2GA3 Tutorial #3

DATE: October 1st, 2021

TA: Jatin Chowdhary

Pay Attention

- Next slide is very important
- Listen to this carefully
  - This information might save your life one day
    - Seriously

FIPPA

- What is FIPPA?
  - Freedom of Information and Protection of Privacy Act
- Why care?
  - Use it for secure/confidential communication
    - Great for disclosing information you do not want anyone else but the intended party to see
      - i.e. Allergies (Food, chemicals, etc.)
    - i.e. Inform on person X to person Y
- How to properly use it?
  - By law it's already used to protect communication
    - But it's good practice to state it
  - i.e. On next slide

FIPPA Example

Dear Professor X,

I am writing this email to inform you about an incident that occurred on March 3rd, 2017, with a person who goes by the name of Logan. Mr. Logan consumed a peanut butter and jelly sandwich while being fully aware of my food allergies. Had I not taken the appropriate action to protect myself, serious harm could have occurred. Please take necessary action to alleviate this issue. Furthermore, I expect the contents of this email to remain confidential in accordance to **<u>FIPPA</u>**. Please do not disclose my identity, or any other identifying information to any party involved directly or indirectly, without my consent.

Regards,

Scott Summers

Participation

- The script
  - Everyone will be called upon, once
- Why?
  - This is the best way to learn
  - Repitition is the language of the brain
    - Speech > Writing
    - It's how our brain are wired
      - Millions of years of evolution
  - Participating > Taking notes
    - Everything will be posted
    - Last tutorial on Friday is recorded
      - Just pay attention and (immediately) ask questions!

I Don't Know

- If I call on you, and you don't know, just say, "I don't know"
  - Ain't nothing wrong with this
  - Everyone is here to learn
  - Proudly say, "I don't know"
    - You'll learn better this way; it takes away the pressure
    - It's the best answer
- If you still don't get it, ask for clarification!
  - No one gets ***it*** right away
    - It takes a few tries, but eventually you will get it, and that's all that matters

Clarification (1)

- Made a mistake about ***sw*** and ***sd*** during last week's tutorial
  - The mistake is underlined below:

64-Bit:

```
sub x30, x28, x29    // Compute [i – j]
slli x30, x30, 3     // Multiply by 8 to get byte offset
add x3, x3, x30      // Calculate location in memory
ld x30, 0(x3)        // Load A[i – j]
sd x30, 64(x11)      // Store in B[8]
```

32-Bit:

```
sub x30, x28, x29    // Compute [i – j]
slli x30, x30, 2     // Multiply by 4 to get byte offset
add x3, x3, x30      // Calculate location in memory
lw x30, 0(x3)        // Load A[i – j]
sw x30, 32(x11)      // Store in B[8]
```

Clarification (2)

- I can't remember what I said, but:
  - The difference in 64-bit and 32-bit RISC-V code (in the previous slide) is:
    - **slli**
      - 3 vs. 2
    - **ld** vs. **lw**
    - **sd** vs. **sw**
    - <u>**64**</u>(x11) vs. <u>**32**</u>(x11)
      - The number to the left of the brackets is the byte offset in memory
        - The data is stored memory that corresponds to the value in ***x11*** (base address), with an offset of 32 or 64
      - Why are 8 (64-bit) and 4 (32-bit) so special?
        - Explained on next slide

Mistake

- What I said last week was a bit misleading
  - Does everything make sense?
- Sorry for the mistake
  - If you make a mistake, come clean
    - Great way to learn, because you'll never forget
  - **Be honest**
- We are all here to learn

Word

- Question: How big is a *word*?
- Options:
  - 16-bits
  - 8-bits
  - 31-bits
  - 32-bits
  - 33-bits
  - None of the above
  - As big as you want

Double Word

- Question: How big is a *double word*?
- Options:
  - 60-bits
  - 61-bits
  - 62-bits
  - 63-bits
  - All of the above
  - None of the above

Byte Offset (1)

- In a 32-bit system, the byte offset is calculated by multiplying by **4**
  - For 64-bit, multiply by **8**
- Why these specific numbers?
  - First of all, what does 32-bit mean?
    - Question: If I say the architecture of this computer is 32-bits, what does that mean?
    - Question: What is 1 bit?
      - In the context of binary numbers
    - How many bits in a byte?
  - Calculation
    - Next slide

Byte Offset (2)

- For a 32-bit system, we divide 32 bits by 8 bits to get the size of the block, in bytes
  - 32 bits / 8 bits = 4 bytes
- What is the size of the block, in bytes, for a 64-bit system?
  - ***X*** bits / ***Y*** bits = ***Z*** bytes

128-Bit Architecture

- Question: To calculate the byte offset in a 128-bit computer, what number would you use to multiply?
- Options:
  - 12 bytes
  - 14 bytes
  - 16 bytes
  - 18 bytes
  - 20 bytes

Any Questions?

- Does that make sense?
- All good?
- Redo it?

Food

- If you have food allergies, let me know!
  - Same with dietary restrictions
    - Use FIPPA
- Anyone hungry?
  - Snacks in my bag
    - Whoever asks the most questions, gets a piece

Recap SLLI (1)

- slli = shift left logical immediate
  - Example for Base 10:
    - Perform slli, once, on (999)10
      - Answer: Multiply 999 by 10, because we are dealing with base 10 numbers
    - Perform slli, twice, on (200)10
      - Answer: Multiply X by Y
        - Help me out!
  - General rule?
    - Next slide

Recap SLLI (2)

- General rule is:
  - For every *slli*, add a 0
    - Translation: Multipy by ***X***, where ***X*** is the base of the numbers you are working with
      - i.e. (slli, 1) on $(900)_{10}$ = 900 x **10** = 9000
      - i.e. (slli, 2) on $(11)_2$ = 11 x **2** x **2** = 01100
        - Note: This is <u>NOT</u> "eleven"
    - Note: Only use this trick for $(\text{Base})_2$

More About SLLI

- Going overboard!
  - Assume you are working with a 4-bit architecture, where the maximum possible number is $(1111)_2$.
    - What is the result of: *(slli, 2) on* $(1001)_2$*?*
      - Remember: The largest possible number is $(1111)_2$
      - Is the answer: $(100100)_2$
- Conclusion: Information is _ _ _ _ !
  - Same thing happens for *srli*
    - *srli* = Shift Right Logical Immediate

Any Questions?

- Does that make sense?
    - Any questions about anything I've said?
- Ask now or forever...

No Time

- I still gotta clarify:
  - CMOS/PMOS/NMOS
    - *Note: Was discussed during office hours*
  - 32-Bit vs. 64-Bit
  - 'strcpy' Example
- Always running out of time
  - Will try to cover next week
    - Or the week after that
      - Or after that
        - Or after that

Question #1 (2.13)

- For the following instruction, what is the instruction type? And what is the hexadecimal representation?

  sw x5, 32(x30)

  rs2 imm rs1

- Answer:
  - Next slide
    - Pay attention; this is **VERY** important

Answer #1 (2.13)

- What is the instruction type?
  - The instruction is ***sw***, which stands for *store word*
    - Hence, it is a _ _ _ _ _ instruction, making it an _ type
- Refer to lecture slides:

RISC-V Instruction Types

R-type (R for registers) instructions use 3 register operands (2 source registers and one destination register).

I-type (I for immediate) instructions use 2 register operands (1 source, 1 destination) and one 12-bit immediate.

S-type (S for stores) instructions use 2 register operands (2 source) and one 12-bit immediate.

SB-type (conditional Branch, fields like the S)

U-type (Upper immediate format)

UJ-type (Unconditional jump)

29

Answer #1 (2.13)

- What is the hexadecimal representation?
  - First, get the binary representation
    - Then, convert to hexadecimal
- Answer:
  - We've already established that we are dealing with an S-type instruction
    - So let's see what the format is for an S-type instruction
      - Next slide

RISC-V S-format Instructions

immediate[11:5]	rs2	rs1	funct3	immediate[4:0]	opcode
7 bits	5 bits	5 bits	3 bits	5 bits	7 bits

- Different **immediate format for store instructions**
  - rs1: base address register number
  - rs2: source operand register number
  - immediate: offset added to base address
    - Split so that rs1 and rs2 fields always in the same place

Answer #1 (2.13)

- So, we need:
  - *opcode (7 bits)*
  - *funct3 (3 bits)*
  - *rs1 (5 bits)*
  - *rs2 (5 bits)*
  - *immediate (5 + 7 bits)*
- How do we get this information?
  - From the RISC-V instruction list!
    - opcode = 0100011
    - funct3 = 010
    - rs1 = $(30)_{10}$
      - $(011110)_2 \rightarrow (11110)_2$
    - rs2 = $(5)_{10}$
      - $(101)_2 \rightarrow (00101)_2$
    - Immediate = $(32)_{10}$
      - $(100000)_2$
- Now what?
  - Next slide

Answer #1 (2.13)

- Put it all together!
  - opcode = 0100011
  - funct3 = 010
  - rs1 = 11110
  - rs2 = 00101
  - immediate = 100000
    - Immediate[4:0] = 00000
      - [4:0] means: Take the first 5 least significant bits; from the 0^{th} bit to the 4^{th} bit
    - Immediate [5:11] = 0000001
      - [5:11] means: The range is from the 5^{th} bit to the 11^{th} bit
- The final answer is:
  - *immediate[5:11]* + *rs2* + *rs1* + *funct3* + *immediate[4:0]* + *opcode*
    - 0000001 00101 11110 010 00000 0100011

Answer #1 (2.13)

- How to convert to hexadecimal?

  1. Take our answer and remove all spaces:

     00000010010111110010000000100011

  2. Add a space after every 4th bit:

     0000 0010 0101 1111 0010 0000 0010 0011

  3. Use the table, on the right, to convert to hex:

     0x**025f2023**

0000	0	1000	8
0001	1	1001	9
0010	2	1010	A
0011	3	1011	B
0100	4	1100	C
0101	5	1101	D
0110	6	1110	E
0111	7	1111	F

Clarification For #1 (2.13)

- **Question:** Why is the immediate value separated into 2 parts?
  - **Answer:** To keep the format of the instructions consistent, which makes decoding easier
    - For more information: See lecture slides
- **Question:** What is *0x*?
  - **Answer:** It is convention to prefix hexadecimal numbers with *0x*. It tells the compiler (and you) that the number is in hexadecimal
    - i.e. 0xFFFF
      - Hexadecimal
    - i.e. 0b11101
      - Binary
    - i.e. 0o35
      - Octal
  - We denote different bases via subscripts, the compiler/parser uses *0x*

The Big Picture!

- Everything in a computer's memory is just 1's and 0's
  - i.e. 0000001001011111001000000000100011
    - Registers, Cache, Main memory, HDD, etc. contain of 0's and 1's
- So when you send an instruction to the CPU, you're just sending a bunch of 0's and 1's like the example above
  - The 1's and 0's are generated by the *waves* *(Draw This)*
    - *Waves* = Low voltage and high voltage
  - The more waves there are, the more _ _ _ _ we can send
    - The _ _ _ _ are sent from main memory to the CPU through the _ _ _
      - The _ _ _ _ _ rate of a CPU measures the "_ _ _ _ _"

Question #2 (2.17)

- Assume that the following registers contain:
  - ***x5*** = 0x0000AAAA
  - ***x6*** = 0x12345678
- For the register values shown above, what is the value of ***x7*** after the following sequence of instructions?
  - ***slli*** *x7, x5, 4*
  - ***or*** *x7, x7, x6*
- For the register values shown above, what is the value of ***x7*** after the following sequence of instructions?
  - ***srli*** *x7, x5, 3*
  - ***andi*** *x7, x7, 0x1EF*

Answer #2 (2.17)

- ***slli*** *x7, x5, 4*

  = 0x0000AAAA × (2 x 2 x 2 x 2)

  = 0x0000AAAA × (16)

  = 0x000AAAA0

- ***or*** *x7, x7, x6*

  0x**000<u>AAAA0</u>** OR 0x**12345678**

000AAAA0:	**0 0 0 0 0 0 0 0 0 0 0 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 0 0 0 0**
12345678:	**0 0 0 1 0 0 1 0 0 0 1 1 0 1 0 0 0 1 0 1 0 1 1 0 0 1 1 1 1 0 0 0**
0123efef8:	**0 0 0 1 0 0 1 0 0 0 1 1 1 1 1 0 1 1 1 1 1 1 1 0 1 1 1 1 1 0 0 0**

Answer #2 (2.17)

- ***srli*** *x7, x5, 3*

  = 0x0000AAAA ÷ (2 x 2 x 2)

  = 0x0000AAAA ÷ (8)

  = 0x00001555

- ***andi*** *x7, x7, 0x1EF*

  = 0x00001555 *AND 0x000001EF*

  = 1555 *AND* 01EF

  = 0???

  = 0145

Hexadecimal	***Binary***
555	**0101 0101** 0101
1EF	**0001 1111** 0 1111
145	**0001 0100** 0 0101

Clarification For #2 (2.17)

- **Question:** Do I have to convert to binary, perform the operation and then convert back?
  - **Answer:** There's a trick to performing these operations for $base_{16}$ numbers
    - Because $base_2$ numbers are 2^1 and $base_{16}$ numbers are 2^4
      - $2^4 = (2^1)^4$
  - You should try to figure it out, before asking me to spill the beans
    - Practice as much as you can!

Question #3 (2.23)

- Homework
  - You can find all questions/solutions on Teams
    - Check the *Files* tab
  - **Don't look at the answer before starting**

Question #4 (2.23)

- For the following *C* statement, write a minimal sequence of RISC-V assembly instructions that performs the identical operation.

  Assume *x6* = *A*, and *x17* is the base address of *C*.

  A = C[0] << 4; ***// C Code***

Question #4 (2.23)

- First of all:
  - What is ***A*** ?
  - What is ***C[0]*** ?
    - Where is ***C[0]*** stored?
      - Which instruction do we use?
  - What is **<<** ?
    - What is the equivalent instruction for this?

Answer #4 (2.23)

- Solution:

ld x6, 0(x17) *// x6 = C[0]*

slli x6, x6, 4 *// x6 = x6 * 16*