Jaehoon Shim
Ikjoon Son
Seongyeop Jeong
(snucsl.ta@gmail.com)

Systems Software &
Architecture Lab.

Seoul National University

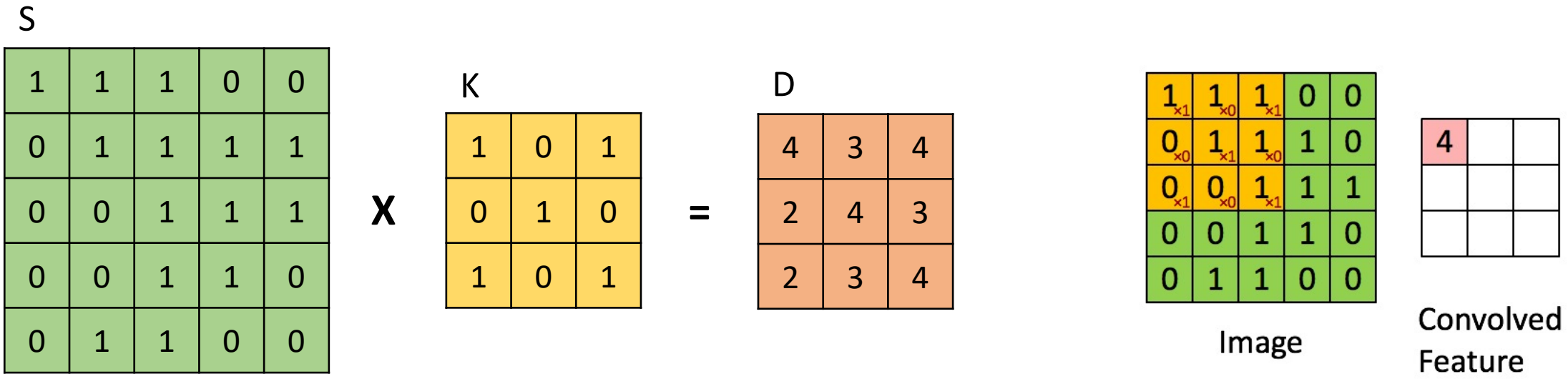
Fall 2021

4190.308: Computer Architecture Lab. 3

Image Convolution in the RISC-V Assembly Language

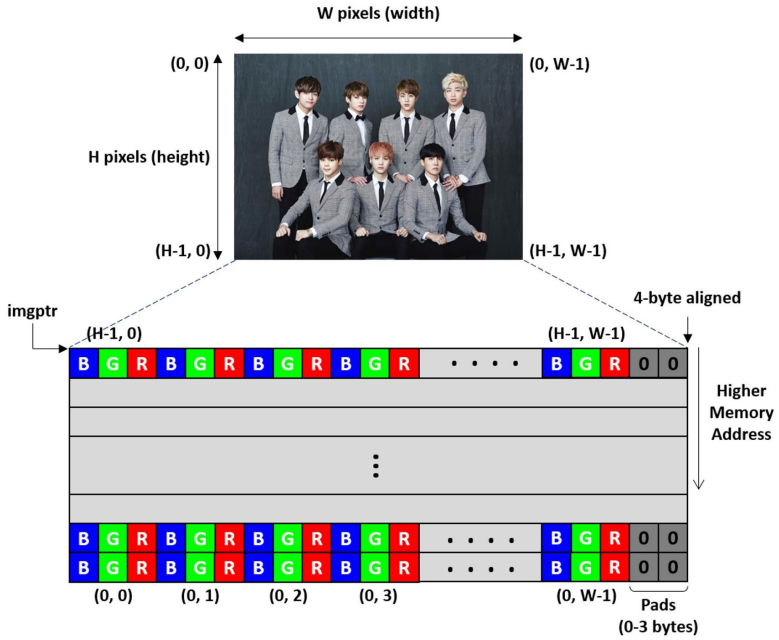
Image Convolution

- Convolution is a process of adding each element of the image to its local neighbors, weighted by the kernel
  - A *kernel* is a small matrix used for blurring, sharpening, and more
- Convolution is also used in machine learning for feature extraction



BMP Data Format

- Bitmap data describes the image pixel by pixel
- Each pixel consists of 8-bit blue(B), green(G), and red(R) byte
- Pixels are stored "upside-down"
- # of bytes occupied by each row should be a multiple of 4
  - If not, the remaining bytes are padded with zero



Specification

- Complete the file bmpconv.s that implements *bmpconv*() in 32-bit RISC-V assembly language
- *void bmpconv(unsigned char *imgptr, int h, int w, unsigned char *k, unsigned char *outptr)*
  - *imgptr* points to the bitmap data that stores the actual image, pixel by pixel
  - *h* & *w* represent the height and width of the given image (in pixels)
  - *k* points to the address of the kernel
  - *outptr* points to the address of the output image
- parameters for *bmpconv*() are available in the a0 ~ a4 registers

Specification

- The kernel is always a 3x3 matrix stored in memory using row-major ordering (as in the C language)
  - k == &(k[0][0]), k+2 == &(k[0][2]), …, k+8 == &(k[2][2])

- Each element in the kernel is set to one of 0, 1, and -1

Specification

- You should use saturation arithmetic operation to get the resulting pixel value

- For the given signed integer v, Sat(v) will return an 8-bit unsigned integer where
  - if v < 0, Sat(v) = 0
  - if v > 255, Sat(v) = 255
  - otherwise, Sat(v) = v

Restrictions

- You should use only the following registers in bmpconv.s
  : zero(x0), sp, ra, a0~a4, t0~t4
  - Use stack as temporary storage if needed (maximum 256 bytes)

- You should use *lw* and *sw* RISC-V instructions to access data in memory
  - *lw/sw*: load/store a 4-byte word from/to memory
  - You should consider byte ordering (little endian)

- The padding area in the output image should be set to 0

Restrictions

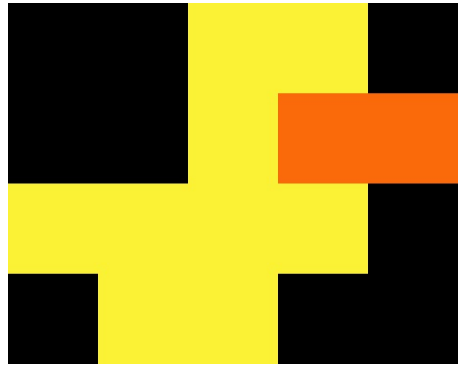
- Your program should work for any input image where $h > 3, w > 3$

- Your solution will be rejected if it contains keywords like
  - .octa, .quad, .long, .int, .word, .short, .hword, .byte, .double, .single, .float, etc

- Your solution should finish within a reasonable time

Example

- Input stream (in 4-byte unit)



* For ease of explanation, pixels are not stored 'upside-down'

0x00	0x00	0x00	0x00	0x00	0x00	0x35	0xf2	0xfb	0x35	0xf2	0xfb	0x00	0x00	0x00	0x00
0x00	0x00	0x00	0x00	0x00	0x00	0x35	0xf2	0xfb	0x0a	0x6a	0xfa	0x0a	0x6a	0xfa	0x00
0x35	0xf2	0xfb	0x35	0xf2	0xfb	0x35	0xf2	0xfb	0x35	0xf2	0xfb	0x00	0x00	0x00	0x00
0x00	0x00	0x00	0x35	0xf2	0xfb	0x35	0xf2	0xfb	0x00	0x00	0x00	0x00	0x00	0x00	0x00

Example

- Input stream (in 4-byte unit)

0x00000000

0x00	0x00	0x00	0x00	0x00	0x00	0x35	0xf2	0xfb	0x35	0xf2	0xfb	0x00	0x00	0x00	0x00
0x00	0x00	0x00	0x00	0x00	0x00	0x35	0xf2	0xfb	0x0a	0x6a	0xfa	0x0a	0x6a	0xfa	0x00
0x35	0xf2	0xfb	0x35	0xf2	0xfb	0x35	0xf2	0xfb	0x35	0xf2	0xfb	0x00	0x00	0x00	0x00
0x00	0x00	0x00	0x35	0xf2	0xfb	0x35	0xf2	0xfb	0x00	0x00	0x00	0x00	0x00	0x00	0x00

Example

- Input stream (in 4-byte unit)

RISC-V uses little endian
(Least significant byte has the smallest address)

0x00000000
0xf2350000

0x00	0x00	0x00	0x00	0x00	0x00	0x35	0xf2	0xfb	0x35	0xf2	0xfb	0x00	0x00	0x00	0x00
0x00	0x00	0x00	0x00	0x00	0x00	0x35	0xf2	0xfb	0x0a	0x6a	0xfa	0x0a	0x6a	0xfa	0x00
0x35	0xf2	0xfb	0x35	0xf2	0xfb	0x35	0xf2	0xfb	0x35	0xf2	0xfb	0x00	0x00	0x00	0x00
0x00	0x00	0x00	0x35	0xf2	0xfb	0x35	0xf2	0xfb	0x00	0x00	0x00	0x00	0x00	0x00	0x00

Example

- Input stream (in 4-byte unit)

RISC-V uses little endian
(Least significant byte has the smallest address)

```
0x00000000
0xf2350000
0xfbf235fb
```

0x00	0x00	0x00	0x00	0x00	0x00	0x35	0xf2	0xfb	0x35	0xf2	0xfb	0x00	0x00	0x00	0x00
0x00	0x00	0x00	0x00	0x00	0x00	0x35	0xf2	0xfb	0x0a	0x6a	0xfa	0x0a	0x6a	0xfa	0x00
0x35	0xf2	0xfb	0x35	0xf2	0xfb	0x35	0xf2	0xfb	0x35	0xf2	0xfb	0x00	0x00	0x00	0x00
0x00	0x00	0x00	0x35	0xf2	0xfb	0x35	0xf2	0xfb	0x00	0x00	0x00	0x00	0x00	0x00	0x00

Example

- Input stream (in 4-byte unit)

RISC-V uses little endian
(Least significant byte has the smallest address)

```
0x00000000
0xf2350000
0xfbf235fb
0x00000000
```

0x00	0x00	0x00	0x00	0x00	0x00	0x35	0xf2	0xfb	0x35	0xf2	0xfb	0x00	0x00	0x00	0x00
0x00	0x00	0x00	0x00	0x00	0x00	0x35	0xf2	0xfb	0x0a	0x6a	0xfa	0x0a	0x6a	0xfa	0x00
0x35	0xf2	0xfb	0x35	0xf2	0xfb	0x35	0xf2	0xfb	0x35	0xf2	0xfb	0x00	0x00	0x00	0x00
0x00	0x00	0x00	0x35	0xf2	0xfb	0x35	0xf2	0xfb	0x00	0x00	0x00	0x00	0x00	0x00	0x00

Example

- Input stream (in 4-byte unit)

0x00	0x00	0x00	0x00	0x00	0x00	0x35	0xf2	0xfb	0x35	0xf2	0xfb	0x00	0x00	0x00	0x00
0x00	0x00	0x00	0x00	0x00	0x00	0x35	0xf2	0xfb	0x0a	0x6a	0xfa	0x0a	0x6a	0xfa	0x00
0x35	0xf2	0xfb	0x35	0xf2	0xfb	0x35	0xf2	0xfb	0x35	0xf2	0xfb	0x00	0x00	0x00	0x00
0x00	0x00	0x00	0x35	0xf2	0xfb	0x35	0xf2	0xfb	0x00	0x00	0x00	0x00	0x00	0x00	0x00

```
0x00000000
0xf2350000
0xfbf235fb
0x00000000

0x00000000
0xf2350000
0xfa6a0afb
0x00fa6a0a

0x35fbf235
0xf235fbf2
0xfbf235fb
0x00000000

0x35000000
0xf235fbf2
0x000000fb
0x00000000
```

Example

- Kernel stream (in 4-byte unit)

0x01	0x01	0x00
0xFF	0x00	0xFF
0x00	0x01	0x01

```
0xFF000101
0x0100FF00
0x01
```

Example

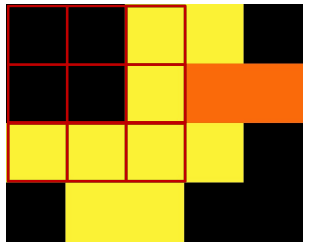
imgptr (*w*: 5, *h*: 4)

0x00	0x00	0x00	0x00	0x00	0x00	0x35	0xf2	0xfb	0x35	0xf2	0xfb	0x00	0x00	0x00	0x00
0x00	0x00	0x00	0x00	0x00	0x00	0x35	0xf2	0xfb	0x0a	0x6a	0xfa	0x0a	0x6a	0xfa	0x00
0x35	0xf2	0xfb	0x35	0xf2	0xfb	0x35	0xf2	0xfb	0x35	0xf2	0xfb	0x00	0x00	0x00	0x00
0x00	0x00	0x00	0x35	0xf2	0xfb	0x35	0xf2	0xfb	0x00	0x00	0x00	0x00	0x00	0x00	0x00

k →

0x01	0x01	0x00
0xFF	0x00	0xFF
0x00	0x01	0x01

Example



Start with 8-bit blue byte

0x00 0x01	0x00	0x00	0x00 0x01	0x00	0x00	0x35 0x00	0xf2	0xfb	0x35	0xf2	0xfb	0x00	0x00	0x00	0x00
0x00 0xff	0x00	0x00	0x00 0x00	0x00	0x00	0x35 0xff	0xf2	0xfb	0x0a	0x6a	0xfa	0x0a	0x6a	0xfa	0x00
0x35 0x00	0xf2	0xfb	0x35 0x01	0xf2	0xfb	0x35 0x01	0xf2	0xfb	0x35	0xf2	0xfb	0x00	0x00	0x00	0x00
0x00	0x00	0x00	0x35	0xf2	0xfb	0x35	0xf2	0xfb	0x00	0x00	0x00	0x00	0x00	0x00	0x00

```
Sat((0x00 *  1)+(0x00 * 1)+(0x35 *  0) +
    (0x00 * -1)+(0x00 * 0)+(0x35 * -1) +
    (0x35 *  0)+(0x35 * 1)+(0x35 *  1) =
```

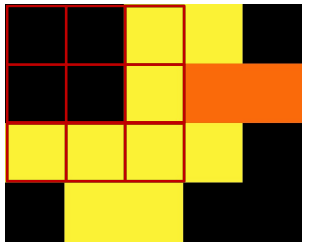
Sat(53) = 53 (0x35)

outptr →

(*w*:3, *h*:2)

0x35								

Example



Continue with 8-bit green byte

0x00	0x00 0x01	0x00	0x00	0x00 0x01	0x00	0x35	0xf2 0x00	0xfb	0x35	0xf2	0xfb	0x00	0x00	0x00	0x00
0x00	0x00 0xff	0x00	0x00	0x00 0x00	0x00	0x35	0xf2 0xff	0xfb	0x0a	0x6a	0xfa	0x0a	0x6a	0xfa	0x00
0x35	0xf2 0x00	0xfb	0x35	0xf2 0x01	0xfb	0x35	0xf2 0x01	0xfb	0x35	0xf2	0xfb	0x00	0x00	0x00	0x00
0x00	0x00	0x00	0x35	0xf2	0xfb	0x35	0xf2	0xfb	0x00	0x00	0x00	0x00	0x00	0x00	0x00

```
Sat((0x00 *  1)+(0x00 * 1)+(0xf2 *  0) +
    (0x00 * -1)+(0x00 * 0)+(0xf2 * -1) +
    (0xf2 *  0)+(0xf2 * 1)+(0xf2 *  1) =
```

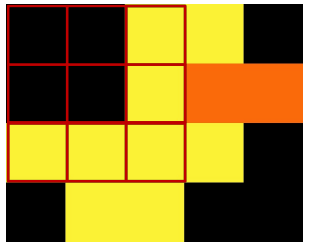
Sat(242) = 242 (0xf2)

outptr →

(*w*:3, *h*:2)

0x35	0xf2							

Example



Continue with 8-bit red byte

0x00	0x00	0x00 0x01	0x00	0x00	0x00 0x01	0x35	0xf2	0xfb 0x00	0x35	0xf2	0xfb	0x00	0x00	0x00	0x00
0x00	0x00	0x00 0xff	0x00	0x00	0x00 0x00	0x35	0xf2	0xfb 0xff	0x0a	0x6a	0xfa	0x0a	0x6a	0xfa	0x00
0x35	0xf2	0xfb 0x00	0x35	0xf2	0xfb 0x01	0x35	0xf2	0xfb 0x01	0x35	0xf2	0xfb	0x00	0x00	0x00	0x00
0x00	0x00	0x00	0x35	0xf2	0xfb	0x35	0xf2	0xfb	0x00	0x00	0x00	0x00	0x00	0x00	0x00

```
Sat((0x00 *  1)+(0x00 * 1)+(0xfb *  0) +
    (0x00 * -1)+(0x00 * 0)+(0xfb * -1) +
    (0xfb *  0)+(0xfb * 1)+(0xfb *  1) =
```

Sat(251) = 251 (0xfb)

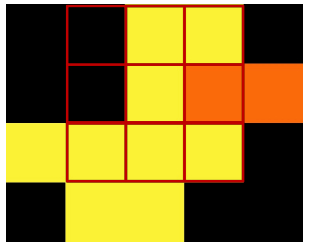
One pixel done!

outptr →

(*w*:3, *h*:2)

0x35	0xf2	0xfb						

Example



While calculating D[0][1] pixel's 8-bit green byte

0x00	0x00	0x00	0x00	0x00 0x01	0x00	0x35	0xf2 0x01	0xfb	0x35	0xf2 0x00	0xfb	0x00	0x00	0x00	0x00
0x00	0x00	0x00	0x00	0x00 0xff	0x00	0x35	0xf2 0x00	0xfb	0x0a	0x6a 0xff	0xfa	0x0a	0x6a	0xfa	0x00
0x35	0xf2	0xfb	0x35	0xf2 0x00	0xfb	0x35	0xf2 0x01	0xfb	0x35	0xf2 0x01	0xfb	0x00	0x00	0x00	0x00
0x00	0x00	0x00	0x35	0xf2	0xfb	0x35	0xf2	0xfb	0x00	0x00	0x00	0x00	0x00	0x00	0x00

```
Sat((0x00 *  1)+(0xf2 * 1)+(0xf2 *  0) +
    (0x00 * -1)+(0xf2 * 0)+(0x6a * -1) +
    (0xf2 *  0)+(0xf2 * 1)+(0xf2 *  1) =
```

Sat(620) = 255 (0xff)

if v > 255, Sat(v) = 255

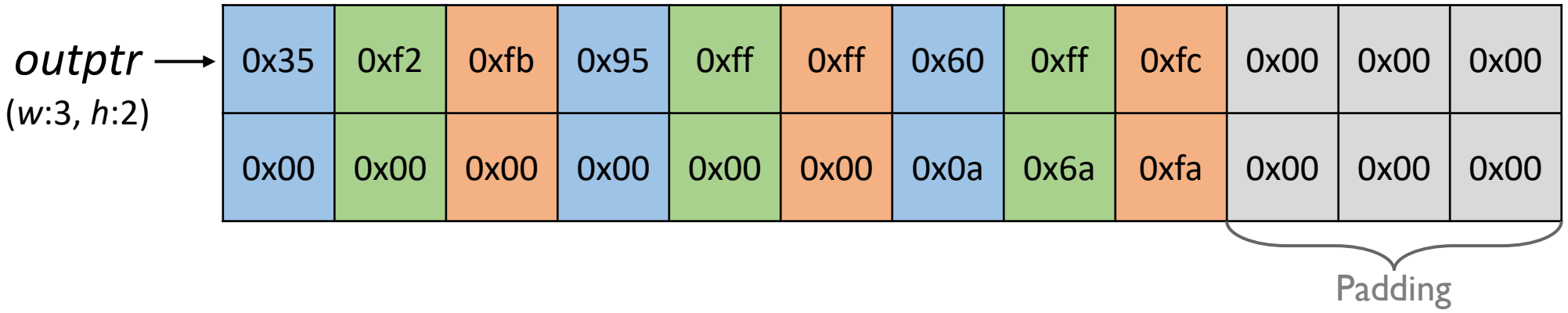
Saturation occurred in this calculation

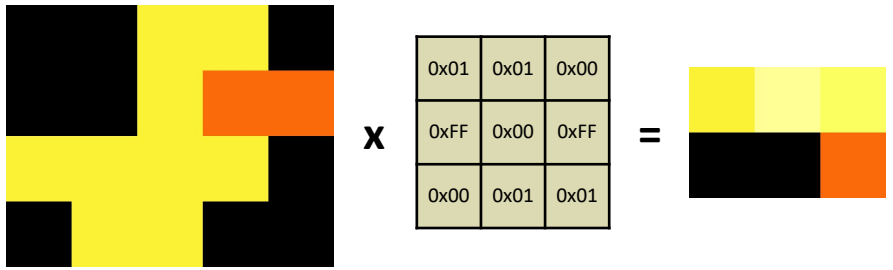
outptr → (*w*:3, *h*:2)

0x35	0xf2	0xfb	0x95	0xff				

Example

- In the bitmap data format, # of bytes occupied by each row should be a multiple of 4
- The remaining bytes should be padded with zeroes





Submission

- Due: 11:59PM, November 14 (Sunday)
  - 25% of the credit will be deducted for every single day delay

- Submit the bmpconv.s file to the submission server
  - Program that contains unallowed keywords or register names will be rejected

- The top 15 fastest implementations will receive a 10% extra bonus
- The next 15 fastest implementations will receive a 5% extra bonus

How to use PyRISC

PyRISC

- It provides various RISC-V toolset written in Python
- It has snurisc, a RISC-V instruction set simulator that supports most of RV32I base instruction set (32-bit version!)
- You should work on either Linux or MacOS
  - We highly recommend you to use Ubuntu 18.04 LTS or later
- For Windows, we recommend installing WSL(Windows Subsystem for Linux) and Ubuntu

PyRISC Prerequisites

- PyRISC toolset requires Python version 3.6 or higher.
- You should install Python modules(numpy, pyelftools)

For Ubuntu 18.04 LTS,

```
$ sudo apt-get install python3-numpy python3-pyelftools
```

For MacOS,

```
$ pip install numpy pyelftools
```

RISC-V GNU toolchain

- In order to work with the PyRISC toolset, you need to build a RISC-V GNU toolchain for the RV32I instruction set

- Please take the following steps to build it on your machine

Building RISC-V GNU toolchain

1. Install prerequisite packages

For Ubuntu 18.04 LTS,

```
$ sudo apt-get install autoconf automake autotools-dev curl libmpc-dev
$ sudo apt-get install libmpfr-dev libgmp-dev gawk build-essential bison flex
$ sudo apt-get install texinfo gperf libtool patchutils bc zlib1g-dev libexpat-dev
```

For MacOS,

```
$ brew install gawk gnu-sed gmp mpfr libmpc isl zlib expat
```

Building RISC-V GNU toolchain

2. Download the RISC-V GNU Toolchain from Github

```
$ git clone --recursive https://github.com/riscv/riscv-gnu-toolchain
```

3. Configure the RISC-V GNU toolchain

```
$ cd riscv-gnu-toolchain
$ mkdir build
$ cd build
$ ../configure --prefix=/opt/riscv --with-arch=rv32i
```

Building RISC-V GNU toolchain

4. Compile and install them

```
$ sudo make
```

5. Add `/opt/riscv/bin` in your PATH

```
$ export PATH=/opt/riscv/bin:$PATH
```

Running RISC-V executable file

- You should modify the `Makefile` in your ca-pa3 directory so that it can find the snurisc simulator

```
# in ca-pa3/Makefile
…

PYRISC = /dir1/dir2/pyrisc/sim/snurisc.py
PYRISCOPT = -l 1

…
```

Write the path where you downloaded pyrisc

Running RISC-V executable file

- Now, you can run your RISC-V executable file for assignment 3 by performing '`make run`'!

```
[csl@csl: ~/ca-pa3]$ make run
/home/csl/riscv-gnu-toolchain/pyrisc/sim/snurisc.py       -l 1 bmpconv
Loading file bmpconv
Execution completed
Registers
=========
zero ($0): 0x00000000    ra ($1):   0x8000017c    sp ($2):   0x80018000    gp ($3):   0x00000000
tp ($4):   0x00000000    t0 ($5):   0x000005a0    t1 ($6):   0x4c000000    t2 ($7):   0x00000000
s0 ($8):   0x00000000    s1 ($9):   0x00000004    a0 ($10):  0x80012d2c    a1 ($11):  0x00000000
a2 ($12):  0x000000bc    a3 ($13):  0x80017ff0    a4 ($14):  0x8001aa30    a5 ($15):  0x00000000
a6 ($16):  0x00000000    a7 ($17):  0x00000000    s2 ($18):  0x8001000c    s3 ($19):  0x8001001c
s4 ($20):  0x8001aa30    s5 ($21):  0x80015934    s6 ($22):  0x80015934    s7 ($23):  0x00000000
s8 ($24):  0x4c120404    s9 ($25):  0x00000000    s10 ($26): 0x00000000    s11 ($27): 0x00000000
t3 ($28):  0x00000000    t4 ($29):  0x00000000    t5 ($30):  0x00000000    t6 ($31):  0x00000000
1961374 instructions executed in 1961374 cycles. CPI = 1.000
Data transfer:    454314 instructions (23.16%)
ALU operation:    1186046 instructions (60.47%)
Control transfer: 321014 instructions (16.37%)
```

If t6 is 0 after executing all instructions, your output image for every test is correct

If t6 is not 0, t6 contains the index of the testcase you failed

Running RISC-V executable file

- When you failed to pass all test cases, check t5 to see where your program has failed

```
[csl@csl: ~/ca-pa3]$ make run
/home/csl/riscv-gnu-toolchain/pyrisc/sim/snurisc.py       -l 1 bmpconv
Loading file bmpconv
Execution completed
Registers
=========
zero ($0): 0x00000000    ra ($1):   0x80000040    sp ($2):   0x80017ffc    gp ($3):   0x00000000
tp ($4):   0x00000000    t0 ($5):   0x000005a0    t1 ($6):   0x4c000000    t2 ($7):   0x00000000
s0 ($8):   0x00000000    s1 ($9):   0x00000000    a0 ($10):  0x80012d2c    a1 ($11):  0x00000000
a2 ($12):  0x000000bc    a3 ($13):  0x80017ff0    a4 ($14):  0x8001aa30    a5 ($15):  0x00000000
a6 ($16):  0x00000000    a7 ($17):  0x00000000    s2 ($18):  0x80010008    s3 ($19):  0x80010018
s4 ($20):  0x80018020    s5 ($21):  0x80012f24    s6 ($22):  0x80015934    s7 ($23):  0x00020000
s8 ($24):  0x00000000    s9 ($25):  0x00000000    s10 ($26): 0x00000000    s11 ($27): 0x00000000
t3 ($28):  0x00000000    t4 ($29):  0x00000000    t5 ($30):  0x80018020    t6 ($31):  0x00000003
1945218 instructions executed in 1945218 cycles. CPI = 1.000
Data transfer:    448930 instructions (23.08%)
ALU operation:    1180659 instructions (60.70%)
Control transfer: 315629 instructions (16.23%)
```

failed to pass test 3

Your program failed to match the value output at memory address 0x80018020

Debugging Tips (1)

- If you want to see the values of registers after the specific instruction, insert 'ebreak' to stop the simulator

```
    .globl bmpconv
bmpconv:
    addi a0, zero, 1  # a0 = 1
    ebreak
    addi a0, zero, 2  # a0 = 2
    ret
```

```
[csl@csl: ~/ca-pa3]$ make run
/home/csl/riscv-gnu-toolchain/pyrisc/sim/snurisc.py      -l 1 bmpconv
Loading file bmpconv
Execution completed
Registers
=========
zero ($0): 0x00000000    ra ($1):   0x80000054    sp ($2):   0x80017ffc    gp ($3):   0x00000000
tp ($4):   0x00000000    t0 ($5):   0x00000000    t1 ($6):   0x00000000    t2 ($7):   0x00000000
s0 ($8):   0x00000000    s1 ($9):   0x00000001    a0 ($10):  0x00000001    a1 ($11):  0x00000004
a2 ($12):  0x00000005    a3 ($13):  0x80010020    a4 ($14):  0x80018000    a5 ($15):  0x00000000
a6 ($16):  0x00000000    a7 ($17):  0x00000000    s2 ($18):  0x80010000    s3 ($19):  0x80010010
s4 ($20):  0x00000000    s5 ($21):  0x00000000    s6 ($22):  0x00000000    s7 ($23):  0x00000000
s8 ($24):  0x00000000    s9 ($25):  0x00000000    s10 ($26): 0x00000000    s11 ($27): 0x00000000
t3 ($28):  0x00000000    t4 ($29):  0x00000000    t5 ($30):  0x00000000    t6 ($31):  0x00000000
18 instructions executed in 18 cycles. CPI = 1.000
Data transfer:    4 instructions (22.22%)
ALU operation:    11 instructions (61.11%)
Control transfer: 3 instructions (16.67%)
```

a0 is 1, not 2

Instructions after ebreak are not executed

Debugging Tips (2)

- You can change the log level by changing the number of **`PYRISCOPT`** in `ca-pa3/Makefile`

```
# in ca-pa3/Makefile
…

PYRISC = /dir1/dir2/pyrisc/sim/snurisc.py
PYRISCOPT = -l 1

…
```

Change this number

0: shows no output message
1: dumps registers at the end of the execution (default)
2: dumps registers and data memory at the end of the execution
3: 2 + shows instruction executed in each cycle
4: 3 + shows full information for each instruction
5: 4 + dumps registers for each cycle
6: 5 + dumps data memory for each cycle

Debugging Tips (3)

- You can add another option(-c) to **PYRISCOPT**

```
# in pa3/Makefile
…

PYRISC = /dir1/dir2/pyrisc/sim/snurisc.py
PYRISCOPT = -l 3 -c m

…
```

Shows logs after cycle m (default: 0)
Note that it is only effective for log level 3 or 4

Thank You!

- Don't forget to read the detailed description before you start your assignment

- If you have any questions about the assignment,
  feel free to ask via KakaoTalk

- This file will be uploaded after the lab session☺