50.002 Computational Structures

Information Systems Technology and Design

Machine Language and Compilers

Natalie Agus (Fall 2018)

1 β Instructions

As a recap, the following shows the two types of β instruction:

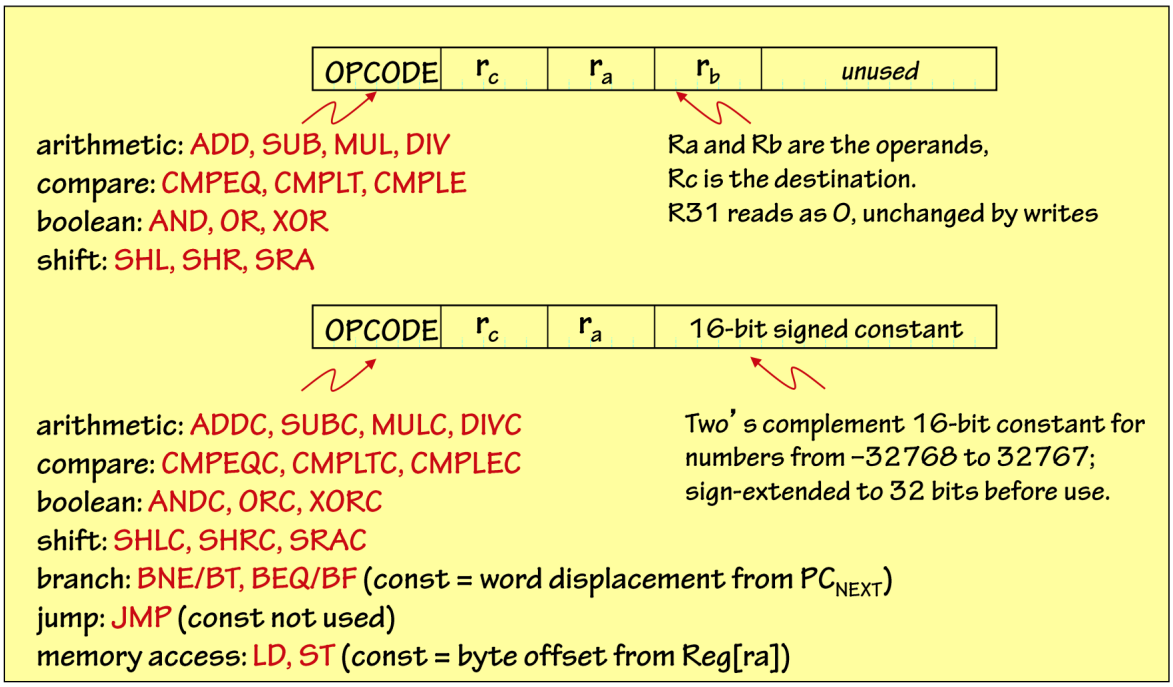

Figure 1

2 Improving β Programmability

Nobody wants to write 32 bits instructions all the time. Let's say we want to do a particular operation like ADD, then would prefer to write it as:

1. ADD(R2, R3, R4) - Called assembler language, a low-level assembler understands this and translates it to binary machine code (that 32-bit β instruction)

2. Or even better, the high-level language thats understandable by humans: R2 = R3+R4 - requires interpreter or compiler for the machine to run

3 Interpretation, Compilation, and Assembly

In the past, people keyed in the assembly language manually (key in 0s and 1s), however nowadays life is made easier with high-level language instructions. To allow high-level language instructions, we have two **different** methods to translate them to something that can directly command the hardwares - that 32-bit instruction. Assume we have this hard-to-use machine M1 and we want to translate our high level code into something thats understandable by M1. We can use either a compiler, or an interpreter.

Compiler:

1. A compiler translates from easy-to-use (high-level) language into a hard-to-use (low-level) assembler machine language thats suitable for a particular hardware

2. Its done before execution

3. It slows program development

4. Decisions are made during compile time, **before execution**

5. During Runtime, only refer to the new low-level program in assembly language, no more reference to the high-level program

6. Illustrated as Figure 2 below. A compiler translates high-level program P2 to low-level program P1 that's suitable for the machine M1

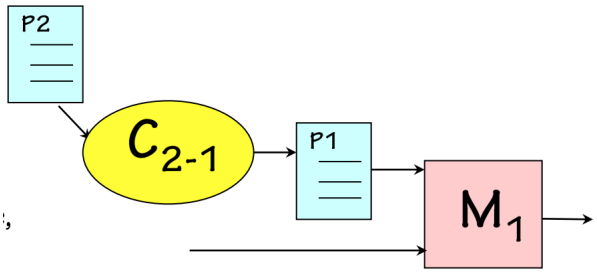

Figure 2

Interpreter:

1. Actually computes the exact instructions
2. Its done after execution
3. It slows program execution
4. Decisions are made during run time, **after execution**
5. Illustrated as Figure 3 below. Let's say M1 is hard to program machine. An interpreter writes a single program for M1, called Pgm, which mimics the behavior of another easier "virtual" machine M2 .

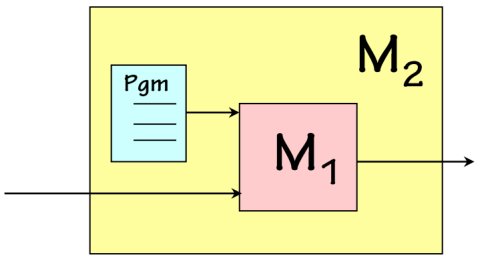

Figure 3

Assembler: (can also be seen as a primitive compiler) As the name suggests, the assembler takes the assembler code outputted by compiler or interpreter and translate it into binary machine code.

Note that a compiler is NOT the same as an interpreter and they are DIFFERENT methods that serve the same purpose: translates from high level language to machine binary low level language. Sometimes, an assembler isn't required by the compiler or interpreter IF they generate machine code directly. The table below might help you understand further the differences between the two:

Interpreter	Compiler
Translates program one statement at a time.	Scans the entire program and translates it as a whole into machine code.
It takes less amount of time to analyze the source code but the overall execution time is slower.	It takes large amount of time to analyze the source code but the overall execution time is comparatively faster.
No intermediate object code is generated, hence are memory efficient.	Generates intermediate object code which further requires linking, hence requires more memory.
Continues translating the program until the first error is met, in which case it stops. Hence debugging is easy.	It generates the error message only after scanning the whole program. Hence debugging is comparatively hard.
Programming language like Python, Ruby use interpreters.	Programming language like C, C++ use compilers.

Figure 4

4 UASM: A Program for Writing Programs

UASM is an assembly language (for the assembler), that makes life a little bit easier to execute instructions. Take a look at the figure 5 below, From a simple assembly language such as ADDC, an assembler unrolls it into 32-bit machine language (expressed as hex), which is the expression at the end of the figure. ADDC is actually **a macro** [1]to **betaopc** operation.

You just need to know how to convert from the β assembly language instruction like ADDC, JMP, etc directly into the 32-bit β instruction format.

[1]macros are parameterized abbreviations, or shorthands to simplify longer codes, think of it as similar to variable naming

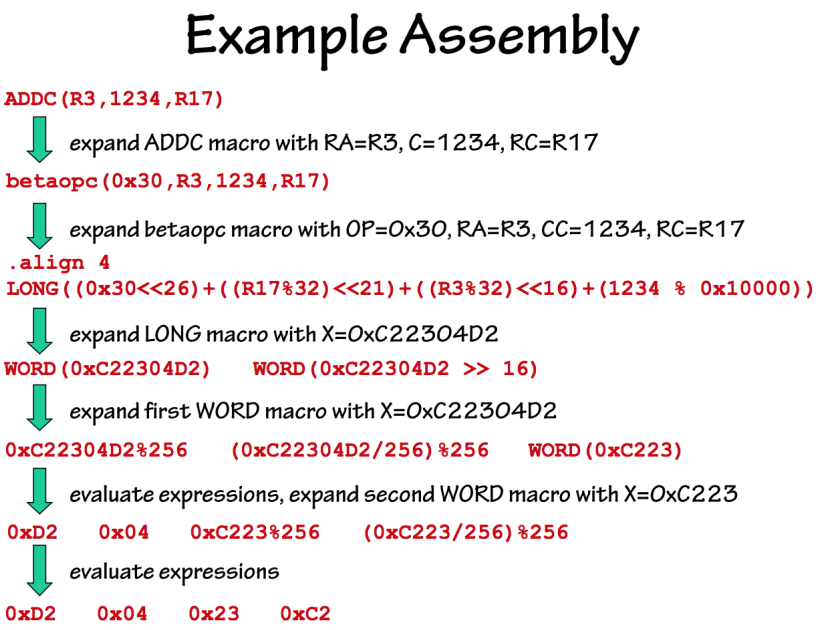

Figure 5

5 Addressing

In β architecture, we have 32 bits of content and also 32 bits of address, and the memory can be thought of like a table as follows:

Important points:

1. A memory device has **content** as well as **address**, BOTH are 32 bits long
2. Each line of a 32-bit content (each row) house 4 bytes of address (think of this like a pigeonhole)
3. So **in non-Beta architecture** if you're told to go to 'pigeonhole number 9', you will retrieve that chunk of 8-bit (a byte) data in orange. An address is 32-bit in length by convention, so 'pigeonhole number 9' is translated into 0x00000009
4. The address of **the entire of each row**, by convention, is the **smallest byte index** of that row, converted into 32 bits. That's why for example, although the third row contains has an address 'pigeonholes number 8 to 11', *the address of that row* is '8', and when it is transformed into 32 bits it becomes : 0x0000 0008 as address.
5. **In Beta architecture**, you can only read or write data in 32-bits in one shot. So there's no accessing individual 'pigeonhole' bytes. You can only access the row address, for example, access row address 0x0000 0004 means to retrieve

Memory

Address (32 bits in Hex)	Content (32 bits)				Remarks
0x0000000	*byte 3* 0010 0100	*byte 2* 0000 1000	*byte 1* 1010 1000	*byte 0* 0000 0000	The addressing is like the index of each byte "cell". In this row, we store byte 0 to byte 3. **Note how the address of an entire row (word) is the byte index of the LSB cell**
0x00000004	*byte 7* 0110 0100	*byte 6* 0000 1000	*byte 5* 1010 1000	*byte 4* 0000 0000	Addressing of each row jumps by 4 bytes (=32 bits), in this row we can store byte number 4 to byte number 7
0x00000008	*byte 11* 1010 0100	*byte 10* 0100 1000	*byte 9* 1010 1000	*byte 8* 0000 0000	If i give you the address: 0009, that means the orange (8bits) chunk of data
0x0000000C	*byte 15* 0010 0100	*byte 14* 1111 1000	*byte 13* 1010 1000	*byte 12* 0000 0000	**In beta architecture, the data always comes in 32 bits (not bytes) So if i tell you to fetch from address: 0x0000000C, you fetch ALL 32 bits of data in this row, which is: 0010 0100 1111 1000 1010 1000 0000 0000**
...	...				

Figure 6

the entire data from byte 4 to byte 7: 0110 0100 0000 1000 1010 1000 0000 0000.

Some math about addressing to think about (skip this if you have no time to spare):

1. In week 1 we learn that 32-bit length of data can hold up to 2^{32} different information, of 32-bit length **each**.
2. That means we can have a table of up to 2^{32} rows to enumerate all possible 1's and 0's permutation of 32-bit length data.
3. However, a 32-bit address can only hold up to 2^{32} **byte indexes**.
4. Lets use a smaller example, with this 4-bits memory addressing for 32-bit content: As you can see, 4 bits can enumerate numbers from 0 to 15, so it can address up to 16 byte cells. since the content for each row is 32-bit, this memory can only hold up to 4 rows (words) worth of 32-bit content.
5. Hence, a 32-bit address is NOT enough to house all 2^{32} permutations of 32-bit of information per row.
6. In fact, 32-bit address can only house $2^{32}/4$ of the possible 2^{32} permutations
7. 2^{32} permutations of 32-bits length of information means that you will have $2^{32} * 4$ bytes. That means you need $2^{32} * 4$ address bits.

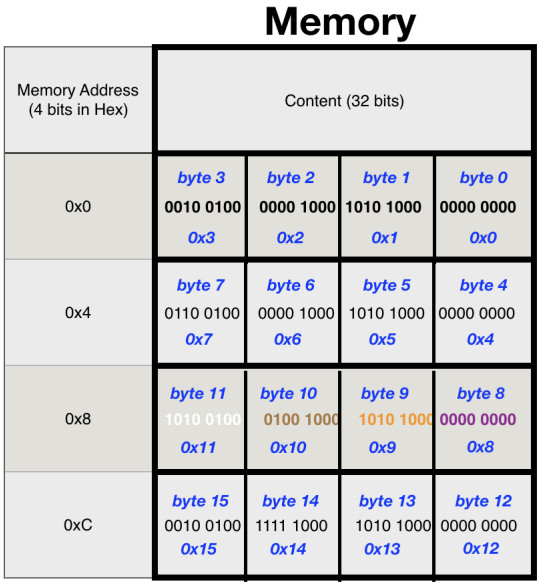

Figure 7

So in MIT slides, you see this. Lets digest abit what the stuff in the MAIN MEMORY means corresponding to the instructions in red:

```
---- MAIN MEMORY ----      . = 0x1000
  1000: 09 04 01 00        sqrs: 0 1 4 9
  1004: 31 24 19 10              16 25 36 49
  1008: 79 64 51 40              64 81 100 121
  100c: E1 C4 A9 90              144 169 196 225
  1010: 00 00 00 10        slen: LONG(. - sqrs)
```

Figure 8

Explanation:

1. Lets look at the MAIN MEMORY, first row. 1000 is the address, and 09 04 01 00 is the content in HEX (so total is 32 bits).

2. Again, since addresses is expressed in terms of bytes by convention, this is why the addresses jump by 4 bytes from consecutive rows (32 bit = 4 bytes)

3. . means the *write to memory pointer* (not PC!). The first code in red (. = 0x1000) means that it directs the pointer . to an address 0x1000 and as a consequence, 0 1 4 9 etc is stored from address 0x1000 onwards as shown.

4. Then we write all the squares values from LSB to MSB, thats why 0 is on the rightmost and 09 is on the leftmost of 0x1000 row (first row of the memory).

6 Summary

Apart from basic theories on compilers/interpreters/assembler, VN architecture, and turing machines, so far we have learned how to convert the β assembly instruction into 32-bit β machine instruction format. We also have learned how the addressing convention in done in the memory unit. In the next note, we will learn how these instructions are made by the compilers/interpreters/assembler from high-level codes, and then loaded into memory for the control unit to execute and pass through to sequential logic (made up of registers and combinational logic unit, ALU) unit - called *data path* in the CPU of the VN architecture (see previous note).