the SAP-1 and introduction to instructions

cmpt-150-arc

Sections 7.1, 7.2, 9.1

1. Have read sections 7.1, 7.2 for background
2. Our text unfortunately goes hog-wild at this point and tries to teach everything that would be in a full assembly language course, PLUS what would be in a second-year hardware course. We'll pick and choose the relevant sections to read, and combined with my notes, should give you a fair grounding.

I. Intro

1. We now have enough knowledge of digital hardware to understand how a computer works.
2. At this point most texts try to do computer design, but for now we will be content to understand how a computer works.
3. From here on in we will be doing mostly assembly language work, but we need to do a bit of hardware work to understand what assembly language is.
4. We'll use a fairly common format, which is to design a really stupid simple computer, and then expand it to something useful. Our first approximation will be called "Simple-As-Possible-1"(SAP-1)

II. Basic Computer Architecture

1. There are some things that are common to a lot of computers (most, actually) and we'll start with a couple of these.

A. Organization

1. Computers are usually broken into two parts: Memory (as we have discussed before), and Processing, which is usually done in a block of hardware called a central processing unit (CPU)
2. Memory holds instructions and data, and the CPU reads instructions from memory and performs actions on the data.

B. Instruction Cycle

1. Every computer operates in what's called an instruction cycle, or fetch/execute cycle:
   a) Get an instruction from memory (fetch)
   b) perform the instruction (execute)

C. Instructions

1. So where do these instructions come from?
2. When you write code in a high level language (HLL), the code is translated into assembly language which is then translated into machine language,

which is composed of individual CPU instructions.

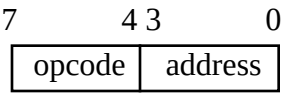
3. These instructions are strings of 1s and 0s, which computers read well but humans have a difficult time with.
4. It is possible but very difficult to write directly in machine language - the acres of 1s and 0s necessary to perform even a simple set of operations quickly humbles the best computer geeks.
5. The next level up is Assembly language, a direct translation of machine language into something moderately human-readable. It's what we'll learn for the rest of this course.
6. Note the difference: Machine language is 1s and 0s. Assembly language is a direct translation into something almost readable.

III. Datapath Design (SAP-1)

1. So we want some hardware that will fetch and execute instructions.

A. Instructions

1. Let's start by assuming what an instruction will look like in our Simple-As-Possible design: [Overhead]

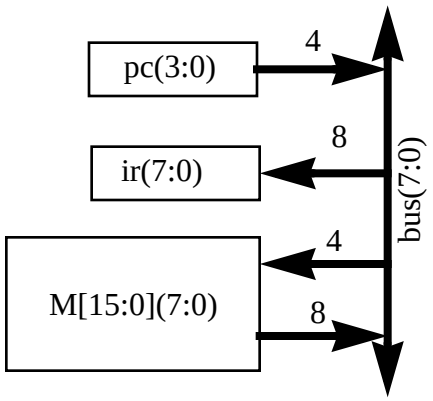


2. This shows how the bits of the instruction are interpreted. The first 4 bits (7-4) are called the opcode field, and the last 4 are called the address field.
3. The *opcode* field contains the opcode (duh) which identifies the instruction. There will be a unique opcode for each instruction the computer will be able to handle.
4. The *address* field contains the memory address of data to be operated on (if necessary)
5. Now we can start designing some hardware.
6. Our first machine will be able to execute: **lda** (load into accumulator); **add** (add memory to accumulator); **sub** (subtract memory from accumulator); **out** (display data); and **hlt** (stop execution) [[show instruction set]]

B. Fetch hardware

1. The first part of every instruction cycle is fetch. What do we need to fetch an instruction from memory?
2. We need memory, and we need a place to put the instruction just fetched.
3. How much memory? Our instructions are 8 bits wide (7:0), and the address portion is 4 bits meaning we can address 2^4=16 words [15:0], so our memory will be 16x8 RAM, called M[15:0](7:0).
4. Let's make a 8-bit register to hold the instruction, the *instruction regsiter(ir)*.

5. We also need to know where in the memory our instruction is to be fetched from. Let's build another register, called *program counter(pc)* which will hold the address of the instruction to be fetched.
   a) it's called a program counter because instructions are often stored in memory in order, so instruction 1 would be at M[1], instruction 2 at M[2] etc.
   b) In SAP-1, we will store our instructions in order, so the PC should be a counter that will increment when we tell it to, so it needs a control signal Cp. When Cp is high, pc=pc+1 at next clock pulse.
6. We also need a way for these three devices to talk to each other. We'll drop a bus for them all, and it needs to be 8 bits wide if it's to carry an instruction. Here's what it looks like so far:



7. Since there is more than one device trying to drive this bus, we need to put tri-state outputs on each, and so we need enable signals for each device trying to drive the bus.
8. Will that do it? almost.
   a) The PC asserts an address, and the memory responds, but it can't put the instruction on the bus because bus(3:0) is full!
   b) and if pc releases the bus, the memory won't have the address anymore!
9. Two options:
   a) Put a register to grab the instruction from memory and wait for the bus, or
   b) put a register to hold the address at the input to memory.
   c) We'll do the latter. It's called a Memory Address Register (*mar(3:0)*).
10. This is sufficient hardware to fetch instructions. (show slide: SAP-1 (a))
    a) *mar* needs a control signal to tell it when to load the address from the bus, call it L_M.
    b) *ir* needs a control signal to tell it when to load the instruction from the bus, call it L_I.
    c) *pc* and *M* need enables (E_p, CE) to let them individually drive the bus.

C. Execute hardware: lda

1. We'll need different hardware for each new instruction we add to the set.
2. lda (load accumulator) is a good place to start.

3. It should get an address from the *address* field of the instruction, access that memory address, and put whatever's there into the accumulator.
4. We need:
   a) a path from ir(4:0) to mar (to transfer the addr) and an enable for ir (E_I)
   b) An accumulator to store the data from the memory. This will be another register, 8 bits, with a control signal to load (L_A). The hardware: (show slide SAP-1(b).
5. The accumulator is a traditional name for the register used to keep a running tally of additions. Let's make our computer capable of adding!

D. Execute hardware: add

1. This instruction will add the contents of the accumulator to the contents of memory in the address, placing the sum back into the accumulator
2. What do we need?
   a) Something to add with - an 8-bit adder will do. It needs to be able to drive the bus with the result, so it needs an enable signal as well.
   b) We could take the second summand directly from the memory, but that would occupy the bus and we'd have nowhere for the sum to go
   c) same problem as with memory - we need to hold the inputs steady while the output is generated.
      (1) Do we put a register at the second input to the adder or at the output?
      (2) We'll put it at the second input, for regularity.
      (3) Thus we need a new register, called B register, at the second input to the adder. It needs to be able to load, so it needs a control signal L_B. (show figure SAP-1(c))

E. Execute hardware: sub and out

1. sub, out should be easy to add now.
   a) sub requires a subtractor instead of an adder, which we know how to make. We add a new control signal, SU, which we activate when we want to subtract.
   b) out requires a place to display data (8 LEDs for example), and a register to hold the data being displayed. The output register should have a control signal that allows it to load (L_O) (show SAP-1(d))
2. There's one block that we haven't looked at here - the control sequencer.
   a) This is a sequential circuit with 4 inputs (the opcode) and 13 outputs (the control signals).

   b) The sequencer is the brain - without it the datapath is dumb circuitry. The sequencer makes everything happen in the right order, and says when to fetch the next instruction.
   c) We'll look at what's inside it in a while
3. Last operation is *hlt*. We need no new hardware for it. It happens inside the sequencer, and stops the computer by not fetching any more instructions.

IV. Instruction Cycle in Detail

1. Let's look specifically at what should happen during the operation of each instruction.
2. We need to keep three principles in mind here. The first is a re-write of the design principle for counters and registers that we saw before
   a) To make a piece of hardware perform an action, you must set up the control signals so the proper action will occur at the ↑ edge of the clock.
   b) In a given clock cycle, there can be only one source of data for the bus, but any number of devices can use the bus as output.
   c) Control signals change once per clock cycle.
3. What this says is that we can perform as many operations in one clock cycle as we want as long as only one of them requires output to the bus.
4. Another thing to notice is that the control sequencer is ↓ triggered.
   a) This is done so that the control signals will be ready and steady for the next ↑ clock edge. (neg edge figure)

A. Instruction Fetch

1. What happens here? mar <- pc, ir <- M[mar], pc <- pc+1
2. Since the first two actions need the bus, this will take at least 2 clock cycles.
3. pc <- pc+1 could happen at the same time as one of the other actions, but for clarity of the description we'll give it its own cycle as well.
4. The first clock cycle, T1, will be used to put pc into mar.
   a) To do this, we enable pc to use the bus (E_P=1) and we set mar to load (L_M=1). (fig)
5. In T2, we will increment the pc. This requires C_P = 1 (fig)
6. in T3, we use the bus to move data from the memory to the instruction register. we need to enable the memory to output to the bus (CE=1) and we need to tell the ir to load (L_I = 1) (fig)
7. These three clock cycles (needed to fetch) are common to all instructions. Once the instruction is in the ir, the steps differ from instruction to instruction.

8. Let's look at the remaining steps for a sub instruction.
9. We need to get the data from memory into B.
   a) First, we need to get the address from the instruction into the mar: Enable ir to output ($E_I = 1$), and load the mar ($L_M = 1$). This is in T4 (fig)
   b) In T5, we enable the memory to output (CE=1) and load the B register ($L_B = 1$). (fig)
10. Once the data is in B, the final task is to cause the adder/subtractor to subtract, and to store the result back in A.
    a) SU=1 will cause put the A/S in subtract mode
    b) Because the A/S is combinational, the result will be at $\sum(7:0)$ as soon as the combinational logic is done.
    c) $E_U = 1$ enables the A/S unit to output to the bus
    d) $L_A = 1$ tells the A register to load.
    e) No conflict, because it won't load till the next rising clock edge.
11. And that's the end of the instruction.

B. Register-Transfer notation

1. A brief aside on some of the notation we've been using so far.
2. mar <- pc means that the value in pc is routed to the input to the mar, and the mar latches this value at the next $\uparrow$ clock edge.
3. This is called **register transfer notation.** It is used commonly because the action of moving data from one register, through combinational logic, to another register, is a common primitive operation at this level of design.

C. Control Points

1. The timing diagram is instructive, but big and messy. We want a notation that's a little more compact.
2. We'll list the action for each clock cycle, and list the corresponding control signals that need to be active. (Show sequence overhead for SUB)
3. We often call these **control points** instead of control signals. This is simply because the points are the places where the control signals feed into the registers etc.
4. A control point is a place to apply a control signal.

D. Loose Ends

1. What happens after T6? T1. We fetch another instruction and execute it.
   a) forever, or until a hlt instruction is fetched and executed.
   b) if pc reaches 1111, it rolls over to 0000 as we would expect a counter to.

2. What's with this "clr" control signal?
   a) This is an initialization signal - starts pc at 0, empties other registers.
   b) like pushing the "reset" button on your computer
3. How do we get the program into the memory?
   a) This is done manually. In later incarnations, we'll have instructions that can write to memory, so we can use a program to write a program to memory.

V. Control Sequencer

1. We know how to do everything except this control sequencer thing.
2. Suppose we had a counter that would generate this T1, T2... sequence on the falling edge of the clock, how would we go about generating the control signals?
3. First, let's look at the required control signals for each instruction (5 instruction control point overhead)
4. First, we see that even though some instructions don't need T4, T5 or T6, we use them anyway, because this significantly simplifies our design.,
5. For the first three steps, all instructions are the same (the fetch part) so it is enough simply to know what clock cycle we are in. For example, E_P will be asserted whenever the sequencer is at T1.
6. The last three steps (T4-T6) are the execution steps, and are different for each instruction.
   a) We can write boolean equations for each control signal as a function of the current state (T4-T6) and the current instruction.
   b) Let's make a signal that is true when we are executing a particular instruction. In fact, let's make a set. (instruction control signals OH)
   c) And the equations for each control signal are thus: (control signal logic OH)
   d) And it's simple to generate a circuit that will implement these functions.

A. Invalid instructions

1. What happens if the value in ir(7:4) isn't one of the opcodes we've defined?
2. This could happen if we fetch data by accident instead of an instruction.
3. If the first 4 bits of data aren't one of our defined opcodes, nothing will happen for the remaining 3 cycles - no control signals will be asserted.
4. But if the data is one of our opcodes, the computer will execute it as per usual, interpreting the last 4 bits as the address, and keep on executing. Who knows what will happen.

VI. Instruction Assembly Language

1. We have a simple instruction set, and we have "english" names for each instruction.
2. We'd like to be able to write a program using these names, as well as symbolic labels for memory addresses.
3. This is what an ASSEMBLER does - it takes assembly language (our symbols and such) and translates them into machine language (the corresponding 1s and 0s that the machine will read)
4. Assembly language statements look like this:

   label: op operands ; comment

   a) The label is a symbolic name for the memory location holding the instruction. the colon is usually required.

   b) "op" is the symbolic name of the instruction. For the SAP-1, this can be one of lda, add, sub, out, and hlt. This name translates to the opcode.

   c) "operands" is the list of operands that is required for the operation. For the SAP-1, this is the address field. In more complicated computers, there may be more than one. Not all instructions need an operand.

   d) The comment is not interpreted by the assembler, and is used to make the assembly language more readable.

      (1) you'll want to use more comments here than in C, for example, because assembly language is more difficult to read and understand.

A. Let's take an example (assembly language overhead)

1. The suffix "H" indicates a hexadecimal number. Another common notation is the prefix "x" or "0x".
2. The "db" which sits in the op field of the instructions op1 and op2 is a **pseudo-instruction**. It stands for "define byte" and is used to assign data to memory.

   a) Each assembler has its own pseudo-instructions, and when we get into HC11 we'll encounter some more.

   b) They are pseudo-ops because they show up in the op field, but are not part of the instruction set.

   c) so "op1:" will define a byte somewhere in memory, and the assembler will take care of where it is when it references it from the lda instruction.
3. So what will this program look like in memory? Our memory has addresses 0 through F, so lets assume we start at address 0

   a) the lda instruction is first, so bits 7:4 of memory address 0H will have 0000 in them. What will be in the address field? We don't know, because we don't know where op1 and op2 will be yet.

   b) Let's assume that the data is put at the other end of the memory, so op2 will be at f, and op1 will be at e.
   c) We can now finish the lda instruction: the last 4 bits should be e=1110.
   d) If we continue, we see that the sub instruction will translate to 2fH, and that since out requires no address, we can put 0 in the address field so the out instruction translates to e0. etc.
4. The assembler is the program that does this for us, and it usually generates a listing file like this (show overhead, note change)
5. The listing file contains the code as you wrote it, plus two more columns on the left.
   a) The Address location of the instruction
   b) The contents of the address location.
6. Locations 4-d are uninitialized and we can't assume anything about their contents.