different types of operation like: ADD (100000), AND (101000), MUL(100010), JMP (011011), etc (see β instruction set in lab handout)
2. For type 1: r_c, r_a, r_b are each 5 bits, to signify each of the 32 registers (from register 0 : 00000 to register 31: 11111) to get input from (a and b) or write output to (c). The last 11 bits are unused, it doesnt mean anything
3. For type 2: r_c is the destination register to write output to. r_a is the source data. The reason for this type 2 is that some operations require a constant like for example you want to add the content of register r_a with a constant c. Then the last 16-bit is the value of that constant c.

Example:

1. Lets say we have this instruction: ADD(r1, -3, r3). How to write this instruction in 32-bit machine language format? [1],
2. This means to add the content of register r_1 with a constant (-3) and put it into r_3.
3. From the β instruction set (next page), we see that:
   (a) The OPCODE for add is 110000.
   (b) The first register r1 in terms of 5 bits is 00001,
   (c) The third register r3 in terms of 5 bits 00011.
   (d) Encoding -3 as 16 signed bit is 1111111111111101.
4. Therefore the complete 32-bit instruction set is is:

110000 00011 00001 1111111111111101

[1]This convention is what we write as a code to program the β, which is just a program instruction that we load to the memory. See Beta lab notes on this. The point is to see how the machine translates this code thats understandable by us into the 32 bit instructions for the CPU to process.

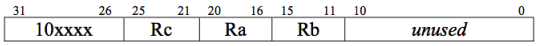
5 β Instruction Set Summary

Once you know how to read the 32-bit instructions, you need to familiarize yourself with ALL the instruction set summarized below. **Make sure you know what each instruction means**. See software labs: Beta documentation for more details on each instruction in the set.

Summary of β Instruction Formats

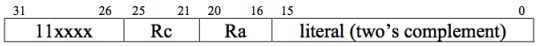
Operate Class:

Register	Symbol	Usage
R31	R31	Always zero
R30	XP	Exception pointer
R29	SP	Stack pointer
R28	LP	Linkage pointer
R27	BP	Base of frame pointer



OP(Ra,Rb,Rc): Reg[Rc] ← Reg[Ra] op Reg[Rb]

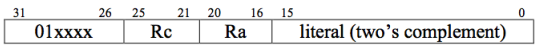
Opcodes: **ADD** (plus), **SUB** (minus), **MUL** (multiply), **DIV** (divided by)
AND (bitwise and), **OR** (bitwise or), **XOR** (bitwise exclusive or)
CMPEQ (equal), **CMPLT** (less than), **CMPLE** (less than or equal) [result = 1 if true, 0 if false]
SHL (left shift), **SHR** (right shift w/o sign extension), **SRA** (right shift w/ sign extension)



OPC(Ra,literal,Rc): Reg[Rc] ← Reg[Ra] op SEXT(literal)

Opcodes: **ADDC** (plus), **SUBC** (minus), **MULC** (multiply), **DIVC** (divided by)
ANDC (bitwise and), **ORC** (bitwise or), **XORC** (bitwise exclusive or)
CMPEQC (equal), **CMPLTC** (less than), **CMPLEC** (less than or equal) [result = 1 if true, 0 if false]
SHLC (left shift), **SHRC** (right shift w/o sign extension), **SRAC** (right shift w/ sign extension)

Other:



LD(Ra,literal,Rc): Reg[Rc] ← Mem[Reg[Ra] + SEXT(literal)]
ST(Rc,literal,Ra): Mem[Reg[Ra] + SEXT(literal)] ← Reg[Rc]
JMP(Ra,Rc): Reg[Rc] ← PC + 4; PC ← Reg[Ra]
BEQ/BF(Ra,label,Rc): Reg[Rc] ← PC + 4; if Reg[Ra] = 0 then PC ← PC + 4 + 4*SEXT(literal)
BNE/BT(Ra,label,Rc): Reg[Rc] ← PC + 4; if Reg[Ra] ≠ 0 then PC ← PC + 4 + 4*SEXT(literal)
LDR(label,Rc): Reg[Rc] ← Mem[PC + 4 + 4*SEXT(literal)]

Opcode Table: (*optional opcodes)

2:0 / 5:3	*000*	*001*	*010*	*011*	*100*	*101*	*110*	*111*
000								
001								
010								
011	**LD**	**ST**		**JMP**		**BEQ**	**BNE**	**LDR**
100	**ADD**	**SUB**	**MUL***	**DIV***	**CMPEQ**	**CMPLT**	**CMPLE**	
101	**AND**	**OR**	**XOR**		**SHL**	**SHR**	**SRA**	
110	**ADDC**	**SUBC**	**MULC***	**DIVC***	**CMPEQC**	**CMPLTC**	**CMPLEC**	
111	**ANDC**	**ORC**	**XORC**		**SHLC**	**SHRC**	**SRAC**	

Figure 6

6 Assigning Write Address in Assembly Language

The '=' sign in assembly language means to assign the **write pointer address** for that instruction, examples:

`. = 0x00001000` : move write pointer (**not PC!**) to address 0x00001000
`LD(R1, 5, R2)` : this instruction is now written in address 0x00001000

`square = 0x00001020` : this means the label square is at address 0x00001020.

The first instruction of function `square` above which is `PUSH(LP)`, is now at address 0x00001020 and 0x00001024[2].

After the instructions are written into memory, the PC will execute from address 0x00000000 onwards. Typically, at address 0x00000000 lies the BR(RESET address) that will direct the PC to another memory location containing the RESET instruction. This is the **entry** code to start up your computer.

[2]recall that PUSH is two instructions