the SAP-2

cmpt-150-arc

Sections 8-8, 8-9, 9-4, 9-5, 9.6, 9.8

1. We'll do this in bits and pieces, doing the beginning of each section first.

I. Intro

1. The SAP-2 adds a lot of functionality to the SAP-1 hardware, and we'll present this to see what the hardware does, but the important additions are to the instruction set.
2. The hardware diagram and instruction set are on the web.

A. What's new

1. Well, before you get worked up, nothing is really that new. Datapath is bigger, but operation remains the same.
2. Datapath size is still 8 bits (i.e. accumulator is still 8 bits, for eg)
3. the RAM is 64k now, so we need 16 bits to address it, so the bus is now 16 bits.
4. Control sequencer is much bigger as a result of all the extra registers we added, but we won't worry about what's inside.

B. New Registers

1. MEMORY DATA REGISTER: This register holds the result of a memory access, so we can do other things with the bus while the memory access happens.
2. B IS NOW CALLED TMP: this makes more sense because it really is an implementation register.
3. B AND C REGISTERS: these can hold operands and can be used for other instructions. These are archetectural registers, because they show up in the instructions (as we'll see)

C. Other new things (will become clear as we do examples)

1. Adder/Subtractor is now called the ARITHMETIC/LOGIC UNIT, because it will be doing other things besides adding and subtraction, specifically logic operations like complement etc.
2. "flags(0:1)" will tell us useful information about the ALU operations we perform
3. Addresses are now 16 bits, and opcodes are now 8 bits (because we will have more instructions than we had room for in 4 bits of opcode) so instructions take 3 memory locations each (at least)
4. Instructions can now be variable length: the first byte we fetch (the opcode) tells us how many more bytes of the instruction to fetch.

II. Basics of assembly languages

A. Organizing principle: Types of instructions

1. DATA TRANSFER: these instructions move data from place to place
   a) REGISTER-REGSITER TRANSFER: between CPU registers
   b) REGSITER-MEMORY or LOAD/STORE: between mem and registers.
2. ARITHMETIC/LOGIC: The instructions alter data.
   a) ARITHMETIC: add, subtract, mult, etc
   b) LOGICAL: and, or, bit set and bit testing
   c) SHIFT/ROTATE: duh shifts and rotates.
3. FLOW-OF-CONTROL
   a) JUMP: start executing instructions at a new address
   b) BRANCH: goto a new address if a condition is met
   c) SUBROUTINE: goto a new address, using paramaters, with the capability to return.
4. almost all assembly languages organize their instructions more or less like this. it's just a matter of figuring out the specific syntax (which is why we have the manuals!)

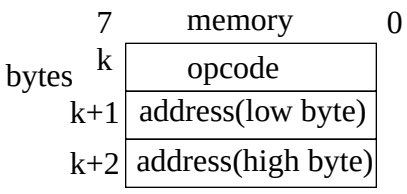
B. Organizing Principle: Operand Format (addressing modes)

1. So far we have seen operands specified as memory locations in the instruction, or specified as registers.
2. We'll look at more of these as we encounter them.

III. Instructions

A. Load/Store instructions

1. The SAP-2 can now store data into the RAM as well as load data.
2. Recall: instructions that need a memory address are 3 bytes long: one for the opcode, and the remaining 2 for the high and low halves (bytes) of the address.



3. This is **direct addressing**: specifying the operand by including the memory address.
4. Example: Store what's in the A register in memory 3c55h
   a) SAP-2: sta 3c55 is translated to 32h / 55h / 3ch
   b) HC11: staa 3c55 is translated to b7h / 3ch / 55
   c) Note that HC11 and SAP-2 use different order for the halves (bytes) of

the address.

(1) in HC11, the most significant byte comes first (smaller address)

(2) in SAP-2, the least significant byte comes first.

d) Note also that the instructions have different opcodes - This is not surprising, as the different instruction sets have different instructions etc.

e) We find the opcode for the HC11 instruction by looking in appendix A

f) it tells us that if we want to store what's in "A" to a two-byte memory address, we use opcode 67.

g) If you look, it also tells us that this is called "extended" addressing. For the HC11, Direct addressing just like we use it, except with the most significant byte ="00h".

h) Extended addressing in the HC11 is direct addressing using the full 16 bit address.

5. Note that while the HC11 has staa and stab, the SAP-2 doesn't have stb or stc

a) This kind of asymmetry is common in assembly language design - some other function was deemed more important in the SAP-2.

B. Immediate Operands

1. An immediate operand is specified in the instruction itself instead of memory.
2. Instead of going to memory to find the value, the value is in the instruction after the opcode, where the address was in direct addressing.
3. This is called immediate addressing.
4. It is used frequently for small integers (i.e. ±1,2,3, ascii characters etc)
5. It is more efficient than direct addressing because it needs only one memory fetch after the opcode (to fetch the next chunk of the instruction)

a) Direct addressing needs two memory fetches: Get the next chunk of the instruction, which is the address, then use that address to get the data.

6. Example: to load ascii "Z" into register B. (SAP-2)

a) Instruction code is MVI (Move Immediate)

b) The hex code for ascii "Z" is 5ah

c) The full instruction is MVI B, 5ah

d) and the machine language for the full instruction is 06h first byte, 5ah second byte. (as shown)

7	memory	0
k	06h	
k+1	5ah	

7. And in HC11:

a) instruction code is LDA (load accumulator)

b) The version we want is LDAB (machine code c6h)

c) The full instruciton is LDAB #5ah. The # indicates an immediate operand (see chap 6). The machine code looks like:

	7 memory 0
k	c6h
k+1	5ah

C. Some notes on opcodes

1. We're looking at two assembly languages at the same time, and they do things a bit differently.
2. The OPCODE is what tells the computer what to do, i.e. how many more bytes to fetch and what to do with them.
3. Different assembly languages use opcodes in different ways.
4. For example, take these two operations: Load register A with the immediate value 2ah (A <- 2ah), and Load register A with the value in memory location 2ah (A <- M[2ah]).
   a) in SAP-2,
      (1) we use MVI A, 2ah for the immediate (3eh / 2ah)
      (2) and LDA 002ah for the direct. (3ah / 2ah / 00h)
   b) in HC11,
      (1) we use LDAA #2ah for immediate (86h / 2ah)
      (2) and LDAA 002ah for the direct (b6h / 00h / 2ah)
5. SAP-2 uses different mnemonics for the different addressing modes, but HC11 uses the same mnemonic and the addressing mode is distinguished by the prefix to the operand.
6. Note that in both cases, the operand is different for different addressing modes.

D. Inherent or implied Operands

1. If there is an instruction that always uses the same operand, we don't need to explicitly state it. the operand is INHERENT in the instruction.
2. The opcode is all we need in this case.
3. This is called INHERENT or IMPLIED addressing mode.
4. All of the SAP-1 instructions used inherent addressing - the A register was always implied (except in HLT).
5. The register - register transfer instructions in SAP-2 are inherent.
6. EXAMPLE: if we want to move what's in A into B (B<-A) we use the instruction "MOV A,B". The opcode is 78h, and the operands (A and B) are implied in the opcode.
7. The comprable HC11 instruction is TAB (transfer A to B). In this case, even the assembly language has no operands. The opcode is 16h and that's all there is.

8. Another example: in SAP-2, the ADD instruction implies ADD to A.
   a) ADD B executes A <- A+B, and the opcode is “80h”, no operands.
   b) in HC11, the instruction is ABA (Add B to A), and the machine language is simply the opcode “1bh”
9. Note that unlike direct and immediate oparands, inherent addressing doesn’t tell you anything about where the operand is. It could be a register, or it could be a constant (increment a register) or other things (as we’ll see)
10. And be aware that it is possible for each operand of an instruction to be specified with a different addressing mode.

E. Register Operands

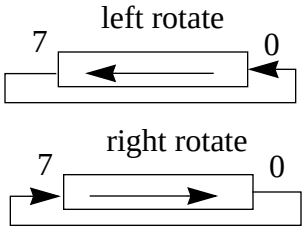
1. Registers are very common inherent operands. In fact, all of the examples we have seen so far of inherent operands have been registers.
2. There is a special name for inherent operands that are registers - Register addressing. It’s a little more useful, though the HC11 doesn’t use it explicityl.

F. Logical instructions

1. We’ve looked briefly at arithmetic instructions (add etc) but one of the improvements in the SAP-2 is this “Arithmetic Logic” unit. Well, as the name implies it can do logic as well.
2. SAP-2 can do NOT, AND, OR, XOR, and the opcodes are CMA (complement), ANA, ORA, and XRA respectively.
3. Each of these use register A as an inherent operand, and the other operand is specified immediately as register B or C.
4. The result is a bitwise logical operation. eg if A=00001111 and B=10101010 then ANA B would result in A=00001010.
5. There are immediate versions of these as well: ANI, ORI, XRI
6. the corresponding HC11 instructions are COM, AND, ORA, EOR

G. Shifts and Rotates

1. SAP-2 has two instructions that do register rotate.
2. Rotate is like a shift except that the MSB is transferred to the LSB for a left rotate, and for a right rotate, the LSB is fed into the MSB.
3. SAP-2 instructions are RAL and RAR (for rotate A left and rotate A right
4. HC11 has ROR and ROL (usage, eg: RORA), but they’re slightly different, as we’ll see.



H. Flow of Control Instructions

1. We've got a lot of functionality so far, but it's not very useful if we have to go from top to bottom
2. We want to be able to change what we do depending on what we have just done (if-then constructions and loops!)
3. Before we see how we do this, we need to talk about condition codes
   a) CONDITION CODES are single-bit values that record useful info about the result of an ALU operation.
   b) In the SAP-2, these are the flags(1:0) that we saw.
   c) One flag is set if the result of the ALU operation is zero (the 'Z' flag)
   d) The other is set if the result of the operation is negative (the 'S' flag, for sign)
   e) Only operations that use the ALU will affect the condition codes. These are in table 11-3 as well.
      (1) Usually, Arith or Logic operations affect the control codes, while data movement instructions do not.
   f) HC11 has 8 condition code bits (see section 6.1.5 on page 6-4 of the manual) and the analgous condition code bits are Z for zero and N for negative.
   g) the HC11 has a C condition code for carry-out, and V condition code for overflow. These are affected by arithmetic operations but not by logic operations.
4. Condition codes are used in flow-of-control operations.
   a) in SAP-2, these are called jumps, and there are two types: Conditional and unconditional.
   b) The operand of the jump is the location in memory to jump to.
5. Recall that for normal operation, an instruction fetch looks like this:
   a) The value in PC is used as the address of the instruction to fetch
   b) PC is incremented during the fetch, so it is ready to get the next instruction
   c) The instruction fetched from PC is stored in ir.
6. The important thing to notice is that the PC is already pointing at the next instruction when the current instruction begins to execute.
7. If we want to fetch a different instruction, we need to change the value in the pc.
8. An unconditional jump loads a new value into the pc.
9. A Conditional jump loads a new value into the pc only if the condition is met.

10. The condition that may or may not be met depends on the result of the last instruction executed.
    a) For example, in the SAP-1, the jz instruction consults the "zero" flag (from the ALU) and will jump if the zero flag is 1. If not, the next instruction in sequence is fetched

I. Example Assembly Code

1. Let's discover what assembly language would be generated from this "High-level language" segment (in pseudo-C) [overhead]

```
if(widget_cnt > 0)
{ got_widgets = 1;}
else
{got_widgets = 0; }
```

2. How would this look in SAP-2?
    a) We need memory storage locations for "widget_cnt" and "got_widgets", so let's assume we've allocated a couple of bytes, say 2ah and 2bh.
    b) We need to figure out what needs to be done.
        (1) Load "widget_cnt" into a register
        (2) test if it's greater than zero
        (3) use jump commands to go to teh appropriate chunk of code
        (4) place a 1 or a 0 in got_widgets, whichever is appropriate.
3. we can use lda to load the value into the A register.
4. We need an instruction that will give A at the ALU without changing A.
5. Lots of instructions affect the condition codes (see overhead) but we want to use one that has the correct result with the least side-effects.
    a) add, sub, anda, ora, xra all require something to be in B or C
    b) ani, ori, xri will work well because they use immediate addressing.
    c) let's use ori 00h.
6. Here's the assembly language fragment (overhead)
    a) org is a "pseudo-op" to tell the assembler to start putting instructions or data in a specific place in memory.
    b) With this command, we needn't write our assembly language program in order, as long as we keep our orgs straight.
    c) but the assembler will overwrite previous org's if we tell it to, so we gotta be careful.
7. the "lda" and "ora" instructions get the value of "widget_cnt" as we have already seen.

8. Then we want to do one of two things depending on the value of the “z” and “s” flags.
9. First, if “z” is true, we jump to “no widgets”.
10. Then, if “s” is true, we also jump to “no widgets” (jm=jump if minus)
11. These two together skip to the “else” clause if widget_cnt ≤ 0.
12. If neither s nor z are true, the next instructions (“widgets”) are executed.
    a) but then we need to skip over the “else” clause, so we jmp to set_value, which is where execution continues after both the if clause and the else clause: we store the A value back into memory.
13. Note that we could do this the other way: have the else clause first, and jump over it if widget_cnt > 0.

J. Example listing file

1. If we were to assemble that code, we’d get this listing file (overhead)
2. Leftmost column shows memory address in hex
3. next column shows memory contents, in hex.
4. During assembly, the assembler resolves labels to memory addresses, and uses them in other instructions as necessary.
    a) eg, for jz no_widgets
    b) This instruction is in memory 0005h to 0007h.
    c) The opcode is cah, stored in memory 0005h.
    d) the target of the jump (the operand) is the mvi instruction, located in memory location 0010h, so that address is placed in memory 0006h and 0007h, LSbyte first as is the convention for SAP-2.
5. Another example: mvi is an immediate instruction, which moves the value in “A” to the indicated memory. The assembly language said put it in “got_widgets”, but the assembler knew that this corresponds to memory location 2bh.