Lecture 4, CPU Operation

We now look at instructions in memory, how they got there and how they execute:

1. Start by using an editor to enter compiler language statements. The editor writes your source code to a disk file.

2. A compiler reads the source code disk file and produces assembly language instructions for a specific ISA that will perform your compiler language statements. The assembly language is written to a disk file.

3. An assembler reads the assembly language disk file and produces a relocatable binary version of your program and writes it to a disk file. This may be a main program or just a function or subroutine. Typical file name extension is .o or .obj

4. A linkage editor or binder or loader combines the relocatable binary files into an executable file. Addresses are relocated and typically all instructions are put sequentially in a code segment, all constant data in another segment, variables and arrays in another segment and possibly making other segments. The addresses in all executable files for a specific computer start at the same address. These are virtual addresses and the operating system will place the segments into RAM at other real memory addresses. Windows file extension .exe

5. A program is executed by having the operating system load the executable file into RAM and set the program counter to the address of the first instruction that is to be executed in the program. All programs might have the same starting address, yet the operating system has set up the TLB to translate the virtual instruction and data addresses to physical memory addresses. The physical addresses are not available to the program or to a debugger. This is part of the security an operating system provides to prevent one persons program from affecting another persons program.

A simple example:

```
  Compiler input        int a, b=4, c=7;
```

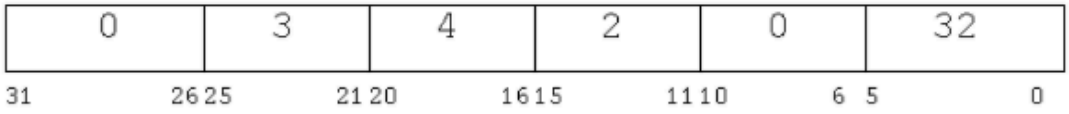
```
                        a = b + c;

  Assembly language fragment (not unique)
          lw     $2,12($fp)    b at 12 offset from frame
pointer
        lw       $3,16($fp)    c at 16 offset from frame
pointer
        add      $2,$2,$3      R format instruction
        sw       $2,8($fp)     a at 8  offset from frame
pointer

  Memory addresses in bytes, integer typically 4 bytes, 32 bits.

  Loaded in machine
    virtual address   content 32-bits  8-hexadecimal digits

    00000000         8FC2000C  lw $2,12($fp)
    00000004         8FC30010  lw $3,16($fp)
    00000008         00000000  nop inserted for pipeline
    0000000C         00431020  add $2,$2,$3
    00000010         AFC20008  sw  $2,8,($fp)

    $fp has 10000000  (data frame)
    10000000          00000000
    10000004          00000001
    10000008          00000000?  a  after execution
    1000000C          00000004   b
    10000010          00000007   c


  Instruction field format for  add $2,$2,$3
    0000 0000 0100 0011 0001 0000 0010 0000  binary for 00431020
hex

    vvvv vvss ssst tttt dddd dhhh hhvv vvvv  6,5,5,5,5,6 bit
fields
       0   |  2  |   3  |  2  | 0   |  32    decimal values of
fields


  Instruction field format for  lw $2,12($fp)     $fp is
register 30
    1000 1111 1100 0010 0000 0000 0000 1100  binary for 8FC2000C
```

hex

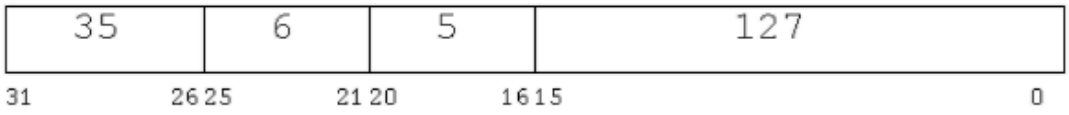
```
    vvvv vvxx xxxd dddd aaaa aaaa aaaa aaaa  6,5,5,16 bit fields
      35   | 30  |   2  |        12          decimal values of
fields
```

The person writing the assembler chose the format of an assembly language line. The person designing the ISA chose the format of the instruction. Why would you expect them to be in the same order?

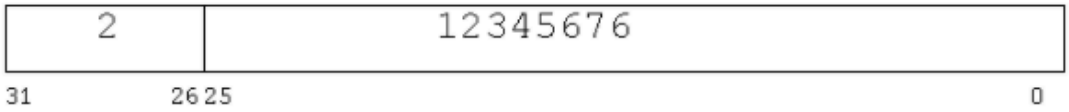
ADD $2,$3,$4 00641020_{16}



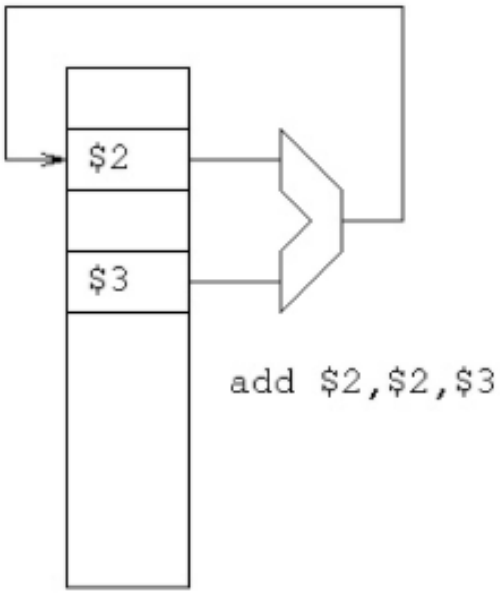
LW $5,127($6) $8CC5007F_{16}$



J 12345676 $082F1853_{16}$



A very simplified data flow of the add instruction. From the registers to the ALU and back to the registers.



The VHDL to use the ALU will be given to you as:

```
  ALU: entity WORK.alu_32 port map(inA    => EX_A,
                                   inB    => EX_aluB,
                                   inst   => EX_IR,
                                   result => EX_result);
```

We will call the upper input "A" and the lower input "B" and the output "result". The extra input, EX_IR, not shown on the diagram above is the instruction the ALU is to perform, add, sub, etc.

The instructions we will use in this course are specifically:

cs411_opcodes.txt

Each student needs to understand what the instructions are and the use of each field in each instruction. (Note: a few have bit patterns different from the book and different from previous semesters in order to prevent copying.)

Our MIPS architecture computer uses five clocks to execute a load word instruction.

```
 1 0 0 0 1 1 x x x x x r r r r r ---2's complement address------
lw r,adr(x)

  1. Fetch the instruction from memory
  2. Decode the instruction and read the value of register xxxxx
  3. Compute the memory address by adding the sign extended
bottom
     16 bits of the instruction to the contents of register
xxxxx.
  4. Fetch the data word from the memory address.
  5. Write the data word from memory into the register rrrrr.
```

When we cover "pipelining" you will see why five clocks are used for every instruction, even though some instructions need less than five.

Computer languages come in many varieties. The information above applies to languages such as C, C++, Fortran, Ada and others.

Many languages abstract the concept of binary relocatable code, in what was originally called "crunch code". These languages use their own form of intermediate files. For example Pascal, Java, Python and others.

Other languages directly interpret the users source files, possibly with some preprocessing. For example SML, Haskel, Lisp, MatLab, Mathematica and others.

With a completely new computer architecture, the first "language" would be an assembly language. From this, a primitive operating system would be built. Then, typically an existing C compiler would be modified for the new computer architecture. An alternative is to build a cross compiler from C and gas, to bootstrap existing code to the new architecture. From then on, "reuse" goes into full effect and millions of lines of existing software can be running on the new computer architecture.

For Homework 3
  The computer irix.gl.umbc.edu is no longer available.
  This was a MIPS architecture using the same instructions

as we are using. The MIPS architecture is studied because it is a much simpler and easier to understand architecture than the Intel X86, IA-32.

Thus, to see the instructions in RAM, we will use the gdb debugger on an Intel X86.

HW3 information

The information in hex.out will have lines similar to:

```
(gdb) disassemble
Dump of assembler code for function main:

 RAM addr    offset    op code  address and register
0x08048384 <main+0>:     lea    0x4(%esp),%ecx
0x08048388 <main+4>:     and    $0xfffffff0,%esp
0x0804838b <main+7>:     pushl  0xfffffffc(%ecx)

End of assembler dump.
(gdb) x/60x main
                     Note: 16 bytes per line, 4  32-bit words
                     but, these are X86 instructions, not MIPS !
0x8048384 <main>:    0x04244c8d 0xfff0e483 0x8955fc71 0x535657e5
0x8048394 <main+16>: 0x58ec8351 0x4589e089 0xe445c7cc 0x00000064
                           ##                                ##
                <main+19>----|                  <main+31>---|
                0x8048397                       0x80483A3
```

Because the MIPS architecture we are studying is a big endian machine, we will count bytes from left to right for homework 3.

```
In hexadecimal, 0x12345678 is stored big end first     12
                                                       34
                                                       56
                                                       78

Little endian   0x12345678 is stored little end first  78
                                                       56
                                                       34
Each byte, 8 bits, is two hex digits                   12
```