SYSTEMS II — PROJECT 4
Bootstrapping our simulated system

1 Updates to the ISA, assembler, and simulated system

The ISA with which we began in Project 1 was a good start, but it had critical limitations. This project includes modifications to the ISA, the system simulator, and the assembler, all of which are described below:

1.1 Basic concepts

Indirection Of the basic concepts presented in Project 1, one of them has been extended. In particular, having *direct and indirect operands* was useful, but insufficient. The newly updated version of the ISA allows each operand to be:

- *Direct:* As before, the operand's immediate value, in the instruction, is used.
- *Singly indirect:* In a manner equivalent to *indirect* operands in the original ISA, the operand's immediate value is taken as a memory address, and the value located in that address is used. That is, a single level of indirection is employed in processing this operand for its instruction.
- *Doubly indirect:* For this new extention to the ISA, an operand's immediate value is again taken as a main memory address. Furthermore, the value found at that address is **also** taken to be a main memory address. The CPU will follow both of these addresses in sequence, obtaining the value at the second memory address via double indirection. This type of indirection is essential for programs to be able to follow pointers stored in main memory.

PC-relative values: In the previous ISA, the labels used for the destination operand of a branching instruction could be a represented, in the machine code, as an absolute value or a PC-relative one. For example . . .

```
JUMP foo
```

. . . stores the branch target as an *absolute address* to which the PC should be set when this instruction is executed, whereas . . .

```
JUMP +foo
```

. . . stores the branch target as a *PC-relative offset*—that is, the PC will be set to the value obtained by adding the PC to the destination operand's value when this instruction is executed.

This updated version of the ISA and its corresponding assembly code now allows **any** operand to be either an absolute or a PC-relative value. In particular, for any operand, you may specify a label name, and you may then refer to that label's address as an absolute address or a PC-relative

one. Note that you may also dereference, singly or doubly, any absolute or PC-relative value, just as you would any other operand value.

Why is this feature useful? When virtual addressing—here, base-offset addressing—is used, the two are equivalent. However, for code that runs in supervisor mode, where only physical addressing is used, the two can be quite different. For example, consider the following line of assembly code:

```
infloop:  JUMP infloop
```

The label `infloop` will have an underlying value of corresponding to the address at which this instruction will be loaded **if its executable image is loaded at address** `0x0000`. That assumption—that the executable image is loaded at address `0x0000`—is correct for code running in user mode with virtual addressing active, since each process's executable image is indeed loaded at its base address in the physical memory space.

However, for code that runs with physical addressing, such as the BIOS and the kernel, its executable images are not loaded at `0x0000`; in fact, RAM begins at that physical address (see Section 1.5. So, the value for `infloop` calculated by the assembler **will be incorrect**—it will not be the actual address at which the corresponding instruction will be loaded. However, if we write the branching instruction like this . . .

```
infloop:  JUMP +infloop
```

. . . then the assembler will emit `0x0000` as a PC-relative offset. That is, when this instruction is executed, the CPU will add `0x0000` to the current PC, and load the result into the PC, correctly looping back to the same instruction. PC-relative offsets are insensitive to location at which code is loaded, making it appropriate for physical addressing use, under which a programmer cannot know where her code will be loaded.

The ability to use this PC-relative labels for any operand will additionally be useful when accessing *statically allocated spaces*, described in Section 1.4. For the same reason, use of these spaces in physical addressing mode in made possible when they are used as PC-relative offsets.

1.2 Machine code format

The 64-bit machine code instruction format is somewhat modified in this new formulation. In particular, because of the addition of doubly-indirect and pervasive PC-relative offset operands, more operand flags were required in each instruction. The format, as show in Figure 1, is now:

- `[63 - 57]` **Opcode:** An 7-bit value that specifies the operation that the CPU should perform. This field used to be 8 bits, but needed to be reduced to make space for additional operand flags.
- `[56 - 48]` **Operand flags:** A collection of 9 boolean values that specify how the operand values should be interpreted. Specifically:

- `[48 - 50]` **PC-relative/absolute values:** Set if the **destination/source A/source B** (respectively) is a *PC-relative value*, unset if it is an *absolute value*.
- `[51 - 53]` **Direct/indirect value:** Set if the **destination/source A/source B** is a *direct value*, unset if it is an *indirect value*.
- `[54 - 56]` **Singly/doubly indirect value:** Set is the **destination/source A/source B** is a *singly indirect value*; unset if it is a *doubly indirect value*. For each operand, its singly/doubly indirect bit is used only if its direct/indirect bit is unset (indicating an indirect value).

• `[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. For instructions that branch, this operand contains the branc target.
- `[31 - 16]` **Source value A:** For instructions that require at least one input value, the first such value.
- `[15 - 0]` **Source value B:** For instructions that require two input values, the second such value.

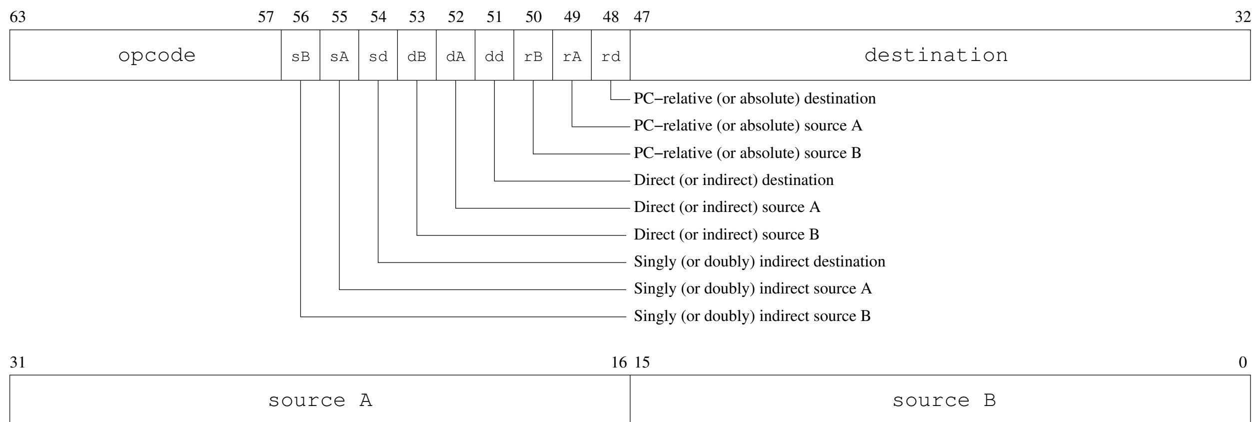

Figure 1: The layout of each 64-bit machine-code instruction.

1.3 Additions to the instruction list

In addition to the instructions presented in Project 1, a few have been added. Specifically, there is a category of instructions, previously unlisted, that are available only in supervisor mode. They are presented here, broken down by category.

1.3.1 Unconditional branching instructions

Unconditional branching instruction alter the program counter without testing or comparing any state. In addition to the previously defined `JUMP` and `CALL` instructions, there is an additional one used only in supervisor mode

- `0x18: EXSUP` *[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.

  Additionally, unset the *supervisor mode flag*, placing the processor into user mode. This instruction is the only one that places the process in user mode, and is intended to allow a kernel to jump into user-level code while reactivating virtual addressing and turning off supervisor privileges.

1.3.2 CPU status setting instructions

These instructions are available only in supervisor mode, and they are used to set the CPU's status flags. It is these status flags that define the operating state of the CPU at any given moment.

- `0x13: SETTT` *[source A]*

  Copy *source A* into the *trap base register*. This value is taken as a pointer to the *trap table*. This table is used by the CPU upon an interrupt to branch into kernel code. The address placed in the TBR, as well as the addresses placed in the trap table, should all be absolute physical addresses, since an interrupt places the CPU into supervisor (and thus physical addressing) mode.

- `0x14: SETBS` *[source A]*

  Copy *source A* into the *base register*. This value is the base physical address that the virtual memory address transator should use. That is, in virtual addressing mode, this value is added to every virtual address to produce a physical address.

- `0x15: SETLM` *[source A]*

  Copy *source A* into the *limit register*. This register is the first *invalid* address, following the base, that a user-level process may not reference. In virtual addressing mode, after the base is added to a virtual address, the resulting physical address is compared to this limit. If the physical address matches or exceeds the limit, an `INVALID_ADDRESS` interrupt occurs.

- `0x16: SETIP` *[source A]*

  Copy *source A* into the *instruction preserve register*. This value is taken as a pointer to the *instruction preserve region*—a physical address to a main memory space into which the CPU will copy critical values upon an interrupt. Currently, this space needs to be one word long, and the CPU copies the PC into this space before jumping into the interrupt handler specified by the trap table. The kernel can then grab the old PC value from this preserve space so that it may return to that location after completing the interrupt handling.

- `0x17:   SETVA` *[source A]*

  If *source A = 0*, then place the CPU in *physical addressing mode*; otherwise, place the CPU in *virtual addressing mode*. Note that virtual addressing applies only in user mode, not in supervisor mode.

1.3.3 Psuedo-instructions

The following instruction is not really an instruction at all, but an easy way of specifying, in assembly source code, something that can be translated into a useful part of the executable image that will trigger a desired result.

- `SYSC`

  This instruction, which uses no operands, generates a instruction value with an invalid op-code. It will therefore trigger an `INVALID_INSTRUCTION` interrupt, transferring control to the kernel-level procedure specified in the trap table for that interrupt type.

1.4 Using statically allocated numeric constants

Recall that our assembler can also allow us to specify statically allocated numerical values. These spaces become part of the executable image that is produced. Thus, when a process is initiated and the executable image loaded, these spaces are immediately available for the process to use, with preloaded values as specified in the assembly code. Moreover, you can mark these constants with labels, providing names for these spaces that you can use in your code.

As an example, consider a program that adds two numbers, where the two source values and the destination are all statically allocated spaces. Note the use of the `.Code` and `.Numeric` markers in the assembly code that tell the assembler how to interpret each portion of the source:

```
        .Code

;;; Peform x = y + z
ADD x   @y   @+z

        .Numeric

x:   0
y:   0x5
z:   -13
```

First, notice that the three labels, `x`, `y`, and `z`, refer each to one-word spaces whose initial values are `0`, `0x5`, and `-13`, respectively. Note that each labeled space can be an arbitrary number of words in length, where the label will be the address of the first word. Thus, you can create and assign multi-word values, or arrays of word-sized values—their type will depend only on how the code uses those spaces.

Second, notice how these labels are used by the `ADD` instruction. Each label is resolved by the compiler into its corresponding main memory address. For example, `y` will be replaced by the offset from the beginning of executable image at which `0x5` will be loaded—that is, it is an address relative to the beginning of the executable image. Therefore, specifying `x` as the target indicates that the initial `0` value at the space named by `x` will be replaced with the result of the addition operation when it is performed at run-time.

For the *source A* operand, `y` will also be resolved to its respective address. However, we do not want to add the **address** of `y` to something; rather, we want to add the **value stored at that address** to something. So, we use the single-dereference operator (`@`) so that the CPU will read the word stored at the address given by `y`, and use that value as an input to the addition operation.

Finally, the *source B* operand looks a bit different from the *source A* one. For *source A*, the address corresponding to `y` will be stored in the machine-code word as an *absolute address*. By additionally using the PC-relative operator (`+`), the *source B* operand will contain a PC-relative offset to refer to the space labeled by `z`. When this operand is evaluated by the CPU, it will first add that offset to the PC, and then it will read the value at that resulting location. This value is then used as the second input to the addition operation. Recall that PC-relative offsets are particularly useful for code that will run in physical addressing mode—code for which the programmer cannot know where the executablke image will be loaded into the address space.

1.5 Bus devices and physical address placement

Previously, the executable image specified at the simulator's command line was loaded, as a ROM, at physical address `0x0000`, with RAM loaded at some address that follows. We wrote programs that used some sufficiently likely RAM addresses, but that's hardly sufficient for building an entire system.

A new arrangement In this version of the system simulator, the bus (implemented by the `Bus` class), which is responsible for placing the various devices within the physical address space, performs this task differently. First, the command-line interface to the simulated system has been slightly modified to allow the user to specify multiple executable images, each of which will be loaded as a separate ROM, like so:

```
$ java VirtualSystem 8192 image-1.vmx image-2.vmx image-3.vmx
```

Given this specification, the bus will perform a number of tasks that result in a layout shown in Figure 2:

1. Create an 8 KB RAM and place it such that its address range is from `0x0000` to `0x2000`.
2. Create three ROM's and place them, in order, in the physical address space following the RAM. Each ROM will have the exact size of the executable image itself. Moreover, there will be a *guard band*—some small number of addresses in the address space that are unallocated to any device, and thus separate one device from another with addresses that cannot be validly used.

3. Place a *bus controller* at the top of the physical address space. The controller provides information about which devices are connected to the bus, and the address ranges to which those devices respond. More information on how to use the bus is given below.

What is critical about this reorganization is that two devices—the RAM and the controller—are always placed at preditable locations at the respective beginning and end of the physical address space. Thus, BIOS or kernel code can be written with physical addresses that will access these two devices to carry out their tasks.

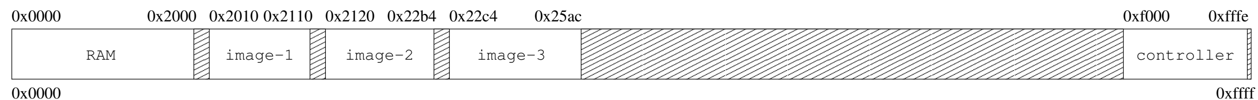

Figure 2: An example placement of devices in the physical address space.

The controller The bus controller provides a window into the placement of bus devices into the physical address space. Each device has a *type* (controller, ROM, or RAM), and each device has physical *base* and *limit* addresses. The controller provides access to a *device table* that contains this information for every device on the bus.

The bus controller is always loaded into the physical address space with a limit of `0xfffe`—the address of the last word in that space. Thus, at address `0xfffc`, the controller provides access to a *device table pointer*, which is the address at which the device table begins.

The device table is an array of values, where each sequential triplet of words is, respectively, the *type*, *base*, and *limit* of a bus device. For example, assume that the device table begins at address `0xf000`. Therefore, if we use a guard band of 16 bytes, and if we assume the 8 KB RAM followed by three ROM's specified in the example above, the contents of this table will be:

```
0xf000: 3
0xf002: 0x0000
0xf004: 0x2000
0xf006: 2
0xf008: 0x2010
0xf00a: 0x2110
0xf00c: 2
0xf00e: 0x2120
0xf010: 0x22b4
0xf012: 2
0xf014: 0x22c4
0xf016: 0x25ac
0xf018: 1
0xf01a: 0xf000
0xf01c: 0xfffe
0xf01e: 0
0xf020: 0
```

```
0xf022: 0
```

So, 0xf000 through 0xf005 contain three word values. The first, at address 0xf000, 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 0xf002, indicates that the base address for the RAM is 0x0000. Finally, the third field, at 0xf004, indicates that the limit of this device is 0x2000. The next three words, from 0xf006 to 0x0f00b, contain these same three fields for image-1.vmx. An entry whose type field is 0 is an empty entry, and indicates that no more meaningful entries exist beyond this point.

2 Your assignment

2.1 Obtaining the tools

To obtain the new assembler and simulator, login to the CS systems and follow the steps below. Notice that **there is a second version of this project code**, because the first version lacked some of the features above, making it unsuitable for your tasks described in Section 2.2.

```
$ cd cs26
$ tar -xzvpf ~sfkaplan/public/cs26/project-4-v2.tar.gz
$ cd project-4
$ cd assembler
$ javac *.java
$ javadoc *.java
$ cd ../system
$ javac *.java
$ javadoc *.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.

2.2 Your assignment

Your task can be divided into two parts:

1. **Write a BIOS:** In assembly, write a program that will be loaded into the system as the **first** executable image. Its job is to find the **second** executable image, copy it into RAM, and jump to its first instruction. That is, it should load and execute some other program.

2. **Write a kernel:** Again in assembly, write a program that will be loaded into the system as the **second** executable. It will be loaded into RAM and executed by a BIOS. Its job is to find the **third** executable image, copy it into RAM, set the trap table, interrupt preserve, base, and limit registers, and jump to its first instruction **while switching into user mode**. That is,

this kernel should run a single user-level process. Moreover, when that process completes, it should use the SYSC instruction to generate an interrupt, bringing control back to the kernel, which should clear the base and limit, and then go into an infinite loop.

These two programs will both run in supervisor mode, and they are likely to have chunks of code in common (e.g., the code that searches the device table for a particular ROM). You should also, of course, write a third, super-simple program intended to run in user mode, that the kernel can invoke. When the project is complete, you should be able to run this code like so:

```
$ java VirtualSystem 8192 bios.vmx kernel.vmx application.vmx
```

3 How to submit your work

Use the cs26-submit command to turn in your programs. Specifically, move into your cs26 directory and submit **your entire project directory**, thus allowing me to test your code with any modifications you may choose to make to the assembler or simulator:

```
$ cd ~/cs26
$ cs26-submit project-4 project-4
```

This command will submit, for this project, everything from your project-4 directory.

This assignment is due at **11:59 pm** on **Sunday, February 29.**