# A1. Assembly Programming in RISC-V

CSE261: Computer Architecture, Fall 2021

Hyungon Moon

Out: Oct 5, 2021

Due: Oct 14 2021 11:59pm

# Goal

- Write programs in RISC-V assembly.
- Understand the calling convention, stack, etc.

# Vagrant and VirtualBox

- Vagrant allows you to easily create and access a virtual machine.
- Install Vagrant from

```
https://www.vagrantup.com/
```

- Vagrant (in this class) need VirtualBox. Install from

```
https://www.virtualbox.org/
```

- Should work on Ubuntu, Mac or Windows.
- May not work inside another virtual machine.

# Initializing VM (Mac, Ubuntu (Linux))

- Use terminal to enter the `hw1` directory
- Make sure that you are seeing `Vagrantfile` in the directory
- Create and boot VM

```
vagrant up
```

- Connect to the VM

```
vagrant ssh
```

- From V2 of the assignment, the `hw1` directory is automatically synced with the `hw` directory

# Initializing VM (Windows)

- Use terminal to enter the `hw1` directory
- Make sure that you are seeing `Vagrantfile` in the directory
- Create and boot VM

```
vagrant.exe up
```

- Connect to the VM

```
vagrant.exe ssh
```

- From V2 of the assignment, the `hw1` directory is automatically synced with the `hw` directory

# Spike, Proxy kernel and Toolchain

- When creating VM, the toolchains and tools are automatically built and installed.
- ***Spike*** is a RISC-V ISA simulator.
- You will run your programs using spike with the ***Proxy Kernel*** (pk).
- You will have toolchain (compiler) for the RISC-V.

# Task 0: Add

- Add is an example assignment for you to test drive.
- Files (at `hw1/add` or `hw/add`)

```
Makefile main.c add.s test.txt
```

- What to edit: `add.s` (already implemented)
- Testing

```
make
```

# Task 0.5: Mem

- Mem is another example assignment for you to test drive.
- Files (at `hw1/mem` or `hw/mem`)

```
Makefile main.c mem.s test.txt
```

- What to edit: `mem.s` (already implemented)
- Testing

```
make
```

- Shows how you can work with data stored in memory.

# Debugging (1/7)

- Command
  - ➢Run spike with -d option
  - ➢Easy way

```
make debug
```

  - ➢Hard way

```
Spike -m128 –d pk add
```

# Debugging (2/7)

- Check the available commands.

```
: help
```

```
Interactive commands:

reg <core> [reg]              # Display [reg] (all if omitted) in <core>
...
until pc <core> <val> ...     # Stop when PC in <core> hits <val>
run [count]                   # Resume noisy execution
                              # (until CTRL+C, or [count] insns)
...
quit                          # End the simulation ...
```

- We have one core, so use 0 for <core>

# Debugging (3/7)

- Show register values

```
: reg 0
zero: 0x0000000000000000  ra: 0x0000000000000000  sp: 0x0000000000000000  gp: 0x0000000000000000
  tp: 0x0000000000000000  t0: 0x0000000000000000  t1: 0x0000000000000000  t2: 0x0000000000000000
  s0: 0x0000000000000000  s1: 0x0000000000000000  a0: 0x0000000000000000  a1: 0x0000000000000000
  a2: 0x0000000000000000  a3: 0x0000000000000000  a4: 0x0000000000000000  a5: 0x0000000000000000
  a6: 0x0000000000000000  a7: 0x0000000000000000  s2: 0x0000000000000000  s3: 0x0000000000000000
  s4: 0x0000000000000000  s5: 0x0000000000000000  s6: 0x0000000000000000  s7: 0x0000000000000000
  s8: 0x0000000000000000  s9: 0x0000000000000000 s10: 0x0000000000000000 s11: 0x0000000000000000
  t3: 0x0000000000000000  t4: 0x0000000000000000  t5: 0x0000000000000000  t6: 0x0000000000000000
```

- Running program for a while (for 5 cycles)

```
: r 5
core   0: 0x0000000000001000 (0x00000297) auipc   t0, 0x0
core   0: 0x0000000000001004 (0x02028593) addi    a1, t0, 32
core   0: 0x0000000000001008 (0xf1402573) csrr    a0, mhartid
core   0: 0x000000000000100c (0x0182b283) ld      t0, 24(t0)
core   0: 0x0000000000001010 (0x00028067) jr      t0
```

# Debugging (4/7)

- Run from my own code
- Find out the address from add.d
  (use search function in your text editor)

```
…
0000000000010250 <add>:
   10250:       952e                    add     a0,a0,a1
   10252:       8082                    ret
```

- Use the address to reach the code

```
: until pc 0 10250
bbl loader
: pc 0
0x0000000000010250
: r 3
core   0: 0x0000000000010250 (0x0000952e) c.add   a0, a1
core   0: 0x0000000000010252 (0x00008082) ret
core   0: 0x00000000000101d8 (0x000087aa) c.mv    a5, a0
```

# Debugging (5/7)

- Run instruction one by one

```
: until pc 0 10250
bbl loader
: pc 0
0x0000000000010250
: reg 0
zero: 0x0000000000000000  ra: 0x00000000000101d8  sp: 0x0000003fffffb00  gp: 0x00000000000262b0
  tp: 0x0000000000000000  t0: 0x0000000000001000  t1: 0x0000000000000001  t2: 0x000000000000000a
  s0: 0x0000003ffffffb50  s1: 0x0000000000000000  a0: 0x0000000000000001  a1: 0x0000000000000002
  a2: 0x0000003fffffea40  a3: 0x0000000000000000  a4: 0x0000000000000002  a5: 0x0000000000000001
  a6: 0x0000000000000003  a7: 0x000000000000003f  s2: 0x0000000000000000  s3: 0x0000000000000000
  s4: 0x0000000000000000  s5: 0x0000000000000000  s6: 0x0000000000000000  s7: 0x0000000000000000
  s8: 0x0000000000000000  s9: 0x0000000000000000 s10: 0x0000000000000000 s11: 0x0000000000000000
  t3: 0x0000000000000000  t4: 0x0000000000000009  t5: 0xffffffffffffffff  t6: 0x0ccccccccccccccc
: r 1
core   0: 0x0000000000010250 (0x0000952e) c.add   a0, a1
: reg 0
zero: 0x0000000000000000  ra: 0x00000000000101d8  sp: 0x0000003fffffb00  gp: 0x00000000000262b0
  tp: 0x0000000000000000  t0: 0x0000000000001000  t1: 0x0000000000000001  t2: 0x000000000000000a
  s0: 0x0000003ffffffb50  s1: 0x0000000000000000  a0: 0x0000000000000003  a1: 0x0000000000000002
  a2: 0x0000003fffffea40  a3: 0x0000000000000000  a4: 0x0000000000000002  a5: 0x0000000000000001
  a6: 0x0000000000000003  a7: 0x000000000000003f  s2: 0x0000000000000000  s3: 0x0000000000000000
  s4: 0x0000000000000000  s5: 0x0000000000000000  s6: 0x0000000000000000  s7: 0x0000000000000000
  s8: 0x0000000000000000  s9: 0x0000000000000000 s10: 0x0000000000000000 s11: 0x0000000000000000
  t3: 0x0000000000000000  t4: 0x0000000000000009  t5: 0xffffffffffffffff  t6: 0x0ccccccccccccccc
```

# Debugging (6/7)

- Investigating memory contents (`mem` example)
  - ➢You can find the address of `copy` from `mem.d`

```
0000000000001039c <copy>:
   1039c:      4e01                   li      t3,0
   1039e:      00ce0f63               beq     t3,a2,103bc <out>

00000000000103a2 <loop>:
   103a2:      002e1293               slli    t0,t3,0x2
   103a6:      00558eb3               add     t4,a1,t0
   103aa:      00550f33               add     t5,a0,t0
   103ae:      000eaf83               lw      t6,0(t4)
   103b2:      01ff2023               sw      t6,0(t5)
   103b6:      0e05                   addi    t3,t3,1
   103b8:      fece15e3               bne     t3,a2,103a2 <loop>
```

- Run until the address and check the register contents

```
: until pc 0 1039c
bbl loader
: reg 0
zero: 0x0000000000000000  ra: 0x0000000000010252  sp: 0x0000003ffffffae0  gp: 0x00000000000264f0
  tp: 0x0000000000000000  t0: 0x0000000000001000  t1: 0x0000000000000002  t2: 0x000000000000000a
  s0: 0x0000003ffffffb50  s1: 0x0000000000000000  a0: 0x0000000000027420  a1: 0x00000000000273f0
  a2: 0x000000000000000a  a3: 0x0000000000000000  a4: 0x000000000000000a  a5: 0x000000000000000a
```

Location of destination buffer

Buffer Length

Location of source buffer

# Debugging (7/7)

- Use the address to see the content

```
: mem 0 273f0
0x0000000200000001
: mem 0 273f4
0x0000000000000002
: mem 0 27420
0x0000000000000000
: until pc 0 103bc
: mem 0 27420
0x0000000200000001
: mem 0 27424
0x0000000000000002
```

Execute through return

Memory copied

Simulator bug

```
mem <hex addr>   # Show contents of …
```

```
mem <core> <hex addr>   # Show contents of …
```

- Simulator bug
  - ➢When reading memory, you need to provide <core> as well.

# Task 1: GCD and LCM (1/2)

- GCD: greatest common divisor
- LCM: least common multiple
- Files (at `hw1/gl` or `hw/gl`)

```
Makefile main.c gcdlcm.s test.txt
```

- What to edit: `gcdlcm.s` (already implemented)
- Test file format: <a> <b> <gcd of a and b> <lcm of a and b>

```
1170 3094 26 139230
```

# Task 1: GCD and LCM (2/2)

- Write two functions: `gcd` and `lcm`

```
.global gcd
gcd:
/* Your Code for gcd from here */
/* a @ x10, b @ x11, return gcd */
/* Your Code for gcd to here */
   jr x1

.global lcm
lcm:
/* Your Code for lcm from here */
/* a @ x10, b @ x11 return lcm */
/* Your Code for lcm to here */
   jr x1
```

# Task 2: Matrix Multiplication (1/4)

- Multiply two square matrices
- Files (at `hw1/matmul` or `hw/matmul`)

```
Makefile main.c gcdlcm.s test.txt matmul.py
```

- What to edit: `matmul.s` (already implemented)
- Test file format

  <dim> <row 1 of a> … <row n of a> <row 1 of b> … <row n of b> <row 1 of res> … <row n of res>

```
1170 3094 26 139230
```

- Use `matmul.py` to generate more test cases.

# Task 2: Matrix Multiplication (2/4)

- Write one function: `matmul` (write and use a helper, `matmul_idx`)

```
matmul:
/* save return address (in x1) in stack*/
/* first argument (x10): the address of output buffer */
/* second argument (x11): the start address of a */
/* third argument (x12): the start address of b */
/* fourth argument (x13): the dimension */
/* your matmul code from here */
   addi  sp, sp, -8
   sd x1, 0(sp)
   /* ...*/
   ld x1, 0(sp)
   addi  sp, sp, 8
/* your matmul code to here */
   jr x1
```

# Task 2: Matrix Multiplication (3/4)

- How data are stored and supposed to be stored.
  - When the base address is 0, the data are stored at
    - 0, 0+ 4, 0x + 8, …
- For example, when dim is 2, `x11` has `0x100` and `x12` has `0x200`
  - First matrix is stored in `0x100, 0x104, 0x108, 0x10c.`
  - Second mattix is stored in `0x200, 0x204, 0x208, 0x20c.`
- If `x10` contains `0x100`, output should be stored at
  - `0x300, 0x304, 0x308, 0x30c.`

# Task 2: Matrix Multiplication (4/4)

- Write one functions: `matmul` (write and use a helper, `lcm`

```
.global matmul_idx
matmul_idx:
/* Recommanded arguments */
/* first argument (x10): the address of output buffer */
/* second argument (x11): the start address of a */
/* third argument (x12): the start address of b */
/* fourth argument (x13): the dimension */
/* fourth argument (x14): row index of the result matrix to fill
out */
/* fourth argument (x15): column index of the result matrix to
fill out */
/* your matmul_idx (helper function) code from here */
/* your matmul_idx (helper function) code to here */
   jr x1
```

# Task 3: (Insertion) sort (1/2)

- Function that sorts a list of integers
- Files (at `hw1/sort` or `hw/sort`)

```
Makefile main.c sort.s test.txt sort.py
```

- What to edit: `sort.s` (already implemented)
- Test file format

  <length> <num 0> <num 1> … <num n>

```
6 85 295 449 32 463 332
```

- Use `sort.py` to generate more test cases.

# Task 3: (Insertion) sort (2/3)

- Write one function: `sort` (write and use a helper, `insert`)

```
.global sort
sort:
/* first  argument (x10): the address of output buffer */
/* second argument (x11): the start address of the incoming list
*/
/* third  argument (x12): the length of incoming list */
/* your sort code from here */
/* your sort code to here */
   jr x1
```

# Task 3: (Insertion) sort (3/3)

- Write one function: `sort` (write and use a helper, `insert`)

```
.global insert
insert:
/* Recommanded arguments */
/* first argument (x10): the address of output buffer */
/* third  argument (x11): the current length of output buffer */
/* third  argument (x12): the integer to insert */
/* your insert code from here */
/* your insert code to here */
   jr x1
```

# (note) RISC-V calling convention and reg names

| Register | ABI Name | Description | Saver |
|---|---|---|---|
| x0 | zero | Hard-wired zero | — |
| x1 | ra | Return address | Caller |
| x2 | sp | Stack pointer | Callee |
| x3 | gp | Global pointer | — |
| x4 | tp | Thread pointer | — |
| x5–7 | t0–2 | Temporaries | Caller |
| x8 | s0/fp | Saved register/frame pointer | Callee |
| x9 | s1 | Saved register | Callee |
| x10–11 | a0–1 | Function arguments/return values | Caller |
| x12–17 | a2–7 | Function arguments | Caller |
| x18–27 | s2–11 | Saved registers | Callee |
| x28–31 | t3–6 | Temporaries | Caller |
| f0–7 | ft0–7 | FP temporaries | Caller |
| f8–9 | fs0–1 | FP saved registers | Callee |
| f10–11 | fa0–1 | FP arguments/return values | Caller |
| f12–17 | fa2–7 | FP arguments | Caller |
| f18–27 | fs2–11 | FP saved registers | Callee |
| f28–31 | ft8–11 | FP temporaries | Caller |

Table 18.2: RISC-V calling convention register usage.