The Simple Class Machine

Our first machine example has an accumulator-based processor with a small number of heterogeneous registers and limited data pathways. It has an instruction set that is sufficient to perform reasonably complex tasks such as adding an array of numbers or finding the largest number in an array. Although such tasks are possible, they are clumsy due to its limited register set.

The simple machine has a main memory consisting of 16-bit words. This memory is initialized with a set of words (called the program) before the machine begins execution. In keeping with the *stored-program concept*, the only distinction between instruction and data words is how they are interpreted. Memory is word-addressed: a word is addressed by its offset as if it is in an array of *shorts*. The first word is at address 0, the next is at address 1, etc. The class machine always begins execution at address **`0`**, uses a classic *fetch-decode-execut*e instruction cycle, and stops when a **`HALT`** instruction (**`0x0000`**) is executed. Thus, the word at address **`0`** must be an instruction, and the natural organization of a program is with instruction words first, followed by data words.

The processor of our machine has seven 16-bit registers:

- The **`pc`** or program counter holds the address of the ***next*** instruction to execute.
- The **`ir`** or instruction register hold the *current instruction being executed*.
- The **`accum`** or accumulator is the only true data register. All loads, stores, adds and subtracts reference the accumulator as at least one of the operands.
- **`ctr`** is the counter register. Although data can be moved from the **`accum`** to the **`ctr`** (using the **`MVAC`** instruction) and the **`ctr`** can be added to the accumulator (using **`ADDC`**), its use mainly is as a simple counter. **`ctr`** is decremented by one using the **`DEC`** instruction, and all conditional branch instructions use it for comparisons.
- **`mar`** is the memory address register. This is the register that indicates the address of the next memory word to be referenced (either loaded from or stored to)
- **`areg`** is the address register. It is used solely for indirect memory operations. The **`LIA`** and **`SIA`** instructions use the address in **`areg`** as the source (**`LIA`**) or target (**`SIA`**) address for the data being placed in (**`LIA`**) or coming from (**`SIA`**) the accumulator.
- **`mbr`** is the memory buffer register. This register is the window to memory. When memory is retrieved, the word loaded goes here. When memory is stored, the word stored comes from here.

`mar` and **`mbr`** are the gateway between the processor and memory. They work in concert. When memory at a particular address is needed, the **`mar`** is set to the address, and a memory **`get`** cycle is initiated. The result of the **`get`** is to place the addressed memory word in the **`mbr`**. Conversely, when a data word must be placed in memory at a particular address, the address is placed in the **`mar`**, the word is placed in the **`mbr`** and a **`put`** operation is performed to place the contents of the **`mbr`** in memory at the address indicated in the **`mar`**. Note that these instructions may fault, if the address in the **`mar`** is not legal.

Besides having a very limited register set, our simple machine also has limited data pathways. These are shown on the diagram in the handout SimpleMachineDatapaths.

Register transfer language

As you might imagine, all the operations of our simple machine consist of altering the internal registers by moving data between them and between memory, possibly accompanied by performing an arithmetic operation in transit. In fact, the instructions that we are going to cover shortly are actually a shorthand for one or more simpler operations of this type (called register transfers). These register transfers can be referred to as microinstructions, and can be described in a language known as a *register transfer language* (**rtl**) (also called a *register transfer notation*). **rtl** simply indicates the target register on the left and the source and operation on the right, with a left-pointing arrow between them. Although the description of the register transfer language of a real machine is complex, that needed to describe our machine is very simple. It only has a few general statements:

rtl statement	*interpretation*
reg <- reg [op reg]	Transfer a register, possibly performing an operation *op* (+ or -) in transit.
reg <- reg op const	Transfer a register, adding or subtracting (*op*) a constant in transit. In our machine this is only used to decrement the counter register (ctr) and increment the PC register (pc), so the only legal **const** is 1
***reg* <- 0**	Clear the register
if (reg log-op const) **{** ***rtl-statements*** **}**	Used as a conditional. **reg** is a register specifier (a register or a part of a register (see below)) **const** is a constant number and **log-op** can be a C logical operator (**==, <, <=, >, >=, !=)** For the current instructions implemented for our SM, the only register **reg** can be is the counter, the only legal const is **0** and only a few ***log-op***s are used.
get(), put()	Transfer information between the processor registers and memory using the **mar** and **mbr** registers.
halt	Halt the machine

In this notation, *reg* is one of our processor registers. It can also be part of a register by indicating the bit positions in the register. For example,. the notation **ctr(a:b)** indicates only bits **a** to **b**, inclusive, of the **ctr** register (see below for bit numbering conventions). In our Simple Machine, the only register fields that we need to use are the ADDR and OPCODE parts (see below).

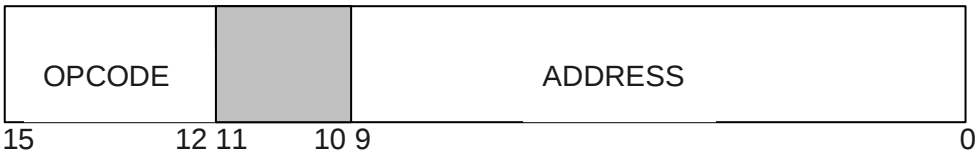
Below are some example descriptions of simple machine operations in our **rtl**:

rtl statement	*interpretation*
accum <- mbr	Transfer the contents of the **mbr** to the **accum**
mar(9:0) <- ir(9:0)	Transfer the least-significant (rightmost) 10 bits of **ir** to the same bits of the **mar**.(Note that this is PART of decoding the address part of an instruction (see below))
pc <- pc + 1	Increment the value in the **pc** by one (transfer the value of the **pc** back to the **pc** after incrementing it by one)
get()	Transfer(**load**) the word found at the address in **mar** to the **mbr**. If we denote memory as an array, this operation could be written as **mbr <- memory[mar]**

Of course, not all combinations of registers, operations and conditionals that can be expressed in our **rtl** are legal in our machine. For example, the statement **ctr <- mbr** is perfectly legal **rtl,** but the **mbr** can only be transferred to either the **ir** or the **accum** in our machine description. However, since the instructions that our simple machine uses express the architecture, you would never have need of this kind of transfer.

Simple Machine Instructions

Instruction words are 16-bits, divided into 4 bits of opcode, 2 bits reserved, and 10 bits of address:



Below is a list of the opcodes for our class machine (from **sim.h**):

```
#define HALT    0       /* halt the machine */
#define LOAD    1       /* load accumulator from memory */
#define STORE   2       /* store accumulator to memory */
#define ADDC    3       /* add counter to accumulator */
#define MVAC    4       /* move accumulator to counter */
#define JEQ     5       /* jump to address if counter equals 0 */
#define JLT     6       /* jump to address if counter is less than 0 */
#define JMP     7       /* jump to address */
#define ADD     8       /* accumulator = memory + accumulator */
#define SUB     9       /* accumulator = memory - accumulator */
#define DEC     0xA     /* decrement counter */
/* LA takes the memory address from the instruction and loads it
   into the accumulator (load address) */
#define LA      0xB
/* LIA uses the memory address in the areg to load memory into the
   accumulator (load indirect address) */
#define LIA     0xC
/* SIA stores the value in the accumulator in the memory address
   found in the areg (store indirect address) */
#define SIA     0xD
/* MVAA moves the value in the accumulator to the areg */
#define MVAA    0xE
```

Let's think about what a couple of these instructions actually do, and express them in our register transfer language. At the beginning of execution of each instruction, the instruction is in the instruction register (**ir**).

Instr.	*Encoding*	*What's required*	*rtl statements*
LOAD 4	**0x1004**	Extract the address (4) from the instruction. **get** the memory at address 4 (put in **mbr**) Transfer it to the accumulator	**mar(15:10) <- 0** **mar(9:0) <- ir(9:0)** **get()** **accum <- mbr**
JEQ 0x24	**0x5024**	If the counter register is zero: set the **pc** to the address in the instruction	**if (ctr == 0) {** **pc(15:10) <- 0** **pc(9:0) <- ir(9:0)** **}**
MVAC	**0x4000**	Move the accumulator to the counter	**ctr <- accum**

Note: the two steps **mar(15:10) <- 0** and **mar(9:0) <- ir(9:0)** comprise a single transfer.

Now we can see how to execute individual instructions. We need to see how to cycle through the instructions so that we can execute each of them in sequence

The Fetch-Decode-Execute Cycle

The processor begins by setting the pc to a known value, in our case zero. Then it continues cyclically executing instructions as follows:

- fetch the instruction whose address is indicated by the **pc**, placing it in the **ir**
- increment the **pc**
- decode the instruction (in our simple machine, decoding the instruction can be represented by referring to pieces of the **ir**, such as **ir(15:12)**
- execute the instruction, possibly altering the **pc**

After the instruction is executed, the cycle begins again, fetching the next instruction. This continues in our simple machine until a **HALT** instruction is executed.

Let's write the algorithm for our simple machine's fetch-decode cycle, and add the conditional execution for the three instructions we coded above. The clauses for the other instructions are left as an exercise.

```
pc <- 0
repeat
      mar <- pc
      pc <- pc + 1
      get()
      ir <- mbr
      if (ir(15:12) == 1) {
            /* LOAD */
            mar(15:10) <- 0
            mar(9:0) <- ir(9:0)
            get()
            accum <- mbr
      }
      if (ir(15:12) == 4) {
            /* MVAC */
            ctr <- accum
      }
      if (ir(15:12) == 5) {
            /* JEQ */
            if (ctr == 0) {
                  pc(15:10) <- 0
                  pc(9:0) <- ir(9:0)
            }
      }
endloop
```

A good exercise for you to perform after you finish reading this handout is to describe the register transfers of the remaining instructions in our **rtl**. When you do this, you should refer to the data pathways of the simple machine from the handout *SimpleMachineDatapaths*. You should also observe the additional restriction that there is a single memory cycle to fetch an instruction and a single memory data cycle per instruction execution.

Although describing instructions in **rtl** is necessary for understanding how the machine works and for simulating it, the goal of any computer is to use it. This requires writing programs with the instructions.

Simple Machine Programs

Our Simple Machine is, of course, fictitious. We simulate its operation using a program or simulator, **sim**. Programs for the simulator consist of a sequence of up to 256 16-bit words. (Note that this is not the limit of the address capability of the Simple Machine (although there is a limit), but this is a reasonable limit for a program for our simulator.) When we create a program for the simulator, it first translates it from text to numbers. We represent the numeric data (the 16-bit words) in hexadecimal notation. For example, **0x4000** represents the **MVAC** instruction. For our convenience, our input file may have annotations, which the simulator ignores. An annotation begins with a **#** character. Annotations may either be at the beginning of the line or to the right of a hexadecimal word. Following is a sample program for this machine. (This is the program **copy0** from the standard test files as detailed later.)

> ***Every line of a Simple Machine program must begin with either a comment character # (for annotations) or a 0x (for the encoding of an instruction). Comments are allowed on instructions to the right of the encoded instruction.*** **If a line begins with whitespace it will be interpreted as the end of the program.**

```
#
# this first test case simply copies
# a data word from one location to another
# the first word is always address 0, the
# second at address 1, etc.
#
0x1003  # (address 0) load word to copy from address 3
0x2004  # (address 1) store it in new location (address 4)
0x0000  # halt
0x4040  # (address 3) the word to copy
0x0000  # (address 4) the location to copy to
```

Let's look briefly at this program:

- The first instruction (at address **0**) has opcode 1 (**load**) and address 3. This loads the 16-bit word from address 3 into the accumulator.
- The next instruction (at address **1**) has opcode 2 (**store**) and address 4. The accumulator is stored at address 4.
- The last instruction (at address **2**) has opcode 0 (**halt**). Each program must end with this instruction.
- The remaining words comprise the *data area*, which starts at address **3**.

The simulator for this machine is available in the **sim** directory beneath the class public data area on *hills* (see the class PolicyStatement). Its name is **sim**. The syntax for running it is simply

```
sim [-v] inputfile
```

where **inputfile** is a program file in the format indicated above. The output of **sim** executing the **copy0** program follows:

```
$ ./sim /pub/cs/gboyd/cs270/sim/tests/copy0
./sim: INPUT FILE = /pub/cs/gboyd/cs270/sim/tests/copy0
******* BEFORE SIMULATION ********
   memory[0]    = 0x1003
   memory[1]    = 0x2004
   memory[2]    = 0x0000
   memory[3]    = 0x4040
   memory[4]    = 0x0000
3 instructions executed.
******* AFTER SIMULATION ********
```

```
   memory[0]    = 0x1003
   memory[1]    = 0x2004
   memory[2]    = 0x0000
   memory[3]    = 0x4040
   memory[4]    = 0x4040
$ # note that the word at memory address 3 has been copied to the
$ # memory address 4. (That's the point of the program!)
$
```

The **-v** option (for verbose) causes **sim** to dump its registers and pause after each instruction is executed. It also outputs a word of status for each memory reference. Here is the output of **sim** on the **copy0** program using the **-v** option:

```
$ ./sim -v /pub/cs/gboyd/cs270/sim/tests/copy0
./sim: INPUT FILE = /pub/cs/gboyd/cs270/sim/tests/copy0
******* BEFORE SIMULATION ********
   __memory[0]  = 0x1003
   __memory[1]  = 0x2004
   __memory[2]  = 0x0000
   __memory[3]  = 0x4040
   __memory[4]  = 0x0000
START:PC = 0x0000, IR = 0x0000, ACCUM = 0x0000, CTR = 0x0000, AREG = 0
FETCH:get() called. mar=0x0000. mbr set to 0x1003
get() called. mar=0x0003. mbr set to 0x4040
PC = 0x0001, IR = 0x1003, ACCUM = 0x4040, CTR = 0x0000, AREG = 0
Hit <ENTER> to continue:
FETCH:get() called. mar=0x0001. mbr set to 0x2004
put() called. mar=0x0004, mbr=0x4040
PC = 0x0002, IR = 0x2004, ACCUM = 0x4040, CTR = 0x0000, AREG = 0
Hit <ENTER> to continue:
FETCH:get() called. mar=0x0002. mbr set to 0x0000
PC = 0x0003, IR = 0x0000, ACCUM = 0x4040, CTR = 0x0000, AREG = 0
3 instructions executed.
******* AFTER SIMULATION ********
   __memory[0]  = 0x1003
   __memory[1]  = 0x2004
   __memory[2]  = 0x0000
   __memory[3]  = 0x4040
   __memory[4]  = 0x4040
$
```