Sunmin Jeong, Injae Kang

(snucsl.ta@gmail.com)

Systems Software &
Architecture Lab.

Seoul National University

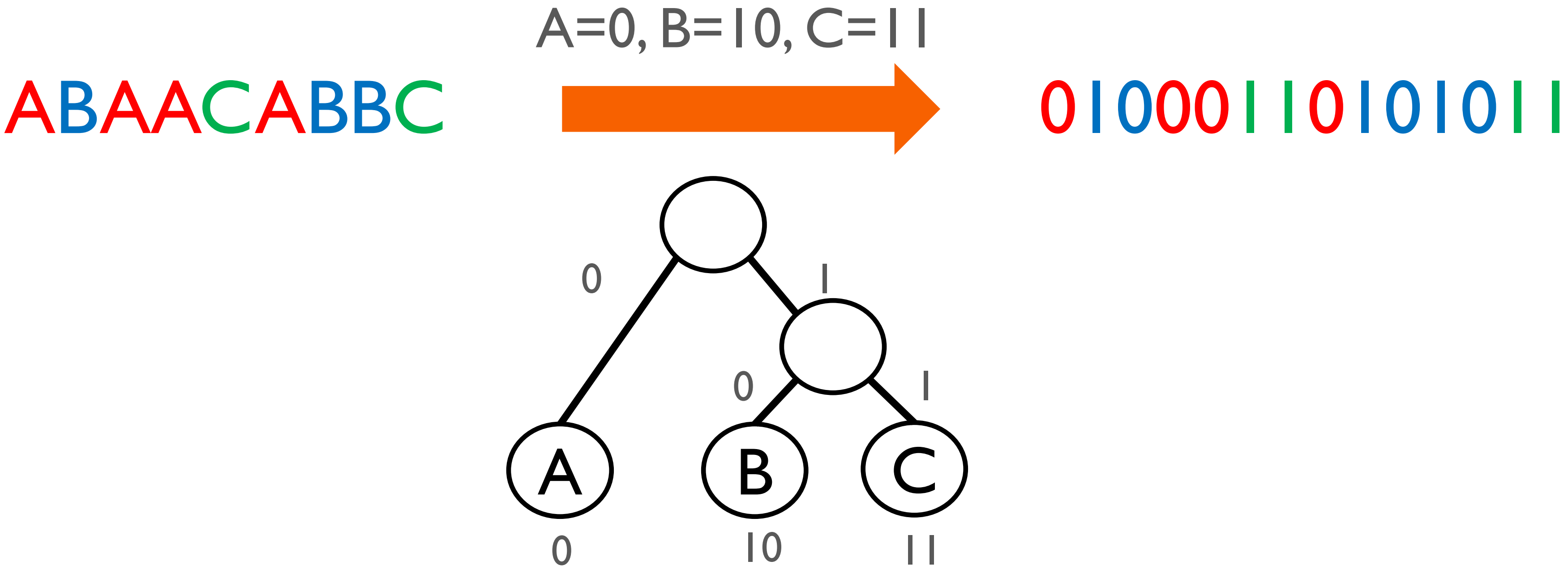
Fall 2020

4190.308:
Computer Architecture
Lab. 3

Huffman Coding (2)

What is Huffman Coding?

- An algorithm used for lossless data compression.
- It represents patterns with high frequency as short binary code.
  - Note that you don't have to know the algorithm in this assignment!

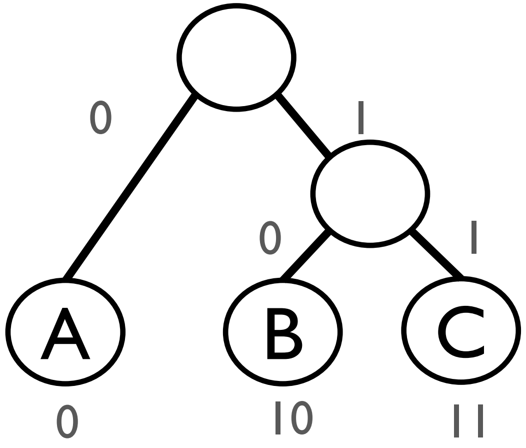


What is Huffman Coding?

- An algorithm used for lossless data compression.
- It represents patterns with high frequency as short binary code.
  - Note that you don't have to know the algorithm in this assignment!

0=A, 10=B, 11=C

ABAACABBC ← 010001101010011



In this assignment, you will decode
the result of assignment #1
to get the original data

Simplified Huffman Coding

- We can divide code into 3 groups(A, B and C) with the prefix
- You can find ranks of each code with the prefix and remaining bits

<Group A>
Starts with '0'
Length 3

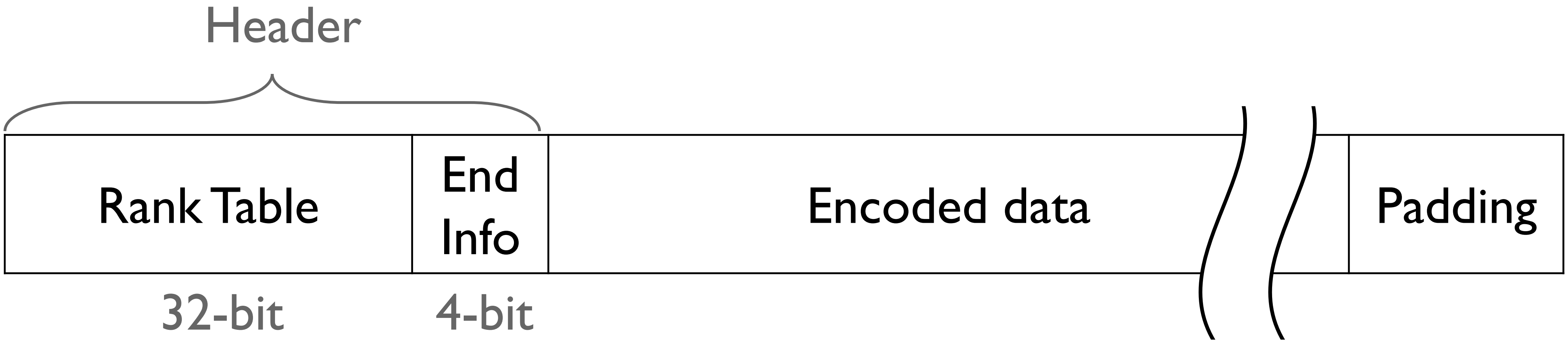
<Group B>
Starts with '10'
Length 4

Rank	Code	Rank	Code
0	000	8	11000
1	001	9	11001
2	010	10	11010
3	011	11	11011
4	1000	12	11100
5	1001	13	11101
6	1010	14	11110
7	1011	15	11111

<Group C>
Starts with '11'
Length 5

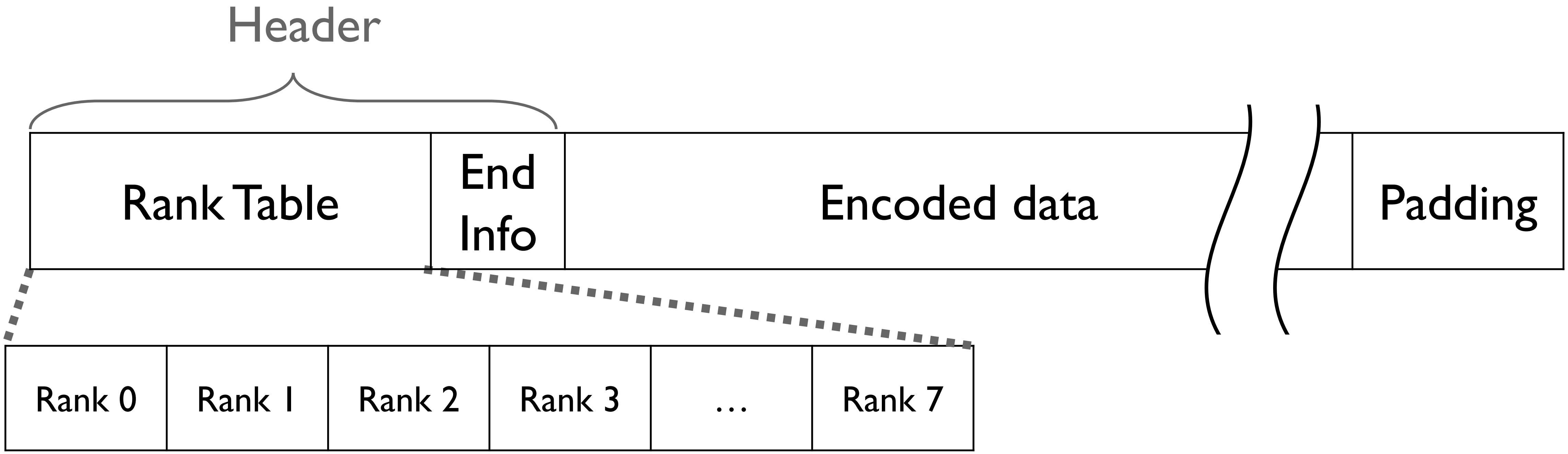
Simplified Huffman Coding

- The input is given in the following format.
- Input consists of Header, Encoded data and Padding.
- Header consists of 32-bit Rank Table and 4-bit End Info.



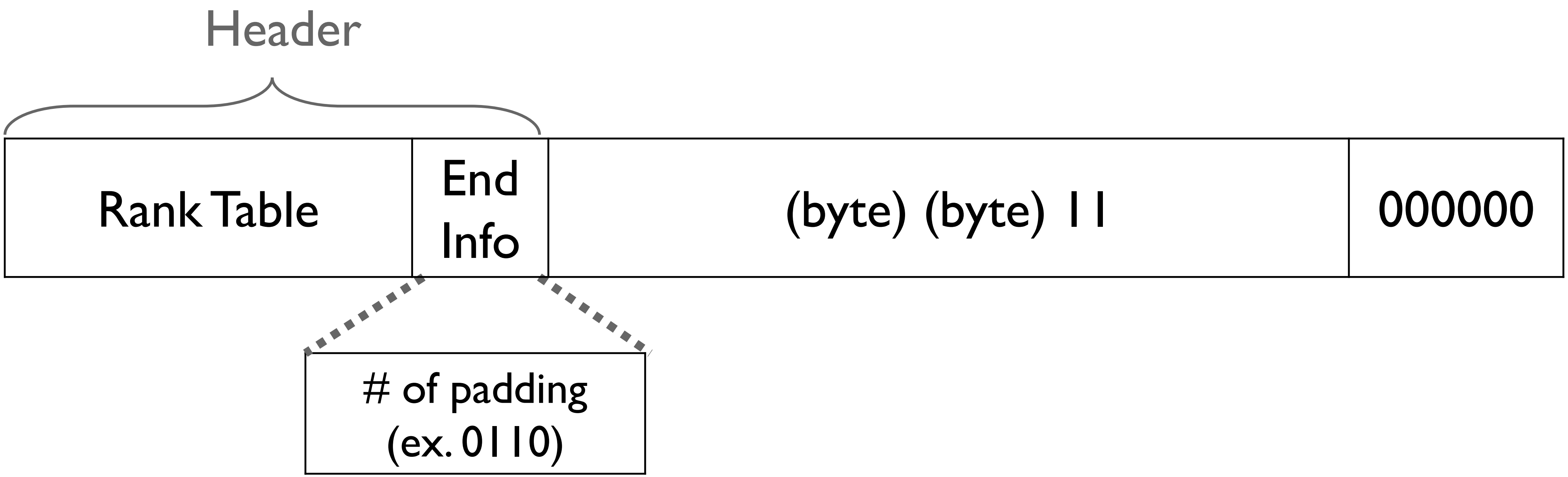
Simplified Huffman Coding

- Rank table has the information for the most frequent 8 symbols.



Simplified Huffman Coding

- End Info records the # of padding bits for byte alignment.
  - e.g., If the last byte of Encoded data ends with 11, 6 padding bits(0's) are added



Specification

- All you need to do is to write a function given in decode.s
- *int decode(const char *inp, int inbytes, char *outp, int outbytes)*
  - *inp* points to the memory address of the input compressed data
  - *inbytes* is the length of input data (in bytes)
  - *outp* points to the memory address for storing decoded data
  - *outbytes* is the length of allocated space for result
  - It returns the length of the output (in bytes)
    - If the length of output is bigger than *outbytes*, return -1
      (In this case, contents of the output is ignored)
    - If *inbytes* is 0 return 0

Specification

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

- You can assume that the size of the memory region allocated for *inp* and *outp* is a multiple of 4
  - Note that it is not the actual length of input/output data

Specification

- You should use only the following registers in decode.s
  : zero(x0), sp, ra, and a0~a5
  - Use stack as temporary storage if needed (**maximum 128 bytes**)

- When you store data to *outp*, you shouldn't write more than *outbytes*

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

Specification

- Your solution should finish within 5 seconds

- The top 10 or next top 10 fastest decode() implementations will receive a 10% or 5% extra bonus

Example(1)

Input stream(in 4-byte unit): 0x4513ac02 0x00208826 (*inbytes* = 7)

- Note that RISC-V uses little endian
  - Least significant byte has the smallest address

inp[0]	inp[1]	inp[2]	inp[3]
02	ac	13	45

- Change the format of input to big endian

→ 0x02ac1345 0x26882000

Example(1)

Input stream(in 4-byte unit): 0x4513ac02 0x00208826 (*inbytes* = 7)

- Input stream as big endian: 0x02ac1345 0x268820~~00~~

7 bytes

02 ac 13 45	2	6 88 20
32-bit	4-bit	

Example(1)

Input stream(in 4-byte unit): 0x4513ac02 0x00208826 (*inbytes* = 7)

- First 4-byte is Rank Table that records rank 0-7

02 ac 13 45	2	6 88 20
32-bit	4-bit	

Rank	0	1	2	3	4	5	6	7
Symbol	0000 (0)	0010 (2)	1010 (a)	1100 (c)	0001 (1)	0011 (3)	0100 (4)	0101 (5)

Example(1)

Input stream(in 4-byte unit): 0x4513ac02 0x00208826 (*inbytes* = 7)

- We can get remaining part of Rank Table by writing unused symbols in increasing order (not used in this example)

02 ac 13 45	2	6 88 20
32-bit	4-bit	

Rank	8	9	10	11	12	13	14	15
Symbol	0110 (6)	0111 (7)	1000 (8)	1001 (9)	1011 (b)	1101 (d)	1110 (e)	1111 (f)

Example(1)

Input stream(in 4-byte unit): 0x4513ac02 0x00208826 (*inbytes* = 7)

- Decode the data with Rank Table

Starts with '0' → length 3, rank 0-3

02 ac 13 45	2	0110 1000 1000 0010 00~~00~~
32-bit	4-bit	r[3] r[2] r[1] r[0] r[1] r[0]

- Result: ca 20 20

Example(1)

Input stream(in 4-byte unit): 0x4513ac02 0x00208826 (*inbytes* = 7)

- Result: ca 20 20 ~~00~~

- Store the result to *outp* as little endian format in 4-byte word

→ 0x~~00~~2020ca

- Return the length of decoded data → 3

Example(2)

Input stream(in 4-byte unit): 0x12af370b 0x9a3f2328 0x15f92189 0x00006c63 (*inbytes* = 14)

- Input stream as big endian: 0x0b37af12 0x28233f9a 0x8921f915 0x636c~~0000~~

0b 37 af 12	2	8 23 3f 9a 89 21 f9 15 63 6c
32-bit	4-bit	

Example(2)

Input stream(in 4-byte unit): 0x12af370b 0x9a3f2328 0x15f92189 0x00006c63 (*inbytes* = 14)

Rank	0	1	2	3	4	5	6	7
Symbol	0000 (0)	1011 (b)	0011 (3)	0111 (7)	1010 (a)	1111 (f)	0001 (1)	0010 (2)
Rank	8	9	10	11	12	13	14	15
Symbol	0100 (4)	0101 (5)	0110 (6)	1000 (8)	1001 (9)	1100 (c)	1101 (d)	1110 (e)

0b 37 af 12	2	8 23 3f 9a 89 21 f9 15 63 6c
32-bit	4-bit	

Example(2)

Input stream(in 4-byte unit): 0x12af370b 0x9a3f2328 0x15f92189 0x00006c63 (*inbytes* = 14)

Starts with '0' → length 3, rank 0-3
Starts with '10' → length 4, rank 4-7
Starts with '11' → length 5, rank 8-15

0b 37 af 12	2	1000 0010 0011 0011 1111 1001 1010 …
32-bit	4-bit	r[4] r[1] r[0] r[9] r[15] r[5] r[6]

Example(2)

Input stream(in 4-byte unit): 0x12af370b 0x9a3f2328 0x15f92189 0x00006c63 (*inbytes* = 14)

Starts with '0' → length 3, rank 0-3
Starts with '10' → length 4, rank 4-7
Starts with '11' → length 5, rank 8-15

1000 1001 0010 0001 1111 1001 0001 0101 0110 0011 0110 11~~00~~

r[4] r[5] r[1] r[0] r[3] r[14] r[2] r[1] r[2] r[7] r[0] r[11] r[3]

Example(2)

Input stream(in 4-byte unit): 0x12af370b 0x9a3f2328 0x15f92189 0x00006c63 (*inbytes* = 14)

- Result: ab 05 ef 1a fb 07 d3 b3 20 87 ~~00 00~~

- Store the result to *outp* as little endian format in 4-byte word

→ 0x1aef05ab 0xb3d307fb 0x~~0000~~8720

- Return the length of decoded data → 10

Submission

- Due: 11:59PM, November 8 (Sunday)
  - 25% of the credit will be deducted for every single day delay

- Submit the decode.s file to the submission server

Submission

- You should write a 1-2 page report to explain your assembly implementation of decode() function
  - It will account for 10% of your score in this assignment
  - It will be graded as pass or fail

- Submit the report.pdf file to the submission server

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 18.04LTS or later

- For Windows, we recommend you to install WSL(Windows Subsystem for Linux) and Ubuntu

PyRISC

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

For Ubuntu 18.04LTS,

```
$ 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

RISC-V GNU toolchain

1. Install prerequisite packages

For Ubuntu 18.04LTS,

```
$ 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
```

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
```

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 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 you downloaded pyrisc

Running RISC-V executable file

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

```
sunmin@sunmin-Z490-VISION-G:~/pa3$ make run
/home/sunmin/CA/pyrisc/sim/snurisc.py   -l 1 decode
Loading file decode
Execution completed
Registers
=========
zero ($0): 0x00000000    ra ($1):   0x80000384    sp ($2):   0x80020000    gp ($3):   0x00000000
tp ($4):   0x00000000    t0 ($5):   0x00000006    t1 ($6):   0x00000006    t2 ($7):   0x80010018
s0 ($8):   0x00000000    s1 ($9):   0x00000004    a0 ($10):  0x0000002c    a1 ($11):  0x0000002c
a2 ($12):  0x00000000    a3 ($13):  0x00000000    a4 ($14):  0x00000156    a5 ($15):  0x00000000
a6 ($16):  0x00000000    a7 ($17):  0x00000100    s2 ($18):  0x8001ff18    s3 ($19):  0x80010130
s4 ($20):  0x2e676f64    s5 ($21):  0x2e676f64    s6 ($22):  0x00000004    s7 ($23):  0x000000ff
s8 ($24):  0x00000000    s9 ($25):  0x00000000    s10 ($26): 0x00000000    s11 ($27): 0x00000000
t3 ($28):  0x80010030    t4 ($29):  0x80010108    t5 ($30):  0x0000002c    t6 ($31):  0x00000000
17521 instructions executed in 17521 cycles. CPI = 1.000
Data transfer:    995 instructions (5.68%)
ALU operation:    12402 instructions (70.78%)
Control transfer: 4124 instructions (23.54%)
```

You passed the test if t6 is 0 after executing all instructions

Debugging tips

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

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

```
sunmin@sunmin-Z490-VISION-G:~/pa3$ make run
/home/sunmin/CA/pyrisc/sim/snurisc.py   -l 1 decode
Loading file decode
Execution completed
Registers
=========
zero ($0): 0x00000000    ra ($1):   0x80000410    sp ($2):   0x8001fef0    gp ($3):   0x00000000
tp ($4):   0x00000000    t0 ($5):   0x00000000    t1 ($6):   0x00000006    t2 ($7):   0x80010000
s0 ($8):   0x00000000    s1 ($9):   0x00000004    a0 ($10):  0x00000001    a1 ($11):  0x00000007
a2 ($12):  0x8001fef0    a3 ($13):  0x00000100    a4 ($14):  0x00000000    a5 ($15):  0x00000000
a6 ($16):  0x00000000    a7 ($17):  0x00000000    s2 ($18):  0x00000000    s3 ($19):  0x00000000
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):  0x80010018    t4 ($29):  0x80010030    t5 ($30):  0x00000000    t6 ($31):  0x00000000
21 instructions executed in 21 cycles. CPI = 1.000
Data transfer:    3 instructions (14.29%)
ALU operation:    15 instructions (71.43%)
Control transfer: 3 instructions (14.29%)
```

a0 is 1, not 2
Instructions after ebreak weren’t executed

Debugging tips

- 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

- 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!

- If you have any question about the assignment, feel free to ask us in email or KakaoTalk

- This file will be uploaded after the lab session☺