Lecture 11 - von Neumann Arch. Continued, the instruction cycle

The von Neumann computer

- historically, the first computers were fixed program computers
- Recall the machines I walked you through in the first lecture. Many of those had hard-wired programs
- Another way to put it is that they were not programmable
- This clearly raises issues. We’d like our machines to be truly general purpose, in that the operations they perform can change over time
- This led to the development of the stored program computer (or the von Neumann computer)
- In a von Neumann machine, programs (instructions for the computer) are stored in read/write memory along with data

Basic von Neumann architecture

- A von Neumann architecture consists of three main parts: memory that is read/write, Input/output devices (I/O devices), and a central processing unit (CPU) which has the job of actually executing instructions in the machine
- the CPU consists of two primary subunits: the control and the processing unit
- The simplest processing unit consists of an arithmetic logic unit (ALU), which performs calculations on data. Remember, we built a small ALU before!
- Which operations the ALU performs is determined by sequencing carried out by the control unit
- A processing unit will also contain a register file, which is just a clump of registers used for temporary storage which the ALU can operate on
- The ALU can get data from the control unit, and may send addresses to the control unit for branches or jumps (control flow)
- Memory you can think of as a storage array and two primary registers, the MDR (memory data reg.) and the MAR (memory addr reg)
- MAR is k-bits wide (address width) and MDR is >= j bits, where j is the addressability of the machine
- We also have I/O devices attached to the machine: e.g. keyboard, mouse, disk, video, printer, etc. They receive commands from the CPU. They can also send data and status back to the CPU. Device sends interrupt signals to the CPU to notify that it’s done something

Instruction Format, ISA

- Every instruction has an opcode to identify it and fields that specify operands and results. It also might have flags to differentiate it from close variants of instructions (e.g. read a character vs. read a string)
• What are some instructions? (Remember we built a very tiny calculator before…)
• We can choose two types of instruction encodings:
  o Fixed length: easier to design, instruction decoding logic much simpler
  o Variable-length: more efficient use of memory, more flexible instruction set, but
    more complicated hardware (and software)
• The set of instructions supported by the hardware, how they specify operands, their
  formats etc. is determined by the Instruction Set Architecture (ISA).
• The ISA is the interface to the hardware. Think of it as being the minimum amount of
  detail you need to program the machine.
• Well known ISAs include ARM, x86, x86_64 (x64), OpenSPARC, MIPS, etc.

Addressing Modes

• Conceptually, instructions are simple. But we’ve really only considered the operators.
  What about the operands? Where do they come from?
• Usually, registers, memory, or in the instruction itself! We need ways to specify
  which register and which memory location
• For this, we use addressing modes
• When an operand is part of the actual instruction, we have immediate operands
• For example ADDI R0, 1 Might be encoded as 0000 000 0001
• If the data is in a register, we need to be able to specify which, so we need bits for that
  in the instruction encoding
• What about memory? We could just put the memory address in the instruction. This is
  absolute addressing. The issue here is that our instructions must be big enough to
  hold the address and the opcode!
• Furthermore, we have to change the instruction itself if we want to manipulate the
  address (for example for an array)
• We could save some space by, e.g. putting the address in a register, and accessing
  relative to that register. This is called indirect addressing. Here we have a base
  address stored in a register, and e.g. a constant offset in another register
• There are more advanced addressing modes, some of which we will see

The instruction cycle

• A computer will run a program by executing a series of instructions in sequence.
• It goes: grab an instruction from memory, execute it; Grab another, execute it. Etc. Etc.
  It’s dumb!
• The instruction cycle is the series of steps involved with the execution of a single
  instruction. This can vary, especially with newer machines, but a common case is as
  follows:
  o Fetch Instruction from Memory
  o Decode instruction
  o Evaluate Operand address(es)
  o Fetch Operands (into registers)
Execute instruction
  - Store the output of the execution (into registers or memory)

- Note that not all instructions have all phases (e.g. a load instruction does not store output to registers or memory)
- **Recall the PC**: program counter is a special register which holds the address of the next instruction to execute

**Fetch**

- Read instruction into the IR (instruction register)
- PC is a memory address; get the thing at that address, copy those contents into the IR: Shorthand for this is IR <- Mem[PC]
- We then increment the program counter so that it points to the next instruction. This of course depends on the size of the instruction: PC <- PC + 4 if our instruction width is 4 bytes. This may not be true of we have a *control flow instruction*, e.g. a JUMP or a conditional instruction.

**Decode**

- Which instruction is it? Feed the opcode (and potentially other bits from the IR) to a decoder. Logically speaking, the decoder is going to select a row in a table in the control that will determine the rest of the instruction cycle
- Particular instructions might need more bits from the IR (e.g. if it has an immediate operands)

**Evaluate Addresses (get effective addresses of operands)**

- If the instruction has operands, we must figure out where each is. Is it in a register? a memory location?
- For example IR[7:4] could be a 3-bit register or PC + IR[15:7] could be a memory address.
- Not all instructions have operands, so not all instructions need this phase

**Fetch Operands**

- Go get the operands and bring them to temporary storage (registers). If they’re in memory, this would involve an access to the memory system. Again, only instructions **with** operands

**Execute**

- Actually run the instruction. There are generally three possibilities:
  - **Data movement**: load or store a value to/from memory
- **Math**: perform some operation on data (add, shift, div, mul, xor, etc.)
- **Control flow**: jump, branch, call (go somewhere else in the program). This is used for conditionals, loops, gotos, function calls. Note that these instructions necessarily modify PC!

**Store Results (Write back)**

- Take results from load or math ops and put them somewhere (register or memory)
- Only instructions that produce results have this phase

**Reality**

- Actual computers much more complicated for performance and power efficiency
- Many processing units (execution units such as ALUs or floating point, vector processing units)
- Many CPU cores
- I/O happens out of band with CPU (DMA)
- I/O processing offloaded from CPU to special-purpose units such as GPUs
- Work on more than one instruction at a time (pipelining, large instruction words) (LIW)
- Execute instructions out of order based on data dependencies
6-stage instruction cycle

1. FETCH "GET INSTR FROM"
   - FR ← MEM[PC]  [MEMORY]
   - PC ← PC+4  [PC = some ADDR]

2. DECODE INSTR

3. EVAL. OPS STAGE
   - MEM[PC + IR[3:7]]

4. FETCH OPS
   - MAR ← PC + IR[5:7]
   - MDR ← MEM[PC + IR[5:7]]

5. EXECUTE (ALU)
   - Data move
   - Math
   - Control flow

6. WRITE BACK
   "STORE RESULTS"

   Memory
   or registers populated
   w/ results
1. **ABSOLUTE (IMMEDIATE OPERAND)**

2. "LOAD"

```
<table>
<thead>
<tr>
<th>4-bit</th>
<th>3-bit</th>
<th>3-bit</th>
<th>1</th>
</tr>
</thead>
</table>
```

Op-code | reg | reg | output | address | reg

Before:

```
LOAD RO, 0xdeadbeef
```

```
LOAD RO, [R1]
```

```
RO ← mem[R1]
```

```
LOAD RO, [ER1]
```

```
char c[8] = "hiiiiiiii"
```

```
mt: 
R2
```

```
LOAD R2, 0x0
LOAD R1, 0x0
(LOAD RO, R2[R1])
ADD R2, 0x1
TUMP
```