CSE 3430

Class 5 and 6

Slide set A-3

**4 Types of CPU Instructions (What CPUs can do)**

Even the most costly, complicated processors can only do:

Data Movement

Program Sequencing and Control

ALU Operations (Arithmetic and Logical Operations)

Input/Output Transfers

**Memory Operations – Data Transfers: Only 2 Types**

**Address (meory array index) used to specify location of data:**

Read: Never changes data read.

Write: Always changes data written (writes will overwrite prior value)

**Processor must keep track of next instruction**

Special register: **PC** (Program Counter) holds address of next instruction.

Address is incremented/changed after address sent to memory to read instruction.

Register also to hold bit string for instruction to be executed: **IR** (Instruction Register)

**Assembly Language**

A text-based (so, human readable) form of **machine language** (Bit strings for instructions the processor can execute). Machine language is the only language the CPU understands or can execute.

**Assembler:** Program to convert assembly language to machine language.

Each instruction in assembly language corresponds to a single instruction in machine language.

Statements in high-level languages (Java, C, C++, Python, etc.) sometimes correspond to a single machine language instruction but usually (almost always) correspond to sequence of a number of machine language instructions.

People rarely write assembly code now (except in security, or for a few other purposes), but it’s useful to understand the basics.

**Processor Families: RISC and CISC**

RISC: Reduced Instruction Set Computer Processors - Key Characteristics

1. Smaller number of instructions; 2) All instructions are one word; 3) ALU operations can only be done on data in registers; 4) Use less power compared to CISC

CISC: Complex Instruction Set Computer Processors - Key Characteristics

1. Much larger number of instructions; 2) Instructions are various lengths (not all one word long); 3) ALU operations can be done on data in memory or in registers; 4) Use more power compared to RISC

For RISC programs, since ALU operations can only be done on data in registers, data must be moved from memory to register (**load**), arithmetic/logical operation performed on data, and result moved back to memory (**store**). This way of writing code typically makes programs longer.

RISC processors are called **load/store architectures**, because the only instructions that dan access data in memory are load and store.

**NO if, else, while, for etc. In Assembly Language!**

**Branch/jump instructions** are used for all of these

-See example in slides (Slide 13)

Branch instructions (also called Jump instructions) allow the CPU to change from sequential execution.

**Sequential execution:** Execution of instructions one after another, in order, in memory (1st instruction, 2nd, 3rd, etc.).

* This is what processors always do unless an instruction is executed which allows for non-sequential execution. There are several types of instructions which allow non-sequential execution; Branch/Jump instructions are one of these types.
* **There are two types of branch/jump instructions:**
* **Conditional branch/jump** - This causes the processor to check some condition:
  + If the condition is TRUE, it will not execute the following instruction in memory next, but instead, will execute the instruction at the address of a LABEL next (WE say that the processor branches or jumps to the LABEL, which is just a marker for an address of some instruction in the program). The BLZ instruction (Branch Less than Zero) on slide 13 is an example of this type of conditional branch instruction: if the last result (the result of the SUB (subtraction) instruction was LESS THAN ZERO, the instruction will branch to the label B1 and execute the instruction there next (STORE R2, C).
  + If the condition is FALSE, the processor will not branch to the address of the LABEL; instead, it will execute the following instruction in memory next (in other words, sequential execution). That is, in this case it will execute STORE R1, C next.
* **Unconditonal branch/jump:** This type of branch/jump will ALWAYS branch/jump to the address of the LABEL, no matter what (no conditions). The BR (Branch) instruction on slide 13 is an example of this; this will go to the address of the LABEL B2 and the instruction at that address will be executed next.
* ALL processors have both of these types of branch/jump instructions.
* How can the processor either follow sequential execution (what usually happens in programs) or change from sequential execution? This is determined by what address is written to the PC (Program Counter), which always has the address of the next instruction to be executed.
  + Suppose we have a RISC processor (What we will generally assume for our discussions in the rest of this course). Remember, we said this means all instructions are the same length (one word long); for a 64-bit RISC processor, all instructions are 8-bytes long.
  + For sequential execution, the processor increments the PC register after it sends the address of an instruction to the meory to read the instruction; for 64-bit RISC, it increments by 8-bytes. For example, if the CPU is executing an instruction at address 1000, after the address 1000 is sent to the memory, the PC will be incremented by 8-bytes to 1008 to get the address of the next instruction.
  + For Branch/Jump instructions, to branch/jump to the address of the label, the CPU will overwrite the PC (which was already incremented) with the address of the label in the instruction. This will cause the next instruction to be executed to be the one at the address of the label.
  + CONDITIONAL BRANCH/JUMP instructions check whether the condition for the instruction is true or not by reading one or more **1-bit flags** in the **PSR (Processor Status Register)**. The PSR has a number of flags, but we will only consider four of them. We already saw two, the C (Carry) flag and the O (Overflow) flag.
  + CONDITIONAL BRANCH/JUMP instructions read the S (Sign) flag and the Z (Zero) flag. See the description on slide 16 of slide set A-3 to see how these flags are set by the processor to store information about the result of the last (most recent) ALU instruction, such as ADD, SUB, MUL, DIV, AND, OR, XOR, etc. (more on these instrucions later).

**PSR FLAGS**

* **CF** (covered before)
* **OF** (also covered before)
* **SF (Sign Flag)**: Only affected by ALU (Arithmetic and Logical) Instruction. SF is set to 1 if result is Negative, or set to 0 if not.
* **ZF** (Zero Flag): Only affected by ALU (Arithmetic and Logical) Instruction. ZF is set to 1 if result is 0, or set to 0 if not zero.

**THERE ARE NO LOOPS IN ASSEMBLY/MACHINE LANGUAGE EITHER!** BRANCH/JUMP instructions are used to implement these kinds of control flow in programs also!

* See the example on slide 17.
* ADDRESSING MODES:
  + Operands with # in instructions are CONSTANTS. These are always stored as part of an instruction (the Assembler converts them to bits which are part of the instruction bit string). One type is # with a numeric value, such as #4 or #1; these are of course numeric constants. The other type is # with a LABEL, such as #Num; this is the ADDRESS of the LABEL.
  + DATA LABEL operands without #, such as N or SUM, will access (read or write) the value of the variable in memory at the LABEL.
  + A register name in parentheses, such as (R4), is used to read or write THE DATA AT THE ADDRESS IN THE REGISTER.

More on LABELS:

* + LABELS are just markers for addresses of data or instructions in memory.
  + LABELS are not part of the program (there are no LABELS in machine language; the Assembler converts any LABEL to AN ADDRESS in the machine code (bit string) version of the program which the processor executes.

**DATA STRUCTURES IN MACHINE/ASSEMBLY LANGUAGE**

* There are NO DATA STRUCTURES recognized by processors either! For example, no CPU has any idea what an array is!!!
* If the programmer wishes to use an array, linked-list, or any other data structure, the programmer must write code to put the data in memory, and then write statements (coverted to instructions by compiler and assembler) which will treat the data as that kind of data structure.

**SUBROUTINES IN ASSEMBLY/MACHINE LANGUAGE**

* SUBROUTINE is the term used in assembly/machine language for blocks of code which perform some useful task which is useful for different sets of data. Typically called Methods, Functions, or Procedures in High-Level languages, but these are all the same in machine language.
* We know we CALL subroutines in programs we write:
  + How is this different from a BRANCH/JUMP?
  + It is different because for a CALL, we know the program RETURNS to the point in the code where the call was made; BRANCH/JUMP instructions DO NOT RETURN, so they cannot be used for CALL. A separate kind of instruction which saves the point to return to must be used, and every processor has kind of instruction.
  + HOW TO SAVE RETURN ADDRESS? Two possibilities:
    - Save in a register (almost never used, because limits number of calls before return).
    - Save on a PROGRAM STACK: Systems we all use do it this way. THIS ONLY LIMITS NUMBER OF CALLS TO SIZE OF STACK (but its quite a lot of memory).
* CALLER and CALLEE SUBROUTIINES:

**THE STACK**

* Used for VARIOUS PURPOSES:
  + Used to save **RETURN ADDRESS** for call;
  + Used to **PASS PARAMETERS** to subroutines;
  + Used for memory space for subroutine to keep variables it needs to execute its code for its task (**LOCAL VARIABLES**).
* The processor has A REGISTER to hold the location of the TOP OF THE STACK (THE ADDRESS in memory where the top of the stack currently is).

**STACK FRAMES**

* Each subroutine gets PART OF THE STACK to sotre what it needs. This part is called its STACK FRAME.
* We will see a little more detail on how this works in Intel processors (which are typical) in the next few slide sets.