Systems II
Fivish system documentation

1 Overview

The *Fivish* system is an implementation of subset of the *RISC-V instruction set architecture (ISA)* designed for undergraduate projects in operating system (OS) design and implementation. This document presents the Fivish assembler and system simulator, including the use of its various devices (e.g., console, block device).

2 The Fivish/RISC-V ISA

Fivish **mostly** implements the *RISC-V RV32I v2.0* instruction set with the *"M" extensions* and standard *pseudo-instructions.* To see these more formally defined, see the *RISC-V instruction set manual*, particularly Chapters 2, 6, and 20.

There are small ways in which it does not fully implement the standard. Specifically, there are common assembly forms and macros that the Fivish assembler does not offer; similarly, the assembler implements some of its own macros to make the management of two-instruction sequence handling of 32-bit constants cleaner. These variations should alter the writing of RISC-V assembly only a little. Likewise, the Fivish CPU and system simulator uses a nonstandard arrangement of *control and status registers (CSRs).* While the instructions opcodes and formats are the same, the details of how those privileged registers are organized and used is specific to Fivish.

We lay out here the details of the Fivish assembler and system simulator, relying heavily on the ways in which it mostly uses the standard RISC-V ISA.

3 Control and status registers

The CSR set of the Fivish simulator is a group of address that controls and behavior of the processor (and the simulator). These are not the standard set of RISC-V CSR's; they are specific to RISC-V. Nonetheless, they can be accessed via the standard CSR instructions (Section 2.8 of the *RISC-V instruction set manual*).

Here is the enumerated list of registers which are accessed by the CSR instructions by number:

0. `pc`: The *program counter*, which points at the current machine code instruction
1. `md`: The *mode register*, whose bits reflect and control the state of the processor. Specifically, the bits (starting from the 0^{th}, or least significant bit) are:

- **Bit 0**—HALTED: `0` when the processor operates normally; `1` when it no longer fetches, decodes, and executes an instruction.
- **Bit 1**—USER: `0` when the processor is in supervisor mode; `1` when it is user mode.[1]
- **Bit 2**—VIRTUAL: `0` when physical addressing is used; `1` to enable the *memory management unit* to perform virtual address translation.
- **Bit 3**—PAGED: `0` to have the MMU perform additive *base/limit* address translation; `1` to direct the MMU to use *paged* translation.
- **Bit 4**—ALARM: `0` the *alarm register* (`al`) is ignored; `1` to generate a `CLOCK_ALARM` interrupt (see Section 4) whenever the *clock register* (`ck`) equals `al`.

2. `tb`: The *trap base register*, which points to the *trap table* used upon an interrupt by the processor to *vector* to a handler function. (See Section 4.2.)

3. `epc`: The *environment program counter*, which contains a copy of the `pc` at the moment an interrupt triggers a vector into the kernel; it is also the source register for the `eret` instruction.

4. `eai`: The *environment auxiliary information*, which stores additional information about the event that triggers an interrupt (e.g., the address that cause an `INVALID_ADDRESS` interrupt).

5. `bs`: The *base register* stores the base address used by the MMU when the processor's mode indicates *user* mode with *virtual, non-paged* address translation.

6. `lm`: The *limit register* stores the limit address used by the MMU for *user* mode with *virtual, non-paged* address translation.

7. `pt`: The *page table register* points to the upper-level page table used by the MMU for address translation when in *paged virtual* mode.

8. `ck`: The *clock register* counts instructions completed.

9. `al`: When *alarm mode* is enabled, this register is compared to the `ck` register, such that `al == ck` triggers a `CLOCK_ALARM` interrupt.

10. `bp`: The *breakpoint register* is compared to the `pc`, and when `bp == pc`, an `EBREAK` interrupt pauses the simulator.

11. `wp`: The *watchpoint register* points to a memory location that, when accessed by the CPU, triggers an `EBREAK` interrupt to pause the simulator.

12. `db`: The *debugging level register* is used by the simulator to control the level of the debugging output generated.

[1] RISC-V defines addition permission levels, but Fivish only implements these two.

4 Interrupts

4.1 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. Fivish 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.[2]
3. `CLOCK_ALARM`: A periodic alarm generated when the *cycle counter* matches the *alarm register*.
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. `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 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.
10. `SYSTEM_BREAK`: Like a system call, except this interrupt is **not** handled by the kernel; instead, it is received by the simulator itself, which pauses its processing to present its own command prompt.
11. `INVALID_DEVICE_VALUE`: A value written to a device is considered invalid by that device (e.g., setting the console's offset control to a value outside the range of the console buffer size).
12. `DEVICE_FAILURE`: When a device cannot complete an operation because of unexpected internal failures (e.g., a storage device that cannot access its memory successfully), it raises this interrupt.

[2] 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 $A\%4 \equiv 0$.

4.2 Vectoring

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

1. **Elevate mode:** Set the processor into *supervisor mode* by clearing the `user` bit in the `md` register.

2. **Preserve state:** Copy the `pc` into the `epc` CSR; also store any auxiliary information about the interrupt (e.g., the address that triggered an `INVALID_ADDRESS` interrupt) into the `eai` CSR.

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 *trap base* t_b (from the `tb` register), the interrupt code c, and the word size $|w| = 4$, in bytes. In short, the interrupt code is an *index* into the array of word-sized addresses.

   To complete the jump, the processor sets `pc` to the value found at address t_e.

5 The assembler

Fivish assembly is primarily RISC-V assembly, so we refer you to documentation on that standard ISA to see how operations, addressing, registers, labels, etc., are expressed. What follows are the Fivish-specific elements of its assembly language.

5.1 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 usual forms (decimal, hexadecimal, binary). 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"
```

5.2 Immediate value modifiers

Many of the RISC-V instructions contain an *immediate value*—a number that is embedded directly into a machine-code instruction. For example, the `addi` instruction adds the contents of a *source register* to the *immediate value*, storing the result in a *destination register*:

```
addi t3, s5, 205  # t3 = s5 + 205
```

The Fivish assembler provides a number of *immediate modifiers* (some standard, some Fivish-specific) to aid in specifying those immediate values:

- `%lo(x)`: Given the value `x`, clear its upper bits, keeping only the *low 12 bits* as the immediate value.

- `%hi(x)`: Given the value `x`, shift it right by 12 bits, keeping only the *high 20 bits* as the immediate value.

- `%add(x,y)`: Given the values `x` and `y`, produce an immediate value that is the result of `x + y`.

- `%and(x,y)`: Given the value `x` and the integer bitmask `y`, produce an immediate value `x & y` (the bitwise AND).

- `%pcrel(x)`: Given the label `x`, produce the immediate value `x - pc` (the *program counter*), which is the distance, at runtime, between the current instruction and the labeled element.

- `%larel(x,y)`: Given the labels `x` and `y`, produce the immediate value `x - y`. That is, produce the distance between elements labeled by `x` and `y`.

Most of these modifiers are employed by the pseudo-instructions that handle 32-bit constants (both values and addresses) via two-step instruction sequences. (See the RISC-V documentation, Section 2.4, on *Integer Register-Immediate Instructions*, particularly `LUI` and `AUIPC`, as well as Table 20.2, to see how these two-instruction sequences operate and use the above modifiers.)

6 The 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.
- **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 6.1 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.

6.1 The bus controller

The device table: 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 1.

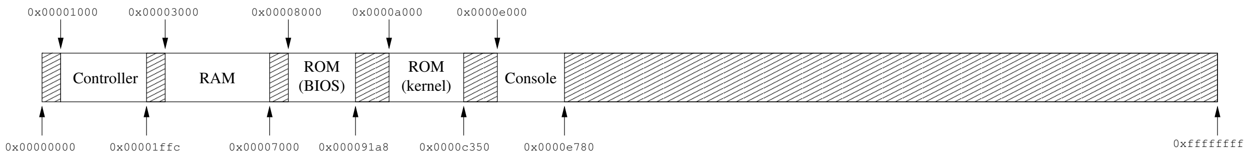

Figure 1: An example placement of devices in the physical address space.

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 1 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
...
0x00001ff8: 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.

The DMA portal: The bus controller also allows for rapid copying of data from one range of physical addresses to another, bypassing participation by the CPU. The *direct memory access (DMA) portal* is a range of addresses that controls this capability.

The portal is a three-word space at the end of the bus controller's address range. So, given the bus controller's `limit`, the three values are:

- The *source address* (at `limit - 12`): The starting address from which to copy.
- The *destination address* (at `limit - 8`): The starting address to which to copy.
- The *length* (at `limit - 4`): The number of bytes to copy. This space is also the *trigger* that initiates the copying action.

At the moment that an instruction stores a word into the *length* space of this portal, the bus controller will immediately copy the bytes from the range `[source, source + length)` to the range `[destination, destination + length)`.

6.2 Starting the simulator

To run the simulator, you may provide as many assembled *executable* files—files produced by the assembler with the suffix `.vmx`—as you like.

```
$ f-simulate bios.vmx kernel.vmx add-two-numbers.vmx do-nothing.vmx
```

The simulator will start. First, it will show, as *debugging output*,[3] the list of devices that are connected to the bus and the address ranges to which each device responds. It also shows the initial state of the *console* device. Then, the simulator presents you with a prompt:

```
[pc = 0x00008000]:
```

This prompt always shows the current value of the *program counter* (`pc`), 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:

```
[pc = 0x00008000]: help
Commands:
  help
  step         <number of steps>
  until        <breakpoint address>
  peek         <hexidecimal memory address>
  peekb        <hexidecimal memory address>
  peekaround   <hexidecimal memory address>
  poke         <hexidecimal memory address> <word value>
  pokeb        <hexidecimal memory address> <byte value>
  unhalt
  showregister <[<register number (0 - 31)] | <register name> | <CSR name> ]>
```

[3] Section 6.3 shows how to increase or decrease the amount of debugging output shown.

```
  setregister  <[<register number (0 - 31)] | <register name> | <CSR name> ]> <value>
  showregisters [ text | graphic ]
  showconsole [ text | graphic ]
  exit
[pc = 0x00008000]:
```

Here is a description of each command:

- `step n`: Execute the next *n* steps of the program. If `n == -1`, then run indefinitely.
- `until n`: Execute instructions until `pc == n`.
- `peek n`: Display the word-sized value at address `n`.
- `peekb n`: Display the byte-sized value at address `n`.
- `peekaround n`: Display a group of word-sized values cenetered on address `n`.
- `poke n v`: Set the word-sized value at address `n` to the value `v`.
- `pokeb n v`: Set the byte-sized value at address `n` to the value `v`.
- `unhalt`: If the processor mode is halted, clear the halt bit to resume fetch/decode/execution.
- `showregister r`: Show the value in register `r`, where `r` may be expressed as a register number (`13` or `x13`, following the RISC-V convention) or a register name (e.g., `ra`, `t3`). Note that CSR's may be shown as well by name, as specified in Section 3 (e.g., `pc`, `md`).
- `setregister r v`: Set the value in register `r` to the value `v`.
- `showregisters d`: Display all of the registers, either in the current terminal (`d == text`) or in its own window (`d == graphic`).
- `showconsole d`: Display the contents of the console device, either in the current terminal (`d == text`) or in its own window (`d == graphic`).

6.3 Debugging

There are a few CSRs that are particularly useful for debugging. The values of these control the simulator's output, and can be used to stop execution at key moments.

Breakpoints: The `bp` CSR holds an address that is compared to `pc` at every instruction. When `pc == bp`, the simulator pauses. The `until` command sets this register to halt execution when the given address is reached.

Watchpoints: The `wp` CSR holds an address that is checked by the MMU at every instruction. When any operation—fetching, loading, storing, DMA transfer—accesses that address, the simulator pauses. This register must be set using the `setregister wp` command.

Debugging output: The *debugging level register* (`db`) controls how much debugging information the simulator emits. The default value is `1`, which emits each instruction as it is fetched, as well as higher-level events like DMA transfers. Setting the debugging level to `2` emits additional debugging information, including the detailed operation of the MMU and its translation of addresses. Level `0` silences all debugging output. The `setregister db` command allows you to set this value.