SYSTEMS II — PROJECT 1
Introducing a new ISA

1 The AMHCS ISA

This ISA with which we will be working is good for simulation and relatively easy to program, but also unrealistic in many ways. It therefore makes a good basis for the kinds of projects we will pursue in this course.

1.1 Basic concepts

There are a few key, high-level elements of this ISA that merit specification:

- **Word size:** This ISA uses 16-bit/2-byte words. The arithmetic/logic sources are 16-bit values, and all main memory addresses are likewise 16-bits each. Consequently, the maximum addressible memory is $2^{16}\ bytes = 64\ KB$.
- **Instruction size:** Each instruction for this ISA is 64-bits/8-bytes/4-words. The instructions are uncommonly large to make programming in this ISA easier. Specifically, each of the operand spaces is a word, allowing a programmer to specify full immediate constants or full main memory addresses.
- **No named registers:** There are no addressible registers in this ISA, leaving only main memory for storage. Source and destination values are all drawn either from immediate values within the instruction or from main memory itself.
- **Endianness:** This ISA is *big-endian*. That is, the most significant bit, labelled bit number `15`, is the left-most one, while the least significant bit, labelled number `0`, is the right-most.
- **Direct vs. indirect operands:** Each operand can be *direct* or *indirect*. That is, each word-sized operand value may be a *direct* representation of the value that should be used by the instruction, or it may be an *indirect* reference to a main memory location that contains the value that should be used. For example, on an arithmetic instruction (e.g., `ADD`), the two input values are provided by the operands *source A* and *source B*. Each operand may be direct or indirect. If one is direct, then the operand value itself is provided as an input to the arithmetic operation; if it is indirect, the operand value is main memory address whose contents provides an input.

1.2 Machine code format

Each machine instruction, which is 64-bits in length, has the following format:

- `[63 - 56]` **Opcode:** An 8-bit value that specifies the operation that the CPU should perform. See Section 1.3 for a complete list of opcodes.
- `[55 - 48]` **Operand flags:** A collection of 8 boolean values that specify how the operand values should be interpreted. Specifically:

  - `[55 - 52]` **Unused:** These flags are unused and reserved for future purposes.
  - `[51]` **Direct source B:** Set is the operand **source B** is a *direct value*; unset if it is an *indirect value*.
  - `[50]` **Direct source A:** Set is the operand **source A** is a *direct value*; unset if it is an *indirect value*.
  - `[49]` **Direct destination:** Set is the operand **destination** is a *direct value*; unset if it is an *indirect value*.
  - `[48]` **PC-relative target:** Set if the operand **destination** is a *PC-relative target*, unset if it is an *absolute target*.

- `[47 - 0]` **Operands:** A collection of 3 operands that specify input and output values and addresses. Specifically:

  - `[47 - 32]` **Destination:** For instructions that produce a result value, the address at which that result should be stored. If this value is *direct*, then this operand is used as the main memory location at which the result is stored; if the value is *indirect*, then the value contained at the memory location specified by the operand is used as the main memory location at which to store the result—the operand specifies a location that contains the destination address.
  - `[31 - 16]` **Source value A:** For instructions that require at least one input value, the first such value. If *direct*, the operand is the input value; if *indirect*, the operand specifies the main memory location that contains the input value.
  - `[15 - 0]` **Source value B:** For instructions that require two input values, the second such value. If *direct*, the operand is the input value; if *indirect*, the operand specifies the main memory location that contains the input value.

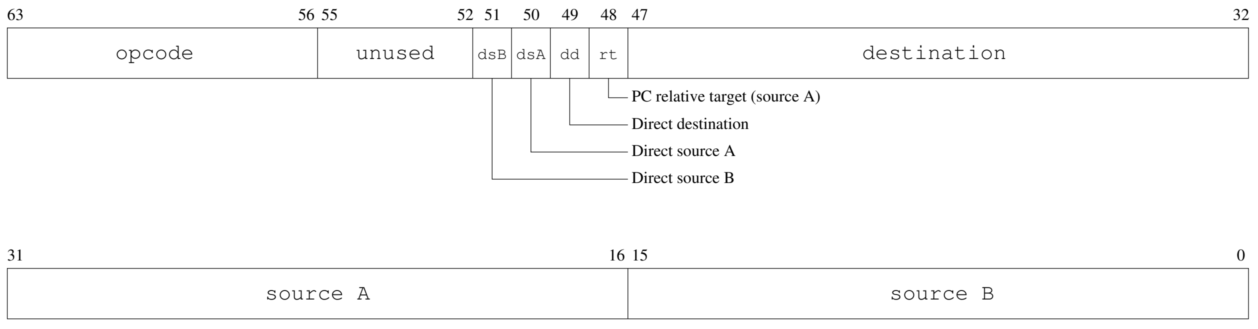

Figure 1: The layout of each 64-bit machine-code instruction.

1.3 Instruction list

The following is a list of the instructions that the AMHCS ISA provides, broken down into categories.

1.3.1 The empty instruction

This special category of instruction has only one entry, and it is notable in that it performs no work.

- `00:` `NOOP`

  This instruction ignores all three operands. It performs **no computation** and modifies no state.

1.3.2 Arithmetic/logic instructions

These instructions all accept one or two source inputs, perform an arithmetic or logic manipulation, and produce a result to be stored.

- `0x01:` `NOT` *[destination] [source A]*

  Take the value specified by *source A* and **invert** each of its bits, storing the result at the *destination*.

- `0x02:` `AND` *[destination] [source A] [source B]*

  Perform the bitwise logical **and** of the values provided by *source A* and *source B*, storing the result at the *destination*.

- `0x03:` `OR` *[destination] [source A] [source B]*

  Perform the bitwise logical **inclusive or** of the values provided by *source A* and *source B*, storing the result at the *destination*.

- `0x04:` `XOR` *[destination] [source A] [source B]*

  Perform the bitwise logical **exclusive or** of the values provided by *source A* and *source B*, storing the result at the *destination*.

- `0x05:` `ADD` *[destination] [source A] [source B]*

  Perform the arithmetic **addition** of the values provided by *source A* and *source B*, storing the result at the *destination*.

- `0x06:` `SUB` *[destination] [source A] [source B]*

  Perform the arithmetic **subtraction** of the values provided by *source A* and *source B* (subtracting *B* from *A*), storing the result at the *destination*.

- `0x07:` `MUL` *[destination] [source A] [source B]*

  Perform the arithmetic **multiplication** of the values provided by *source A* and *source B*, storing the 32-bit, double-word result at the *destination*.

- `0x08:` `DIV` *[destination] [source A] [source B]*

  Perform the arithmetic **division** of the values provided by *source A* and *source B* (dividing *A* by *B*), storing the 32-bit, double-word result at the *destination*.

- `0x09:` `SHFTL` *[destination] [source A] [source B]*

  Shift the bits of the *source A* value to the **left** (from less to more significant) by the number of bits specified by *source B*, storing the result at the *destination*. `0`-valued bits will be inserted into the less significant positions.

- `0x0A:` `SHFTR` *[destination] [source A] [source B]*

  Shift the bits of the *source A* value to the **right** (from more to less significant) by the number of bits specified by *source B*, storing the result at the *destination*. `0`-valued bits will be inserted into the more significant positions.

1.3.3 Unconditional branching instructions

Unconditional branching instruction alter the program counter without testing or comparing any state.

- `0x0B:` `JUMP` *[destination]*

  Set the program counter to the target given in the *destination*. If the destination is *relative*, then the value given in the operand is an offset from the current PC, and thus is added to it; if the destination is absolute, then the value specified in the operand is copied into the PC.

- `0x0C:` `CALL` *[destination] [source A]*

  Like the `JUMP` instruction, **set the program counter** to the target given in the *destination*, whether relative or absolute. Additionally, **store the `PC + 2`**—the address of the instruction that follows the `CALL`—at the address specified (directly or indirectly) by *source A*.

1.3.4 Conditional branching instructions

Unlike unconditional branching instructions, these alter the program counter only if the particular test of existing state provides a true result.

- `0x0D:` `BEQ` *[destination] [source A] [source B]*

  Compare the values specified by *source A* and *source B* for **equality**. If this comparison yields a true result, then **set the program counter** to the target specified by the *destination*.

- `0x0E:` `BNEQ` *[destination] [source A] [source B]*

  Compare the values specified by *source A* and *source B* for **inequality**. If this comparison yields a true result, then **set the program counter** to the target specified by the *destination*.

- `0x0F:` `BGT` *[destination] [source A] [source B]*

  Compare the values specified by *source A* and *source B*. If *A* is **greater than** *B*, then **set the program counter** to the target specified by the *destination*.

- `0x10:` `BGTE` *[destination] [source A] [source B]*

  Compare the values specified by *source A* and *source B*. If *A* is **greater than or equal to** *B*, then **set the program counter** to the target specified by the *destination*.

- `0x11:` `BLT` *[destination] [source A] [source B]*

  Compare the values specified by *source A* and *source B*. If *A* is **less than** *B*, then **set the program counter** to the target specified by the *destination*.

- `0x12:` `BLTE` *[destination] [source A] [source B]*

  Compare the values specified by *source A* and *source B*. If *A* is **less than or equal to** *B*, then **set the program counter** to the target specified by the *destination*.

2 Obtaining the tools

Before we examine the assembler and simulator, begin by obtain the source code and compiling it. Specifically, follow these steps:

1. Login to one of the CS department systems. Either login to a workstation in Seeley Mudd 007, or connect via `ssh` to `castor.cs.amherst.edu`. If you are unfamiliar with `ssh`, it is available for Windows, Mac, and Linux machines, so contact me for more information on how to install and use it on your machine.
2. Be sure that you have a shell or terminal window at which you can type commands.
3. At the shell prompt, create a directory for your work for this class and then change into that directory:

   ```
   $ mkdir cs26
   $ cd cs26
   ```

4. Now make a directory for this project and change into it:

   ```
   $ mkdir project-1
   $ cd project-1
   ```

5. Obtain the source code from my directory. When you enter this command, you should see a list of files and directories that are created at the source code is copied into your current directory:

   ```
   $ tar -xzvpf ~sfkaplan/public/cs26/vp-project-1-v3.tar.gz
   ```

6. Change into the newly created directory for the assembler and compile the code there:

   ```
   $ cd assembler
   $ javac *.java
   ```

7. Change into the directory for the system simulator and compile that code as well:

```
$ cd ../system
$ javac *.java
```

You now have the completed tools. Feel free to examine and modify (if you choose) the source code that you find in the `.java` files.

3 The AMHCS assembler

Since writing in machine code is unpleasant, there is an assembler for this ISA. Please note that this assembler has been tested, but it has not been thoroughly used. It likely contains bugs. More importantly, although I have made efforts to provide a useful and usable assembler, it is likely to provide error messages that are difficult to decipher when your assembly code is malformed. If you find bugs or poor error messages, please send me the assembly code that produces the undesirable result so that I may improve the assembler.

Below is a description of the assembly code syntax, as well as instructions on how to use the assembler and examine its output.

3.1 Assembly code syntax

Most likely, the best way to absorb the syntax used for writing `AmhCS` assembly programs is by example. There is a small example that does not show all features of this syntax, but it does get you started. To see it, assuming that you've begun with the instructions above:

```
$ cd ~/cs26/project-1/assembler
$ emacs add-two-numbers.asm &
```

You should see the following file contents:

```
;;; Add two integers.

        .Code

        ;; Store two integers in main memory locations.
        OR  0x200 0 5
        OR  0x202 0 -3

        ;; Add the two integers, using an indirect destination.
        OR  0x300 0x204 0
        ADD @0x300 @0x200 @0x202   ; Indirect sources, too.
```

There are a number of critical features in this example worthy of mention:

- **Comments:** A semicolon (`;`) marks the beginning of a comment; any text that follows a semicolon is ignored by the assembler. The use of additional semi-colons are part of an assembly convention employed by Emacs: 3 semicolons (line 01) for comments that begin at the start of the line of text; 2 semicolons (line 05 and 09) for comments that begin tabbed to the depth of an opcode; and 1 semicolon (line 11) for comments that follow an actual line of assembly code.
- **Mode markers:** Line 03 sets the *assembly mode*. A mode marker is always on a line of its own, and always begins with a period (`.`). A more thorough description of the modes is provided in Section 3.1.1. In this specific case, the `Code` mode is one in which the assembler expected to process a sequence of instructions. Therefore, this particular mode marker must appear before any instructions in your assembly programs.
- **Operands values:** First, notice that the operand values are sometimes decimal, sometimes hexidecimal. See Section 3.1.2 for more on specifying word-sized values in different ways. Additionally, notice that each operand may be direct or indirect. See Section 1.1 for a clear definition of the difference between those two, as well as Section 3.1.2 for more on how to specify the direct/indirect attribute for each operand.

3.1.1 Mode change markers

There are four assembly modes:

1. **Preamble:** The assembler begins in this mode, processing only comments while waiting for a mode change to specify another mode.
2. **Code:** The primary mode that you will use, in which you can list the sequence of instructions that compose a program. In this mode, comments, intrustions, and labels on instructions are recognized.
3. **Numeric:** In this mode, you can specify a sequence of literal integer values. You may specify one or more labels, thereby marking the address of a constant. Each sequence of word-sized values can be of any length, and may be expressed in any of the forms show in Section 3.1.2. For example:

```
        .Numeric
0 0b10110001 0xe39a
  L5: -12
```

4. **Text:** Specify a literal sequence of byte values, where each byte is provided as an ASCII character. Labels can be provided to specify where a string begins. For example:

```
        .Text
  MSG: "The quick brown fox jumps over the dazy log (spoonerism intenti
```

3.1.2 Word values

In any place where word-sized values are expected, the assembler allows three methods for specifying these 16-bit values:

- **Decimal:** With no prefix, a number in the range $[-32,768, 32,767]$ (the minimum and maximum values for two's complement). For this form, numbers use the usual digits 0 to 9, with an optional negative designation (with a leading $-$), and no other charcaters (e.g., commas).

- **Hexidecimal:** With the prefix `0x`, any 16-bit value. Since each each hexidecimal digit (0 to 9, A to F) corresponds to four bits, then any sequence of up to four such digits will compose a 16-bit value.

- **Binary:** With the prefix `0b`, any 16-bit value. Using only the digits 0 and 1, a sequence of up to 16 such digits.

3.1.3 Labels and branch targets

Any instruction, numeric constant, or text constant (string) may be prefixed with a *label*—a symbolic name that represents the address at which that isntruction or constant will be loaded in memory. The label itself preceeds an instruction or constant, is composed of some unbroken sequence of characters, begins with a letter, and is followed by a colon (`':'`). A label may or may not be on the same line as the instruction or constant to which it corresponds. For example, the following are correct label definitions:

```
    .Code
  L1: MUL 0x20f4 25 @0x339c
  L2:   ; How about a comment?  Blah blah.
     OR 0x1128 0 -1

  .Numeric
array3: 0xffef 0x2322
```

A defined label may be used for any operand. The assembler will translate that label into a word-sized memory address. By default, that memory address is the literal, complete address at which the labeled instruction or constant will be loaded; alternatively, the use of the label may be prefixed by the plus sign (`'+'`), indicating that the assembler should treat the operand as a relative offset from the PC. For example:

```
   JUMP L2
CALL +fib @0xff00
ADD 0x500 1 array3
```

Warning: Using labels for branch targets currently works correctly, but using them for arbitrary, non-branch-target operands, as in the `ADD` example above, does **not** yet work. That is a feature that I will add to the assembler soon.

3.2 Running the assembler

The assembler assumes that your assembly code is written in a file whose suffix is `.asm`, and that it will create a machine code file with the suffix `.vmx`. So, for example, the sample program provided with the assembler is named *add-two-numbers*. Thus, it's assembly code is in the file `add-two-numbers.asm`, and when it is assembled, its machine code will be in the file `add-two-numbers.vmx`.

Invoking the assembler is a simple matter. Here, use it to assemble the simple program provided with the assembler and system:

```
$ java Assembler add-two-numbers
```

That is, the assembler needs only the program name, assuming the assembly file suffix, and creating the output file with the machine code suffix.

Critically, the assembler attempts to provide meaningful error messages for syntactically incorrect assembly code. However, it will only provide a message for the first error that it encounters, and it will then abort assembly. A correctly formed assembly program will be assembled without any error messages—in other words, no news is good news.

3.3 Examing the machine code

Once a machine code file has been generated, you can use Emacs to examine its binary contents. For example, open the machine code file that you just created:

```
$ emacs add-two-numbers.vmx &
```

What you will see is not really human readable. What you want to see is not the ASCII (text) characters to which the byte values correspond, but rather the actual binary values that make up each instruction. To do that, type into Emacs, remembering that `M-x` means `alt-x`:

```
M-x hexl-mode
```

This command will change the display, showing you, with two-digit hexidecimal values for each byte, the underlying binary values in the file. Thus, given the machine code layout described in Section 1.2, you should be able to see the three instructions that make up the *add-two-numbers* program.

4 The AMHCS simulator

The *system* directory contains a complete system simulator, focused on a CPU simulator that implements the AMHCS ISA. Eventually, it will also simulate a hard disk and a display; perhaps later, other devices will also be added. Currently, this software simulates a CPU attached to a memory bus that is connected to a BIOS and a main memory. The BIOS is a simulated ROM whose contents are taken from a user-supplied file. The main memory is a simulated RAM.

4.1 Starting the simulator

To run the simulator, first change into its directory. Assuming that you've been following the instructions above:

```
  $ cd ~/cs26/project-1/assembler
```

To invoke the simulator, you need to provide two pieces of information:

1. **Main memory size:** You must indicate, as a number of bytes, the size of the RAM that this simulated system will have. For our purposes, a small, 1 KB memory will be plenty. I commonly specify 4 KB, just to be sure that I have more than enough. In later projects, we may possibly need a larger value.
2. **BIOS image pathname:** The name of a file that contains the binary image of a BIOS. Specifically, this file should contain executable machine code, since its contents will be the first set of instructions that the CPU will attempt to execute.

Note that, until we've implemented an operating system that we can boot, we use whatever program we want to run as the BIOS, thus running it immediately and directly on our CPU. For example, to run *add-two-numbers*, invoke the simulator like so:

```
  $ cp ../assembler/add-two-numbers.vmx .
  $ java VirtualSystem 4096 add-two-numbers.vmx
```

The simulator will load and present a prompt:

```
  [pc = 0x0000]:
```

This prompt always shows the current value of the *program counter (pc)*, which is initialized to address `0`. At this prompt, you can examine or change any of the system's state—specifically, any memory location, or any CPU state register. You can also control progression of the CPU's execution. To see the list of valid commands, use the `help` command:

```
[pc = 0x0000]: help
Commands:
  help
  step         <number of steps>
  peek         <hexidecimal memory address>
  poke         <hexidecimal memory address> <word value>
  showregister <register name [pc|tbr|base|limit|ip|debug]>
  setregister  <register name [pc|tbr|base|limit|ip|debug]> <word value>
  showflag     <flag name [supervisor|vmem]> <flag value [true|false]>
  setflag      <flag name [supervisor|vmem]> <flag value [true|false]
  exit
```

4.2 Setting the debugging level

There are a number of *CPU state values*—that is, logical registers in the CPU that control how the CPU behaves. A number of these—`tbr`, `base`, `limit`, `ip`, `supervisor`, and `vmem`—we will ignore for now. Later, when we attempt to implement an operating system, we will need to manipulate those values and flags.

The one value that is worth manipulating for this project is `debug`, which controls the level of debugging output that the simulator will emit. By setting this register to 1, we will induce the simulator to provide a good deal of useful information:

```
[pc = 0x0000] setregister debug 1
```

4.3 Manipulating main memory

The `peek` and `poke` commands allow you to read and write the contents of main memory. You can examine, one word at a time, the four-word instruction to which the PC refers:

```
[pc = 0x0000]: peek 0
@0x0000 = 0x030e
[pc = 0x0000]: peek 1
@0x0001 = 0x0e02
[pc = 0x0000]: peek 2
@0x0002 = 0x0200
[pc = 0x0000]: peek 3
@0x0003 = 0x0000
```

If you want, you may also change any word of memory. However, that should be an unusual operation to perform. For example, you may discover that a program has a bug, and that you can fix the bug by modifying an instruction in-memory, while the program is running. More likely, though, you'll want to stop the program, fix the bug, assemble the correction, and re-run the program.

4.4 Stepping through instructions

You may instruct the simulator to perform any number of instructions before stopping to present the prompt again. You can execute a single instructions like so:

```
[pc = 0x0000]: step 1
DEBUG [0]: [@0x0000] 0x 030e 0200 0000 0005:
                     OR 0x0200 0x0000 0x0005
[pc = 0x0008]:
```

Here, the CPU executes one instruction. The CPU, through debugging output, shows both the machine code instruction and its *disassembly*. It executes the instruction and advances the program counter to the next instruction. You can use the `peek` command, after this instruction, to see that add `0x0200` contains the value `0x0005`:

```
[pc = 0x0008]: peek 0200
@0x0200 = 0x0005
```

You can reset the program counter and execute all three instructions of the program like so:

```
[pc = 0x0008]: setregister pc 0
pc = 0x0000
[pc = 0x0000]: step 4
DEBUG [0]: [@0x0000] 0x 030e 0200 0000 0005:
                     OR 0x0200 0x0000 0x0005
DEBUG [0]: [@0x0008] 0x 030e 0202 0000 fffd:
                     OR 0x0202 0x0000 0xfffd
DEBUG [0]: [@0x0010] 0x 030e 0300 0204 0000:
                     OR 0x0300 0x0204 0x0000
DEBUG [0]: [@0x0018] 0x 0500 0300 0200 0202:
                     ADD @0x0300 @0x0200 @0x0202
```

Notice that if you step beyond the end of the program, the CPU may incur an interrupt. Since we've not installed a trap table with pointers to interrupt handlers, the simulator simply aborts:

```
[pc = 0x0020]: step 1
ERROR in CPU.preserveCPUState(): Interrupt BUS_ERROR while preserving CPU
```

With these commands, you should be able to run and debug your assembled programs. Notice that there is no means by which to print any output—for that we will need an operating system and a simulated output device. Instead, your programs should assign, into some pre-determined location, a *result*. You should then be able to `peek` into that memory location to see if the result is correct.

5 Your assignment

To review basic assembly program structures, and to get familiar with the assembler and the simulator, you must write three small assembly program, assemble them, and run them on the simulator to test them. Here are the three programs that you should write:

- **Conditional:** A program that carries out a simple *if-then-else* structure. Of course, since there is no user input, I should be able to change the values of the variables to make either the *then* branch or the *else* branch be taken. In Java-like code, it should be the assembly equivalent of:

```
x = 5;
y = -3;
if (x < y) {
  z = x;
} else {
  z = y;
}
z = z * 2;
```

- **Loop:** Perform a loop that calculates the n^{th} Fibonacci number. That is, assign n into some location, run a loop that calculates $fib(n)$, and leaves the result in some other location. Comment your assembly code so that it's clear which location contains n, and which contains the result.
- **Procedure and procedure call:** Place your loop code into a procedure named `fib`. (You may also try a recursive procedure, but that's not necessary here.) Noting the CPU starts computation on the first instruction in the assembly file, have it place n in some location (as an argument), and have it `CALL` the `fib` procedure. That procedure should copy its result into some return location, and that return location should **negate** the result (just to prove that it received the result). Again, your assembly code should be commented clearly to indicate at what address n is passed, where the return value of fib is provided, and where the final result of the program is.

6 How to submit your work

Use the `cs26-submit` command to turn in your programs, like this:

```
cs26-submit project-1 conditional.asm loop.asm procedure.asm
```

This assignment is due at **11:59 pm** on **Monday, February 9.**