SYSTEMS II — PROJECT 0
Revision 3 — 2014-Feb-03
Assembly, procedures, and stack management

1 Overview and motivation

For this course, I assume that you have some background with assembly programming. However, before we delve into more complex tasks, this project will serve as a "warm-up." Specifically, you will write an assembly program using the assembler and simulated processor that will serve as the basis of our projects throughout the semester.

The K-SYSTEM is designed for a first exploration of operating system kernel design. It simulates a complete system for which hardware has never existed. The processor's instructions are unrealistically large and easy to program; the console shown as a window, and appears as a memory where byte values are turned into displayed characters; the simulated processor is controlled by you at a command-line so that you may examine the machine's state at every step.

This document introduces the system and shows how to run both its *assembler* (for which you will write code) and its *simulator* (which will run that code). It will detail the *opcodes* that the processor understands, as well as the *physical address space layout* that defines how the processor can communicate with other devices.

2 The K-SYSTEM ISA

The 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]

2.1 Basic concepts

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

- **Word size:** This ISA uses 32-bit/4-byte words. The ALU is 1-word wide, and all main memory addresses are likewise 32-bits each. Consequently, the maximum addressible memory is $2^{32}\ bytes = 4\,GB$.
- **Instruction size:** Each instruction for this ISA is 128-bits/16-bytes/4-words. The instructions are uncommonly large to make programming in this ISA easier. Specifically, each of the operand spaces is 1 word, allowing a programmer to specify full immediate constants or full main memory addresses.
- **Named registers:** There is a small set of named registers—that is, registers that can be specified as operands. Specifically, some of these 8 registers are intended for prescribed purposes, while others are for general use.

[1] Do not confuse this architecture with the *mini-k* ISA that is used in Computer Systems I.

- **Endianness:** This ISA is *big-endian*. That is, the most significant bit, labeled bit number `31`, is the left-most one, while the least significant bit, labelled number `0`, is the right-most.
- **Multiple addressing modes:** Each operand can be one of three types:
  1. **Constant:** The operand is itself a value on which an operation should be performed.
  2. **Direct:** The operand is a main memory address, and the value contained at that address is the one to be used by the instruction.
  3. **Indirect:** The operand is a main memory address whose contents are themselves *another* main memory address—that is, the main memory address itself contains a pointer. It is the datum located at this second main memory address that should be used by the instruction.

2.2 Machine code format

Each machine instruction, which is 128-bits in length, has the following format:

- `[127 - 112]` **Opcode:** An 16-bit value that specifies the operation that the CPU should perform. See Section 2.3 for a complete list of opcodes.
- `[111 - 96]` **Operand flags:** Bits that indicate how the operand values should be interpreted. Specifically, for each operand, there is a group of 4 bits that specify how a given operand should be interpreted. If the bits for a given operand begin at bit number `n` (starting from the most significant bit), then the four bits that describe how to interpret that operand are:
  - `[n]` **Constant:** If this bit is `1`, then the operand is an *immediate constant* that should itself be used by the instruction; if the bit is `0`, then the operand is some kind of *memory location*.
  - `[n - 1]` **Register:** If this bit is `1`, then the operand is the number of a register in the register file; if the bit is `0`, then the operand is a main memory address. Note that this flag is relevant only if the operand is a memory location (and not an immediate constant).
  - `[n - 2]` **Relative:** If this bit is `1`, then the operand value is *IP relative*—that is, the current *instruction pointer* (a.k.a., the *program counter*) is added to the value; if the bit is `0`, the operand is *absolute*, and used without modification.
  - `[n - 3]` **Indirect:** If this bit is a `1`, then the memory location specified by the operand is taken to be *indirect* in that it contains a secondary main memory location, and it is this main memory location's value that is used by the instruction; if the bit is a `0`, then the memory location is taken to be *direct*, meaning that the memory location specified by the operand contains the final value used by the instruction. Note that this flag is relevant only if the operand is a memory location (and not an immediate constant).

  The 16 bits for these operand flags leave room for four of these per-operand groups of 4 bits. Since there is room for only three operands in each instruction, then one of those four is unused. Specifically:

- `[111 - 108]` 0^{th}**/destination:** The flags that specify how to interpret the 0^{th} operand, also known as the *destination operand*.
- `[107 - 104]` 1^{st}**/source A:** The flags that specify how to interpret the 1^{st} operand, also known as the *source A operand*.
- `[103 - 100]` 2^{nd}**/source B:** The flags that specify how to interpret the 2^{st} operand, also known as the *source B operand*.
- `[99 - 96]` **Unused:** These flags are unused and reserved for future purposes.

- `[95 - 0]` **Operands:** A collection of 3 operands that specify input and output values. Specifically:
  - `[95 - 64]` **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.
  - `[63 - 32]` **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.
  - `[31 - 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.

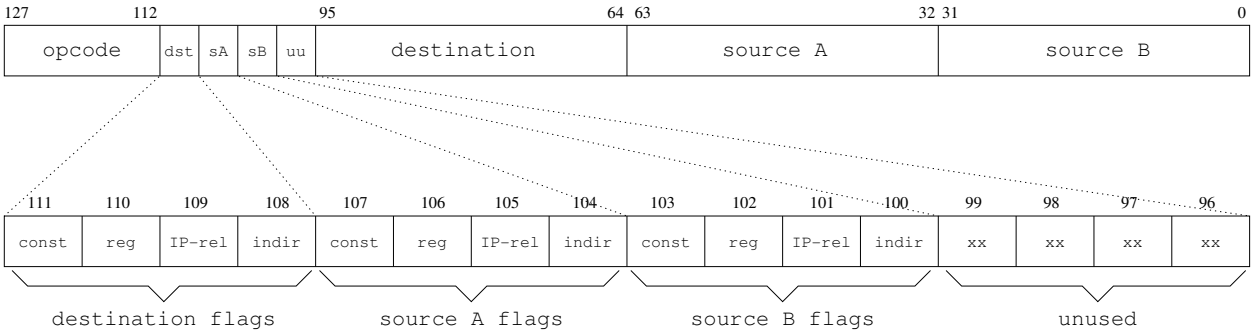

Figure 1: The layout of each 128-bit machine-code instruction.

2.3 Instruction list

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

2.3.1 The empty instruction

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

- `0x0000: NOOP`

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

2.3.2 Arithmetic/logic and memory instructions

These instructions all accept one or two source inputs, perform an arithmetic or logic manipulation, and produce a result to be stored in some memory location. Notice that each main memory address specifies a byte location, but the loading and storing of any word-sized value implies the use of the four-byte sequence that **begins** at the specific main memory address. That is, if address k is specified as the destination to store a word-sized value, then main memory locations $(k, k+1, k+2, k+3)$ will be written.

- `0x0001: NOT` *[destination] [source A]*

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

- `0x0002: COPY` *[destination] [source A]*

  Copy the word value specified by *source A* and store it at the *destination* memory location.

- `0x0003: COPYB` *[destination] [source A]*

  Copy the least significant byte specified by *source A* and store it at the *destination* memory location. Since only a byte-sized value is being stored, only the single main memory address given as the *destination* is written.

- `0x0004: 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*.

- `0x0003: 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*.

- `0x0004: 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*.

- `0x0005: ADD` *[destination] [source A] [source B]*

  Perform signed integer **addition** of the values provided by *source A* and *source B*, storing the result at the *destination*. The integers are assumed to be encoded in two's complement. If the addition operation causes an arithmetic overflow, then an *interrupt*[2] will occur.

[2] A later project will address this ISA's interrupt structure in more detail.

- `0x0006:` `ADDUS` *[destination] [source A] [source B]*

  Perform unsigned integer **addition** of the values provided by *source A* and *source B*, storing the result at the *destination*. Because the values are unsigned, no effort is made to detect overflow, even though it is possible for the sum to exceed the capacity of the word-size, producing an incorrect result. This instruction is intended to be used for addition where an overflow interrupt would be undesirable.

- `0x0007:` `SUB` *[destination] [source A] [source B]*

  Perform signed integer **subtraction** of the values provided by *source A* and *source B*, specifically subtracting the latter from the former, and storing the result at the *destination*. The integers are assumed to be encoded in two's complement. If the subtraction operation causes an arithmetic overflow, then an *interrupt* will occur.

- `0x0008:` `SUBUS` *[destination] [source A] [source B]*

  Perform unsigned integer **subtraction** of the values provided by *source A* and *source B*, specifically subtracting the latter from the former, and storing the result at the *destination*. Because the values are unsigned, no effort is made to detect overflow, even though it is possible for the sum to exceed the capacity of the word-size, producing an incorrect result. This instruction is intended to be used for subtraction where an overflow interrupt would be undesirable.

- `0x0009:` `MUL` *[destination] [source A] [source B]*

  Perform signed integer **multiplication** of the values provided by *source A* and *source B*, storing the single-word result at the *destination*. Notice that although multiplication produces a double-word result, only the less significant word is stored, while the more significant word is discarded. If the multiplication causes *arithmetic overflow*, then an *interrupt* is generated.

- `0x000a:` `MULUS` *[destination] [source A] [source B]*

  Perform unsigned integer **multiplication** of the values provided by *source A* and *source B*, storing the single-word result at the *destination*. Notice that although multiplication produces a double-word result, only the less significant word is stored, while the more significant word is discarded. Because the values are unsigned, no effort is made to detect overflow, even though it is possible for the sum to exceed the capacity of the word-size, producing an incorrect result. This instruction is intended to be used for multiplication where an overflow interrupt would be undesirable.

- `0x000b:` `DMUL` *[destination] [source A] [source B]*

  Perform unsigned integer **multiplication** of the values provided by *source A* and *source B*, storing the double-word result at the main memory location *destination*. Because this instruction stores the complete double-word product, the *destination* cannot be a register; instead, it must be the main memory address of the first of eight bytes into which the product is written. Because the values are unsigned, no effort is made to detect overflow, even though it is possible for the sum to exceed the capacity of the word-size, producing an incorrect result.

- `0x000c:` `DIV` *[destination] [source A] [source B]*

  Perform the signed integer **division** of the values provided by *source A* and *source B* (specifically, $\frac{A}{B}$), storing the single-word result at the *destination*. Notice that overflow cannot occur with single-word integer division. However, if $B = 0$, then an *interrupt* is generated since the result is undefined.

- `0x000d:` `MOD` *[destination] [source A] [source B]*

  Perform the signed integer **modulus** of the values provided by *source A* and *source B* (specifically, $A \pmod B$), storing the single-word result at the *destination*. Notice that overflow cannot occur with single-word integer division. However, if $B = 0$, then an *interrupt* is generated since the result is undefined.

- `0x000e:` `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.

- `0x000f:` `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.

2.3.3 Unconditional branching instructions

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

- `0x0010:` `JUMP` *[destination]*

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

- `0x0011:` `CALL` *[destination] [source A]*

  Like the `JUMP` instruction, **set the instruction pointer** to the target given in the *destination*, whether relative or absolute. Additionally, **store the `IP + 16`**—the address of the instruction that follows the `CALL`—at the memory location given by *source A*.

- `0x0012:` `JUMPMD` *[destination] [source A]*

  Set the instruction pointer to the target given in the *destination* and **change supervisor and/or addressing mode**. The unconditional branching aspect of this instruction is performed in a manner identical to the `JUMP` instruction 2.3.3. However, this instruction also changes the *mode register*, which contain flags that control the behavior of the processor. Specifically, this word-sized register contains two flags, with the remaining bits being unused:

`[0]` **User/Supervisor protection mode:** When this flag is *clear* (that is, it has a value of `0`), then the processor is in *supervisor mode*. When in this mode, certain instructions are enabled. If this flag is *set* (that is, it has the value of `1`), then the processor is in *user mode*, where the aforementioned instructions cannot be used.

`[1]` **Virtual/Physical addressing mode:** When this flag is *clear*, then the processor will use *physical addressing*. That is, each main memory address will be transmitted to the device bus (on which the memory devices reside) unmodified, thus assuming that the instructions being executed are using the physical address to which those memory devices respond. If this flag is *set*, then the processor assumes *virtual addressing*. Each main memory address used by the processor is automatically mapped from its virtual address to some corresponding physical address; this conversion is performed automatically by the *memory management unit (MMU)*. More detailed on virtual memory will be provided in a later assignment.

Note that this instruction itself can **only be used in supervisor mode**. If the processor attempts to execute this instruction while in user mode, an interrupt will occur. Note also that upon any interrupt, the processor automatically clears the user/supervisor flags, thus putting itself into supervisor mode. The handling of interrupts, and more information on user-vs-supervisor modes will be provided in a later assignment.

2.3.4 Conditional branching instructions

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

- `0x0013: 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 instruction pointer** to the target specified by the *destination*.

- `0x0014: 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 instruction pointer** to the target specified by the *destination*.

- `0x0015: 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 instruction pointer** to the target specified by the *destination*.

- `0x0016: 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 instruction pointer** to the target specified by the *destination*.

- `0x0017: 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 instruction pointer** to the target specified by the *destination*.

- `0x0018: 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 instruction pointer** to the target specified by the *destination*.

2.3.5 Supervisor instructions

These instructions are used to control the state and operation of the processor. They are intended only for use by the kernel, or any program designed to control the hardware (e.g., a hypervisor).

- `0x0019: SETTBR` *[source A]*

  Set the *trap base register (TBR)*. This one-word register takes on whatever value is specified by *source A*. For more information on the purpose and function of the TBR, see Section 5.

- `0x001A: SETIBR` *[source A]*

  Set the *interrupt buffer register (IBR)*. This one-word register stores the address of a buffer into which the CPU can store critical information about an interrupt before vectoring into the kernel. For more information about the processor's interrupt control, see Section 5.

- `0x001B: SETPTR` *[source A]*

  Set the *page table register (PTR)*. This one-word register stores a pointer to the root of a page table that provides the mappings that the *memory management unit* uses to translate *virtual addresses* to *physical addresses* when the *virtual memory system* is active. For this project, we will be using solely physical addresses, so this register's value is irrelevant.

- `0x001C: GETCLK` *[destination]*

  Read the *cycle counter register*. This two-word register is reset to zero when the processor is initialized, and then increments with each clock cycle. Since this processor executes exactly one instruction per cycle, this register stores the number of instructions that have been executed. Because the value is two words, the destination operand must specify a main memory location.

- `0x001D: SETALM` *[source A] [source B]*

  Set the *clock interrupt alarm register*. When the cycle counter register matches the value in this two-word register, a *clock interrupt* is generated. Because this alarm register is a two-word value, *source A* be indirect, providing a main memory location from which the full two-word value is taken. Moreover, if *source B* is 0, then *source A* is an absolute value that is copied into the alarm register; otherwise, *source A* is a *clock offset* to be added to the current cycle counter register before being written to the alarm register.

2.3.6 Atypical flow control instructions

These instructions control the flow of a program, but in non-standard ways. That is, they are not part of the normal construction of conditional statements, loops, or procedure calls.

- `0x001E: SYSC`

  An **intentionally invalid opcode** that is reserved so that user-level processes may trigger an interrupt into order to intentionally vector into the kernel, yet indicating to the kernel itself that the interrupt was not the result of an error. For more informatio on the processor's interrupt control, see Section 5.

- `0x001F: HALT`

  An instruction that leaves the IP unchanged and disables all interrupts. As a consequence, the processor ceases progress: any attempt to fetch-decode-execute another instruction causes no state change, and thus no progress.

3 The K-SYSTEM assembler

Because writing programs in machine code is unpleasant, inconvenient. and often downright unproductive, there is an assembler and corresponding assembly language. 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 K-SYSTEM 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 obtain an example, do the following:

```
$ mkdir -p ~/COSC-261/project-0
$ cd ~/COSC-261/project-0
$ cp ~sfkaplan/public/COSC-261/project-0/add-two-numbers.asm .
$ emacs add-two-numbers.asm &
```

You should see the following file contents:

```
01: ;;; A simple program that adds two numbers, demonstrating a few
02: ;;; of the addressing modes.
03:
04:  .Code
05:
06: ;;; The entry point.
07: __start:
08:
09:  ;; Initialize the stack at the limit.
10:  COPY %SP *+limit
11:
12:  ;; Copy one of the source values into a register.
13:  COPY %G0 *+x
```

```
14:
15:  ;; Allocate a space on the stack for the result.
16:  SUBUS %SP %SP 4        ; 4 bytes per word.
17:
18:  ;; Sum the two values.  In particular:
19:  ;;   src A (%G0): A value taken from a register.
20:  ;;   src B (*+y): A indirect value stored in a static space.
21:  ;;   dst   (%SP): A register that contains a pointer.
22:  ADD *%SP %G0 *+y
23:
24:  ;; Halt the processor.
25:     HALT
26:
27:  .Numeric
28:
29:  ;; The source values to be added.
30: x: 5
31: y: -3
32:
33: ;; Assume (at least) a 16 KB main memory.
34: limit: 0x5000
```

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 (e.g., lines 01, 02, and 06) for comments that begin at the start of the line of text; 2 semicolons (e.g., lines 09 and 12) for comments that begin tabbed to the depth of an opcode; and 1 semicolon (e.g., line 16) for comments that follow an actual line of assembly code.

- **Mode markers:** Lines 04 and 27 set 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.

- **Registers:** Note that there are 8 registers in this ISA. Each can be specified with a leading percentage sign (`%`), followed by a symbolic name. Each such name is mapped, by the assembler, to a unique integer identifier for that register. Specifically, the register names, their corresponding values, and their intended uses are:

Symbolic name	Integer ID	Intended use
%SP	0	*Stack pointer*
%FP	1	*Frame pointer*[3]
%G0	2	*General purpose*
%G1	3	*General purpose*
%G2	4	*General purpose*
%G3	5	*General purpose*
%G4	6	*General purpose*
%G5	7	*General purpose*

- **Labels:** These are symbolic markers that the assembler ties to a particular memory location. For example, `x` is defined on line 30 by beginning the line with that name, and following it immediately with a colon (`:`). Thus, the address at which the integer constant `5` will be loaded into main memory is used wherever `x` appears elsewhere in the code. On line 13, `x` is used as an operand. When the assembler translates this instruction into machine code, it will use the address associated with `x` to form the *source A* operand of that instruction.

- **IP relative offsets:** Many labels, when used as operands, are prefixed with a plus sign (+) (e.g., lines 10, 13, and 22). Doing so tells the assembler to translate that operand as being *IP relative*. That is, when the processor executes an instruction that has a IP-relative operand, it will take the value of that operand and **add the current IP value to it**. Here, the addresses associated with labels are expressed as IP relative because the assembler cannot know at what starting address the machine code will be loaded. Thus, by expressing those addresses as offsets from the IP, that starting address is made irrelevant. Later we will see instances where labels can be used as *absolute* addresses, but for now, they all should be IP relative.[4]

- **Indirection:** Some operands are marked as *indirect* by using an asterisk ($\star$) prefix, such as on lines 10, 13, and 22. Doing so tells the processor that the desired value or location to be used for that operand is not the expressed register or address itself, but rather the main memory location stored within that register or address. For example, on line 22, the use of `*%SP`) implies that the result of the `ADD` instruction should be stored not in the stack pointer register itself, but rather at whatever main memory address is contained in the stack pointer register.

There are more details to address here, and so the following sections will explain some of them more carefully.

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.

[4]While we use physical addressing, the use of IP-relative offsets is necessary. When we later introduce virtual memory, each program can be loaded into its own space, and therefore be assigned a fixed loading address, thus allowing the use of absolute addresses.

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 0x10e3e39a
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
MSG1: "The quick brown fox jumps over the dazy log\n"
MSG2: "(spoonerism intentional)\n"
```

3.1.2 Word values

In any place where word-sized values are expected, the assembler allows three methods for specifying these 32-bit values:

- **Decimal:** With no prefix, 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 32-bit value. Since each each hexidecimal digit (0 to 9, A to F) corresponds to four bits, then any sequence of up to eight such digits will compose a 32-bit value.
- **Binary:** With the prefix `0b`, any 32-bit value. Using only the digits 0 and 1, a sequence of up to 32 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     %G0     25    *%G1
L2:   ; How about a comment?  Blah blah.
      COPY    %FP     0xdeadbeef

.Numeric
array3: 0xffef 0b1001001 -82 25
```

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 IP. For example:

```
JUMP    L2
CALL    +fib   %G3
ADD     0x500  1     array3
```

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`. Consider the example starting in Section 3.1, where you obtained and examined a copy of `add-two-numbers.asm`. To assemble this source code, do the following:

```
$ k-assembler add-two-numbers.asm
```

The assembler should show, as debugging output, what it translated. Moreover, it should create an *executable file* of machine code named `add-two-numbers.vmx`. Although you will not see them for this example, 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:

```
$ hexedit add-two-numbers.vmx
```

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 2.2, you should be able to see the three instructions that make up the *add-two-numbers* program.

4 The K-SYSTEM simulator

Once you have assembled machine code, you may use the *system simulator*—a program that performs all the actions of all of the hardware components in a computer system. In doing so, software meant to run on such a hardware system can instead be run on the simulator, with that software not being able to tell the difference. Our system simulator comprises the following components:

- **Central Processing Unit (CPU):** A single datapath and control that fetches, decodes, and executes the instructions of the programs run on the system. This CPU carries out the K-SYSTEM ISA as described in Section 2.
- **Memory/peripheral bus:** A centralized medium to which all other devices are connected and through which they communicate.
- **Bus controller:** The device that provides a map of all of the devices on the bus (see Section 4.1 for more details). In real systems, this device is also the *arbitrator*, controlling the use of the shared medium to avoid collisions.
- **Read Only Memory (ROM):** Memory units with pre-assigned contents that cannot be altered. On our simulated system, each ROM is defined by a file in the host (non-simulated) system. The size and contents of that file are taken as the size and contents of the ROM. A system may have many ROMs. By convention the first ROM is taken as the *Basic Input/Output System (BIOS)*. Consequently, the program counter (IP) in the CPU resets to the starting address of the first ROM on the assumption that it contians the first instruction for bootstrapping the system.
- **Random Access Memory (RAM):** A memory unit whose contents have no defined initial value, and whose contents can be read and written. Typically, there will be only one RAM device per system, although in principle there could be many.
- **Console:** A two-dimensional text display. You are unlikely to want to use this device right away, but later it will be valuable.
- **Hard disk:** A persistent storage device. Backed by a file on the host (non-simulated) system, its contents can be read and written through signalling between the CPU and the device, transferring larger blocks of data. *Warning:* This device is not yet written and does not yet appear on the simulated system by default as yet.

4.1 The bus controller

The bus controller provides a window into the placement of bus devices into the physical address space. Each device has a *type* (controller (1), ROM (2), RAM (3), etc.), and each device has physical *base* (the address of first valid byte to which that device responds) and *limit* (the address of first byte **after** the **last** valid byte to which the device responds). The controller provides access to a *device table* that contains this information for every device on the bus. The devices are typically laid out in the physical address space as shown in Figure 2.

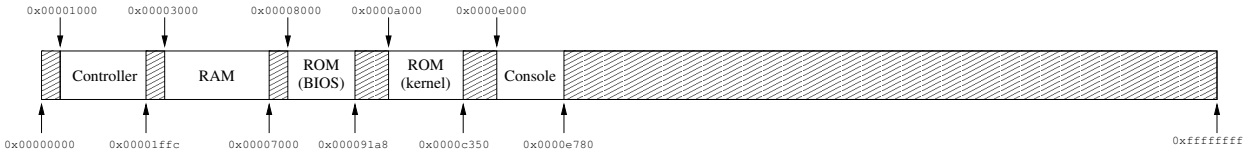

Figure 2: An example placement of devices in the physical address space.

The bus controller is always loaded into the physical address space with a base of `0x00001000`—the address of the first byte in the physical address space. This addres is also the start of an array of values, where each sequential triplet of words is, respectively, the *type*, *base*, and *limit* of a bus device. For example, the set of devices shown in Figure 2 would yield the following device table:

```
0x00001000: 1
0x00001004: 0x00001000
0x00001008: 0x00001ffc
0x0000100c: 3
0x00001010: 0x00003000
0x00001014: 0x00007000
0x00001018: 2
0x0000101c: 0x00008000
0x00001020: 0x000091a8
0x00001024: 2
0x00001028: 0x0000a000
0x0000102c: 0x0000c350
0x00001030: 4
0x00001034: 0x0000e000
0x00001038: 0x0000e780
0x0000103c: 0
...
0x00000ff8: 0
```

So, `0x0000100c` through `0x00001017` contain three word values. The first, at address `0x0000100c`, indicates that this is a RAM device (where `2` would indicate a ROM, and `1` would indicate a bus controller). The second field, at address `0x00001010`, indicates that the base address for the RAM is `0x00003000`. Finally, the third field, at `0x00001014`, indicates that the limit of this device is `0x00007000`. The next three words, from `0x00001018` to `0x00001023`, contain these same three fields for the ROM assumed to contain the BIOS. An entry whose type field is `0` is an empty entry (e.g., at address `0x0000103c`), and indicates that no more meaningful entries exist beyond this point.

4.2 Starting the simulator

To run the simulator, you first need some kind of assembled machine code for the simulator to run. Let's ass that you have followed the `add-two-numbers` example from Section 3.2,

and thus have an *executable image*—a file of machine code that can be loaded and run—named `add-two-numbers.vmx`. To then invoke the simulator, you use the following command:

```
$ k-simulator add-two-numbers.vmx
```

The simulator will start. First, it will show, as *debugging output*,[5] the list of devices that are connected to the bus and the address ranges to which each device responds. Second, the simulator presents you with a prompt:

```
[ip = 0x00008000]:
```

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

```
[ip = 0x00008000]: help
Commands:
  help
  step         <number of steps>
  until        <breakpoint address>
  peek         <hexidecimal memory address>
  poke         <hexidecimal memory address> <word value>
  showregister <[register name %<register number (0 - 7)] |
                [ IP | MD | TB | PT | IB | DB | CK | AL ]>
  setregister  <[register name %<register number (0 - 7)] |
                [ IP | MD | TB | PT | IB | DB | CK | AL ]>
               <word value>
  showconsole
  exit
[ip = 0x00008000]:
```

We will now explain what many of this commands do, although we will not address every detail of each one.

4.3 Setting the debugging level

There are a number of *CPU state values*—that is, registers in the CPU that control how the CPU behaves. A number of these—`TB` (trap base), `IB` (interrupt buffer), `MD` (mode)—we have discussed in class or have been presented along with instructions that manage them in Section 2.3. We focus here, though, on a register that is not part of the K-SYSTEM ISA, but just part of this implementation: the `DB` (*debug*) register, 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:

[5] This output is provided both as an aid to me for debugging the simulator and as an aid to you in debugging your programs. Section 4.3 shows how to increase or decrease the amount of debugging output shown.

```
[ip = 0x00008000] setregister DB 1
```

The extra debugging output can be eliminated by resetting this register to its default, 0. The register can also be set to higher values, but any value larger than 1 will currently yield an erratic collection of output used for debugging the simulator, so I don't recommend it.

4.4 Manipulating main memory

The `peek` and `poke` commands allow you respectively to read and write the contents of main memory. You can examine any valid address using `peek`:

```
[ip = 0x00008000]: peek 0xa000
@0x0000a000 = 0x00024200
```

If you want, you may also change any word of memory by using `poke`. 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. In case you wish to use this command, it looks like:

```
[ip = 0x00008000]: poke 0x3000 0xdeadbeef
@0x00003000 = 0xdeadbeef
```

4.5 Stepping through instructions

You may instruct the simulator to perform any number of instructions—to execute a number of *steps*—before stopping to present the prompt again. You can execute any constant number of instructions in a row. For example, to execute three instructions:

```
[ip = 0x00008000]: step 3
DEBUG [0]: <0>[@0x00008000] 0x 00024200 00000000 00000058 00000000:
           COPY %0x00000000 @+0x00000058
DEBUG [0]: <1>[@0x00008010] 0x 00024200 00000002 00000040 00000000:
           COPY %0x00000002 @+0x00000040
DEBUG [0]: <2>[@0x00008020] 0x 000a4480 00000000 00000000 00000004:
           SUBUS %0x00000000 %0x00000000 0x00000004
```

Here, the CPU executes three instructions. The CPU, through debugging output, shows a great deal of information about what happened:

- **The cycle counter:** The number provided in angle brackets (e.g., `<2>`) indicates the value of the cycle counter register as this instruction was executed. Since one instruction is performed per clock cycle, this value also doubles as an indication of the number of *steps* that have been taken since the start of the program.
- **The IP:** Shown in square brackets (e.g., `[@0x00008020]`), the debugging output shows the address from which the instruction was fetched—that is, the value of the program counter.

- **The machine code:** Next, the four-word machine-code instruction is provided as a 32-digit hexidecimal value. The exact bits, as read from the executable image file (e.g., `add-two-numbers.vmx`, are shown here for your manual verification or inspection.
- **Disassembly:** The simulator disassembles the machine code instruction. That is, it converts the machine code back into an assembly form, thus approximating the assembly source code that you may have written.

The simulator also allows you to execute instructions until the IP takes on some particular value, effectively executing until a *breakpoint* in encountered:

```
[ip = 0x00008000]: until 0x8040
DEBUG [0]: <0>[@0x00008000] 0x 00024200 00000000 00000058 00000000:
           COPY %0x00000000 @+0x00000058
DEBUG [0]: <1>[@0x00008010] 0x 00024200 00000002 00000040 00000000:
           COPY %0x00000002 @+0x00000040
DEBUG [0]: <2>[@0x00008020] 0x 000a4480 00000000 00000000 00000004:
           SUBUS %0x00000000 %0x00000000 0x00000004
DEBUG [0]: <3>[@0x00008030] 0x 00075420 00000000 00000002 00000024:
           ADD *%0x00000000 %0x00000002 @+0x00000024
```

5 Interrupts

5.1 Codes

For a kernel to establish control of the CPU and the hardware overall, it must establish its procedures as the ones to call when CPU interrupts occur. The K-SYSTEM ISA enumerates the following interrupts:

0. `INVALID_ADDRESS`: Some operand specified a memory address that is invalid. Typically used when an invalid or impermissible *virtual* address cannot be translated.
1. `INVALID_REGISTER`: Some operand specified a register number that is invalid.
2. `BUS_ERROR`: An operand provided an address that yielded an address on the bus that was invalid. The bus may have received an address for which there is no responding device, or the bus may have refused to process a misaligned address.[6]
3. `CLOCK_ALARM`: A periodic alarm generated when the *cycle counter* matches the *alarm register*. (See the `SETALM` instruction in Section 2.3.5.)
4. `DIVIDE_BY_ZERO`: Occurs when one of the arithmetic division instructions receives a denominator operand whose value is zero.
5. `OVERFLOW`: Occurs when a *signed* arithmetic operation yields an overflowed result.

[6] A 32-bit system will expect any request for a word-sized value (e.g., **not** a `COPYB` instruction) to be *word-aligned*—that is, the address A, given 4-byte words, should satifying the property that $A\%4 \equiv 0$.

6. `INVALID_INSTRUCTION`: If an instruction contains an invalid opcode, or if an operand has invalid status bits, then this interrupt occurs.

7. `PERMISSION_VIOLATION`: A supervisor-only instruction (see Section 2.3.5) was issued while the processor was in user mode.

8. `INVALID_SHIFT_AMOUNT`: When one of the arithmetic *shift* instructions is used, the number of bits to shift can be no more than the word size.

9. `SYSTEM_CALL`: A special case of the `INVALID_INSTRUCTION` interrupt reserved for the use of a particular invalid opcode used for system call vectoring.

5.2 Vectoring

When an interrupt occurs, the processor performs a specific sequence of steps:

1. **Elevate mode:** Set the processor into supervisor mode.

2. **Preserve state:** Store into the *interrupt buffer* the program counter and any auxiliary information about the interrupt (e.g., the virtual address that triggered an `INVALID_ADDRESS` interrupt). The interrupt buffer is a main memory space—at least two words in size—that the processor finds via the *interrupt buffer register*. (See the description of the `SETIBR` instruction in Section 2.3.5.)

3. **Vector to interrupt handler:** The processor uses the interrupt code to find the corresponding handler procedure. Specifically, the *trap table* is an array of pointers to the entry points of interrupt handler procedures. The processor looks up the correct entry in this table by calculating ...

   $$t_e = t_b + c|w|$$

   ...where t_e is the address of correct trap table entry, which is obtained from the t_b *trap base*—the starting address of the trap table—as well as c, the interrupt code, and $|w|$, the word size. In short, the interrupt code is an *index* into the array of word-sized addresses.

   Once the handler finds the correct table entry address (t_e), then it can grab the value from that address, thus pulling the address to which the processor then jumps in order to handle the interrupt.

6 Your assignment

Your goal with this assignment is to write, in assembly, the code needed to bootstrap our simulated, hard-disk-less system. Doing so requires you to carry out two tasks:

1. **Write a BIOS:** When assembled, this executable image should be provided to the simulator first on the command line so that it is loaded as the BIOS and executed first. It should perform a few critical tasks to get the kernel running:

(a) Read the bus controller map (see Section 4.1) to find both the *second ROM* (assumed to be the kernel) and the *RAM*.

(b) Copy the contents of the second ROM into RAM.

(c) `JUMP` to the first kernel instruction in RAM.

2. **Write a kernel that initializes the system:** The executable image, provided as the second ROM to the system, will be loaded and called by the BIOS. Although later it will perform a larger number of more complex tasks, for now, it needs only to assert control over the hardware, like so:

   (a) Find a new, unused portion of RAM (that is, part of RAM not storing the executable image of the kernel itself).

   (b) Create a *trap table*, and load it with the addresses of interrupt handler procedures. For whichever interrupts the kernel does not handle in any specific way—initially, all of them—insert the address of a *null interrupt handler* that simply halts the system.

   (c) Install the trap table by using the `SETTBR` instruction.

   (d) Load a single user-level program—here, the `add-two-numbers` program will do—from the executable image in the third ROM.

   (e) Jump into that user-level program, exiting supervisor mode by using the `JUMPMD` instruction.

In later projects, we will substantially augment this kernel. However, its initialization routine is likely to stay largely unchanged.

7 How to submit your work

Use the CS submission systems to submit your work. Specifically, you will need to submit your `bios.asm` and `kernel.asm` files.

This assignment is due at **11:59 pm** on **Sunday, February 16.**