Runtime Environment

Zihao Yu

Institute of Computing Technology
Chinese Academy of Sciences

Introduction

You have already implement a minirv CPU

How to build a program to run on your CPU?

We have some programming tasks in this lecture

- But if your implementation of CPU is correct, only some software programming is needed

Some Preparations

minirv CPU

You should have

- the RTL implementation of minirv CPU
- a reference model implemented in C/C++

minirv CPU

You should have

- the RTL implementation of minirv CPU
- a reference model implemented in C/C++

Some programs we are going to build may execute millions of instruciton to finish

- If you do not have the reference model, it will be extremely HARD to debug
  - It will be VERY HARD to find the wrong instruction

Confirm the Memory Model

From the RISC-V manual:

```
        1.4 Memory

        A RISC-V hart has a single byte-addressable address space of 2^{XLEN} bytes
for all
        memory accesses.
```

- hart = Hardware Thread, which indicates CPU in our context

Confirm the Memory Model

From the RISC-V manual:

```
        1.4 Memory

        A RISC-V hart has a single byte-addressable address space of 2^{XLEN} bytes
for all
        memory accesses.
```

- hart = Hardware Thread, which indicates CPU in our context

- Logically, ISA sees only one memory
  - This is the Von Neumann Architecture

Confirm the Memory Model

From the RISC-V manual:

```
        1.4 Memory

        A RISC-V hart has a single byte-addressable address space of 2^{XLEN} bytes
for all
        memory accesses.
```

- hart = Hardware Thread, which indicates CPU in our context

- Logically, ISA sees only one memory
  - This is the Von Neumann Architecture
- For example, the following read results should be the same
  - CPU fetches a 4-bytes instruction from address `0x10`
  - CPU issues a `lw` to load a 4-bytes data from address `0x10`

Confirm the Memory Model

From the RISC-V manual:

```
        1.4 Memory

        A RISC-V hart has a single byte-addressable address space of 2^{XLEN} bytes
for all
        memory accesses.
```

- hart = Hardware Thread, which indicates CPU in our context

- Logically, ISA sees only one memory
  - This is the Von Neumann Architecture
- For example, the following read results should be the same
  - CPU fetches a 4-bytes instruction from address `0x10`
  - CPU issues a `lw` to load a 4-bytes data from address `0x10`
- Software also sees only one memory, conforming to the ISA spec
  - Programs generate by modern toolchains satisfy this property

The Implementation of Memory Model

- Physically, implementation can contain multiple memory units
  - But it should conform to the memory model, too

The Implementation of Memory Model

- Physically, implementation can contain multiple memory units
  - But it should conform to the memory model, too

For Logisim, there will be a little trouble for implementation

- Logisim does not provide memory storage with multiple ports

The Implementation of Memory Model

- Physically, implementation can contain multiple memory units
  - But it should conform to the memory model, too

For Logisim, there will be a little trouble for implementation

- Logisim does not provide memory storage with multiple ports
- You have ROM for instruction and RAM for data in your design
  - But they are different memory units

The Implementation of Memory Model

- Physically, implementation can contain multiple memory units
  - But it should conform to the memory model, too

For Logisim, there will be a little trouble for implementation

- Logisim does not provide memory storage with multiple ports
- You have ROM for instruction and RAM for data in your design
  - But they are different memory units
- To really implement the memory model, we should add extra circuit to provide a unified memory interface

```
        +-----+     inst. fetch    +------------+       +--------+
        |     | <----------------> |   Memory   |       | Memory |
        | CPU |                    |            | <---> |        |
        |     | <----------------> | Controller |       |  Unit  |
        +-----+  load/store inst.  +------------+       +--------+
```

The Implementation of Memory Model

- Physically, implementation can contain multiple memory units
  - But it should conform to the memory model, too

For Logisim, there will be a little trouble for implementation

- Logisim does not provide memory storage with multiple ports
- You have ROM for instruction and RAM for data in your design
  - But they are different memory units
- To really implement the memory model, we should add extra circuit to provide a unified memory interface

```
        +-----+     inst. fetch    +------------+       +--------+
        |     | <----------------> |   Memory   |       | Memory |
        | CPU |                    |            | <---> |        |
        |     | <----------------> | Controller |       |  Unit  |
        +-----+  load/store inst.  +------------+       +--------+
```

- Instead, by loading the same image file (the `.hex` file), we make ROM and RAM consistent

Recommandation for Implementation in RTL

- Implement memory as an array in C/C++
- Use DPI-C to let RTL code access the memory array in C/C++

```
#define MEM_SIZE (128 * 1024 * 1024)
static char memory[MEM_SIZE];
extern "C" int mem_read(int raddr) {
  // Return the 4-byte data at address `raddr & ~0x3u`.
  ...
}
extern "C" void mem_write(int waddr, int wdata, char wmask) {
  // Write `wdata` according to `wmask` to the 4-byte data at address `waddr & ~0x3u`
  // Every bit in `wmask` represents the mask of 1-byte in `wdata`.
  // For example, `wmask = 0x3` means only writing to the lower 2 bytes,
  // with other bytes in memory unchanged.
  ...
}
```

Recommandation for Implementation in RTL

- Implement memory as an array in C/C++
- Use DPI-C to let RTL code access the memory array in C/C++

```
#define MEM_SIZE (128 * 1024 * 1024)
static char memory[MEM_SIZE];
extern "C" int mem_read(int raddr) {
  // Return the 4-byte data at address `raddr & ~0x3u`.
  ...
}
extern "C" void mem_write(int waddr, int wdata, char wmask) {
  // Write `wdata` according to `wmask` to the 4-byte data at address `waddr & ~0x3u`
  // Every bit in `wmask` represents the mask of 1-byte in `wdata`.
  // For example, `wmask = 0x3` means only writing to the lower 2 bytes,
  // with other bytes in memory unchanged.
  ...
}
```

Advantages:

1. Only one memory physically, which conforms to the ISA spec

Recommandation for Implementation in RTL

- Implement memory as an array in C/C++
- Use DPI-C to let RTL code access the memory array in C/C++

```c
#define MEM_SIZE (128 * 1024 * 1024)
static char memory[MEM_SIZE];
extern "C" int mem_read(int raddr) {
  // Return the 4-byte data at address `raddr & ~0x3u`.
  ...
}
extern "C" void mem_write(int waddr, int wdata, char wmask) {
  // Write `wdata` according to `wmask` to the 4-byte data at address `waddr & ~0x3u`
  // Every bit in `wmask` represents the mask of 1-byte in `wdata`.
  // For example, `wmask = 0x3` means only writing to the lower 2 bytes,
  // with other bytes in memory unchanged.
  ...
}
```

Advantages:

1. Only one memory physically, which conforms to the ISA spec
2. Unnecessary to deal with the details of RTL

Recommandation for Implementation in RTL

- Implement memory as an array in C/C++
- Use DPI-C to let RTL code access the memory array in C/C++

```
#define MEM_SIZE (128 * 1024 * 1024)
static char memory[MEM_SIZE];
extern "C" int mem_read(int raddr) {
  // Return the 4-byte data at address `raddr & ~0x3u`.
  ...
}
extern "C" void mem_write(int waddr, int wdata, char wmask) {
  // Write `wdata` according to `wmask` to the 4-byte data at address `waddr & ~0x3u`
  // Every bit in `wmask` represents the mask of 1-byte in `wdata`.
  // For example, `wmask = 0x3` means only writing to the lower 2 bytes,
  // with other bytes in memory unchanged.
  ...
}
```

Advantages:

1. Only one memory physically, which conforms to the ISA spec
2. Unnecessary to deal with the details of RTL
3. Easy to implement simple device models
   - which will be introduced in the next lecture

Recommandation for Implementation in RTL

- Implement memory as an array in C/C++
- Use DPI-C to let RTL code access the memory array in C/C++

```
#define MEM_SIZE (128 * 1024 * 1024)
static char memory[MEM_SIZE];
extern "C" int mem_read(int raddr) {
  // Return the 4-byte data at address `raddr & ~0x3u`.
  ...
}
extern "C" void mem_write(int waddr, int wdata, char wmask) {
  // Write `wdata` according to `wmask` to the 4-byte data at address `waddr & ~0x3u`
  // Every bit in `wmask` represents the mask of 1-byte in `wdata`.
  // For example, `wmask = 0x3` means only writing to the lower 2 bytes,
  // with other bytes in memory unchanged.
  ...
}
```

Advantages:

1. Only one memory physically, which conforms to the ISA spec
2. Unnecessary to deal with the details of RTL
3. Easy to implement simple device models
    - which will be introduced in the next lecture

We will finally change it to a real memory controller in SoC

Runtime Environment

The Running of a Hello Program

```c
// hello.c
#include <stdio.h>
int main() {
  printf("Hello, World!\n");
  return 0;
}
```

```
gcc hello.c
./a.out
```

But, …

- Who invokes the Hello program?
- Where is the actual code of `printf()`?
- How does the Hello program end?

Intuition - The Hello program can not run only by itself

- There must be something helps the Hello program to run!
- That is runtime environment - a set of software helps other programs to run

Responsibility of the Runtime Environment

1. Before program execution - perform preparation
   - Load the program, set up the connection to libraries, set up arguments…
2. During program execution - provide the support of libraries
3. After program execution - clean up before the program really exits

Two Types of Runtime Environments

In the C standard document:

```
        5.1.2 Execution environments

        Two execution environments are defined: freestanding and hosted. In both 
cases,
        program startup occurs when a designated C function is called by the 
execution
        environment... Program termination returns control to the execution 
environment.
```

```
        5.1.2.1 Freestanding environment

        1. In a freestanding environment (in which C program execution may take place
        without any benefit of an operating system), the name and type of the 
function
        called at program startup are implementation-defined.
```

- This is actually the runtime environment for a program to run on your CPU
  - Your CPU can not boot an OS now

Two Types of Runtime Environments(2)

```
        5.1.2.2 Hosted environments
        5.1.2.2.1 Program startup

        1. The function called at program startup is named main...
```

- In a hosted environment, there is a hosted OS
  - This is the runtime environment used when you learn C Programming Language
  - Linux provides a hosted environment to program

```
        2. If they are declared, the parameters to the main function shall obey the
following
        constraints:
        ...
        — If the value of argc is greater than zero, the string pointed to by argv[0]
          represents the program name; ...
          If the value of argc is greater than one, the strings pointed to by argv[1]
          through argv[argc-1] represent the program parameters.
```

- The convention of the program parameters

Freestanding Environment

Freestanding Environment for `1+2+...+10`

1. Before program execution
   - Program loading
     - Use GUI to choose the image file in Logisim
     - Initialize with a global array, or read from file in sEMU
   - No support for argument passing

2. During program execution
   - No support for library functions

3. After program execution
   - Use an infinity loop to indicate the end of the program
     - Implemented by `bner0`

Enhancement of Runtime Environment

- The summation program does not need a complicated runtime environment to run
  - In other words, to run more complicated programs, the runtime environment should provide more functionalities

TASK 1 - displaying an integer in sEMU

- sISA - add a new instruction `out rs`
- sEMU - when executing `out rs`, display `R[rs]` to the terminal
- program - use this instruction

Enhancement of Runtime Environment(2)

TASK 2 - input `n` as an argument from command line, then compute `1+2+...+n`

- Let runtime environment places `n` somewhere before the program runs
- Program reads `n` from the same place
- This is a kind of convention between the runtime environment and the program

- sEMU - place the argument input by user to register `r0`
- program - get the last term of the summation from `r0`, and compute the summation

Summary - the Enhanced Runtime Environment

1. Before program execution
   - Program loading - initialize with a global array, or read from file in sEMU
   - Argument passing - use register `r0` to pass an integer to the program

2. During program execution
   - Can display an integer with `out` instruction

3. After program execution
   - Use an infinity loop to indicate the end of the program
     - Implemented by `bner0`

Abstract Machine - a Freestanding Environment for Building Computer System

Challenges during Building Computer System

1. Several platforms are slightly different to support program
   - Logisim, instruction set simulator, RTL simulation for CPU, RTL simulation with SoC, real chip, …
     - The way to load a program
     - The way to display a digit/character
     - The way to terminate a program
     - …
   - For the program, how to code once, run everywhere?

2. CPU (or computer system) is developed step by step
   - Pure computation, I/O, exception & interrupt, virtual memory, …
   - How to let runtime environment and program enhance gradually along with CPU?

Two Important Principles in Computer System

1. Abstraction - add a new layer to hide the underlying differences
   - Abstract the differences as APIs
     - They are implemented by the runtime environment
   - Program calls these APIs, without knowing the implementation

2. Modularity - group these APIs by the phases of development, and select them on demand
   - TRM(TuRing Machine) - pure computation, which is mandatory
   - IOE(I/O Extension)
   - CTE(ConText Extension)
   - VME(Virtual Memory Extension)
   - MPE(Multi-Processor Extension)

Abstract Machine - a New Runtime Env.

- Following the principles, we propose Abstract Machine (AM)
  - A new freestanding runtime environment suitable for building computer system
- The power of abstraction
  - For program - write/port once, run everywhere
  - For platform - support AM, run everything
  - For Debugging - can benefit from the decoupling
    - First debug the program on Linux native
    - Then debug your CPU with the program

```
                                         +-------+------+-----+------------------+---
--+
                                         | Mario | Demo |Shell|        ...       |
... |
        +----------+-------+-------------+------ +------+-----+------------------+---
--+
        | CoreMark | LLaMa | Typing Game | FCEUX | RTOS | xv6 | Self-designed OS |
... |
        +----------+-------+-------------+-------+------+-----+------------------+---
--+
```

Use AM to Build Programs for Your CPU

Build a minirv Program

1. Install tools

```
        sudo apt install g++-riscv64-linux-gnu git
```

2. Clone the repo

```
        git clone https://github.com/NJU-ProjectN/abstract-machine
        git clone https://github.com/NJU-ProjectN/am-kernels
```

3. Set environment variable

```
        cd abstract-machine
        echo "export AM_HOME=`pwd`" >> ~/.bashrc  # change this if your shell is not bash
        source ~/.bashrc  # or close the terminal and open a new terminal
```

4. Check the environment variable

```
        echo $AM_HOME   # should display the path you just set in the previous step
```

5. Build a program

```
        cd am-kernels/tests/cpu-tests
        make ARCH=minirv-npc ALL=dummy  # npc stands for New Processor Core
```

You may encounter a compile error, fix it according the next page…

Build a minirv Program(2)

```
make ARCH=minirv-npc ALL=dummy
```

Try to fix the compile error as following:

```
/usr/riscv64-linux-gnu/include/bits/wordsize.h:28:3: error: #error "rv32i-based targets are not supported"
```

```
--- /usr/riscv64-linux-gnu/include/bits/wordsize.h
+++ /usr/riscv64-linux-gnu/include/bits/wordsize.h
@@ -25,5 +25,5 @@
 #if __riscv_xlen == 64
 # define __WORDSIZE_TIME64_COMPAT32 1
 #else
-# error "rv32i-based targets are not supported"
+# define __WORDSIZE_TIME64_COMPAT32 0
 #endif
```

```
/usr/riscv64-linux-gnu/include/gnu/stubs.h:8:11: fatal error: gnu/stubs-ilp32.h: No such file or directory
```

```
--- /usr/riscv64-linux-gnu/include/gnu/stubs.h
+++ /usr/riscv64-linux-gnu/include/gnu/stubs.h
@@ -7,3 +7,3 @@
 #if __WORDSIZE == 32 && defined __riscv_float_abi_soft
-# include <gnu/stubs-ilp32.h>
+//# include <gnu/stubs-ilp32.h>
 #endif
```

Run the Program

```
        make ARCH=minirv-npc ALL=dummy
        ls build/  # should contain the following files
            dummy-minirv-npc.bin # the binary to load into the memory array in the simulation
            dummy-minirv-npc.elf # ELF program, which contains some structural information
            dummy-minirv-npc.txt # disassembly result
```

TASK 3 - run `dummy` in your CPU

- Change the reset value of PC to `0x80000000`
- It should get stucked in `halt()` with `a0 = 0`

TASK 4 - add `ebreak` in `halt()` to finish simulation automatically

- `asm volatile("mv a0, %0; ebreak" : :"r"(code));`

TASK 5 - implement the `run` rule in Makefile

TRM - A Basic Part of Runtime Environment for Pure Computation

TRM - A Simple Freestanding Runtime Environment

For pure computation, program needs:

- A way to exection instructions
  - Provided by CPU
- Memory to store data - heap
- An entry - `main(const char *args)`
  - The prototype of `main()` is different from `int main(int argc, char *argv[])`
  - But it is OK, since it is implementation-defined
- A way to exit - `halt()`
- A way to display a character - `putch()`
  - Every meaningful program should output something

Read the Friendly Source Code

- `abstract-machine/am/` - implementations of AM APIs for different platforms
  - `abstract-machine/am/include/am.h` lists all APIs for reference
- `abstract-machine/klib/` - platform-independent library functions
  - Include functions commonly used (`printf()`, `strcpy()`…)
- `abstract-machine/scripts/` - build scripts
  - Indicate what to be compiled and linked, and how

- `am-kernels` - some programs developped over AM

How to Build a Program

1. `abstract-machine/scripts/minirv-npc.mk` indicates source files for implementation of AM APIs
   - `gcc` will compile them into object files
   - then `ar` packs the object files into an archive file
2. `gcc` and `ar` do similar things to klib
3. `gcc` compile source files indicated by `SRCS` into object files
   - For example, `am-kernels/kernels/hello/hello.c`
4. `ld` links the generated files above into an executable file, according to the linker script
   - `abstract-machine/scripts/linker.ld`
5. `objcopy` copies code and data in the executable file into a binary image
6. Load the binary image into the memory of your CPU and run!

How does the Program start and end?

1. The linker script indicated that the entry of the program is `_start`
   - which is defined in `abstract-machine/am/src/riscv/npc/start.S`
2. `_start` will do the following:
   - first set the stack pointer
   - then call `_trm_init()`, which is defined in `abstract-machine/am/src/riscv/npc/trm.c`
3. `trm_init()` will call `main()`
4. After `main()` returns, call `halt()`
5. `halt()` will finally terminate the program

OPTIONAL: How to control `gcc` to only generate 8 instructions?

See `abstract-machine/tools/minirv/`

In general:

1. Let `gcc` compile a `.c` file to a `.S` file
2. Insert one line of `#include "inst-replace.h"` at the beginning of the `.S` file
   - `inst-replace.h` contains code to replace other instructions by the 8 target instructions
     - In other words, we implement other instructions using the 8 target instructions
3. Let `gcc` compile the `.S` file into a `.o` file
   - The `.o` file should contain only 8 instructions

Run More Programs to Test Your CPU

Run More Programs

First change the memory size to 128MB

- It should be enough for the following programs

TASK 6 - run `riscv-tests` to test your CPU

```
git clone https://github.com/NJU-ProjectN/riscv-tests-am
cd riscv-tests-am
make ARCH=minirv-npc run           # run all testcases
make ARCH=minirv-npc run ALL=addi  # only run one testcase
```

TASK 7 - run `cpu-tests` to test your CPU

```
cd am-kernels/tests/cpu-tests
make ARCH=minirv-npc run          # run all testcases
make ARCH=minirv-npc run ALL=fib  # only run one testcase
```

NOTE: `string` and `hello-str` should fail

- They depend on klib, which we will provide later
- Just ignore them now

END