NCDC

NUST CHIP DESIGN CENTRE

Computer Architecture

RISCV - Data Transfer and Upper Immediate Instructions

Week 5

Agenda

- Main Memory
- Data Transfer Instructions
- C Code Example
- Upper Immediate Instructions

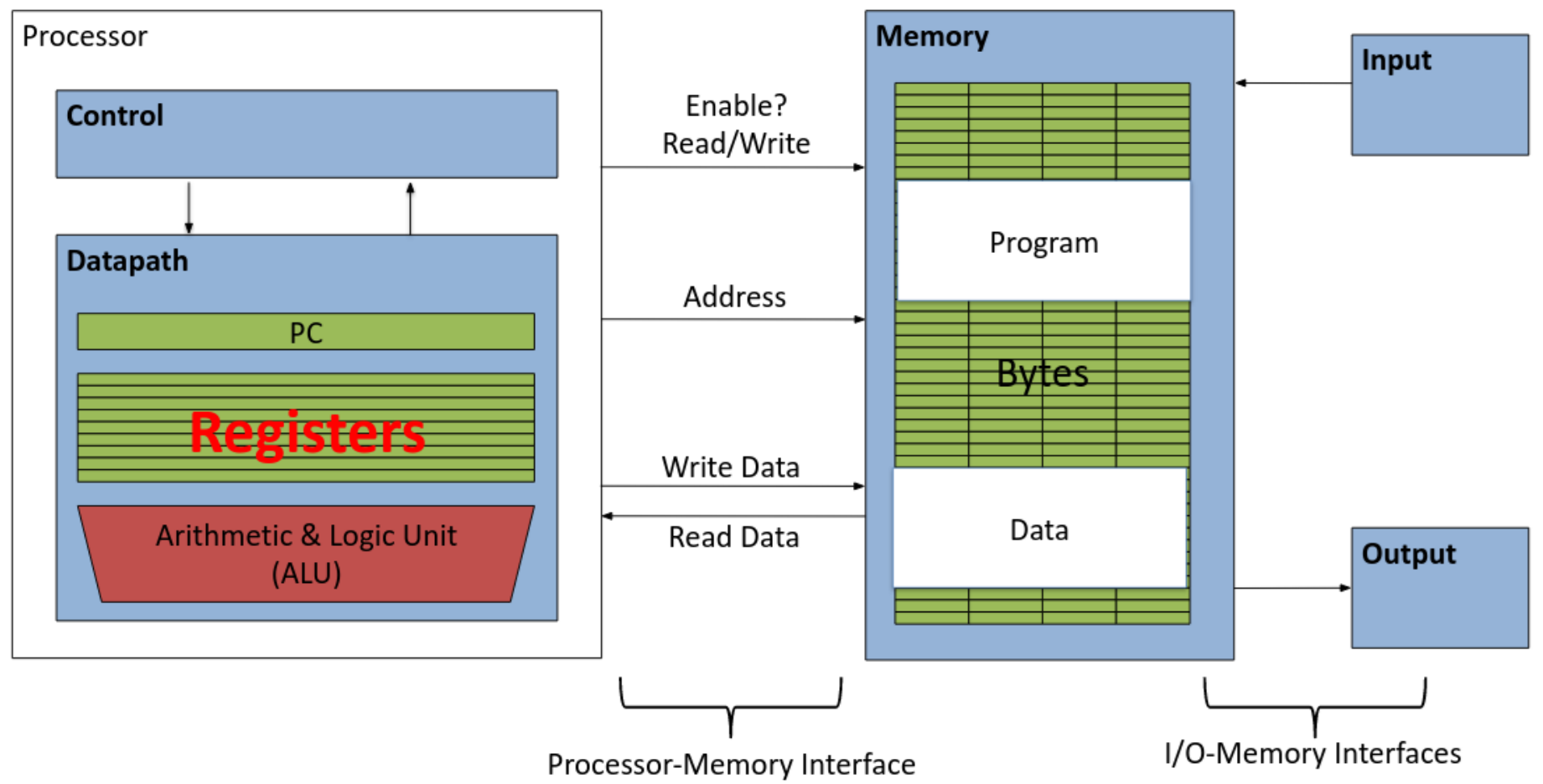
Agenda

- **Main Memory**
- Data Transfer Instructions
- C Code Example
- Upper Immediate Instructions

RV32 So Far…

op	funct3	funct7	Type	Instruction	Description	Operation
0110011 (51)	000	0000000	R	`add rd, rs1, rs2`	add	`rd = rs1 + rs2`
0110011 (51)	000	0100000	R	`sub rd, rs1, rs2`	sub	`rd = rs1 - rs2`
0010011 (19)	000	–	I	`addi rd, rs1, imm`	add immediate	`rd = rs1 + SignExt(imm)`
0110011 (51)	110	0000000*	R	`or rd, rs1, rs2`	or	`rd = rs1 | rs2`
0110011 (51)	111	0000000	R	`and rd, rs1, rs2`	and	`rd = rs1 & rs2`
0110011 (51)	100	0000000	R	`xor rd, rs1, rs2`	xor	`rd = rs1 ^ rs2`
0010011 (19)	110	–	I	`ori rd, rs1, imm`	or immediate	`rd = rs1 | SignExt(imm)`
0010011 (19)	111	–	I	`andi rd, rs1, imm`	and immediate	`rd = rs1 & SignExt(imm)`
0010011 (19)	100	–	I	`xori rd, rs1, imm`	xor immediate	`rd = rs1 ^ SignExt(imm)`
0110011 (51)	001	0000000*	R	`sll rd, rs1, rs2`	shift left logical	`rd = rs1 << rs2`$_{4:0}$
0110011 (51)	101	0000000*	R	`srl rd, rs1, rs2`	shift right logical	`rd = rs1 >>` $rs2_{4:0}$
0110011 (51)	101	0100000	R	`sra rd, rs1, rs2`	shift right arithmetic	`rd = rs1 >>>` $rs2_{4:0}$
0010011 (19)	001	0000000	I	`slli rd, rs1, uimm`	shift left logical immediate	`rd = rs1 << uimm`
0010011 (19)	101	0000000	I	`srli rd, rs1, uimm`	shift right logical immediate	`rd = rs1 >> uimm`
0010011 (19)	101	0100000*	I	`srai rd, rs1, uimm`	shift right arithmetic imm.	`rd = rs1 >>> uimm`

Data Transfer: Load from and Store to memory



Very fast, but limited space to hold values!

Review: How memory works

- On a 32-bit system, main memory contains 2^{32} bytes. Every 32-bit number acts as the **address** of one byte.
- 4 bytes together make a **word**. In order to store a word in memory, we cut the word up into 4 bytes, then store those bytes in consecutive addresses. In order to read a word, we read 4 consecutive addresses, then stitch those bytes back together.
- RISC-V uses little-endian to store data: The least significant byte gets stored at the lowest address.
- By convention we say that a word is stored at its lowest address
  - Ex. The word stored at address 0x1000 is composed of the bytes 0x1000, 0x1001, 0x1002, and 0x1003

Agenda

- Main Memory
- **Data Transfer Instructions**
- C Code Example
- Upper Immediate Instructions

Loading and Storing Data

- Data transfer instructions load operands from memory and store result to memory
- Format
  - *inst_name reg, offset(base address in register)*

- Loading data
  - Load word (load 32-bit data)
    - lw x7, 0(x6)
    - x7 = Memory[x6+0]
    - Registers are 32-bit in 32-bit implementation!

Load Word

Load Word syntax:

- `lw rd imm(rs1)`

Means: Compute `imm+rs1`, then load the 4 bytes at that address into rd

Byte (0x)	EF	BE	AD	DE	43	53	36	31	43	20	52	49	53	42	56	00
Address (0x)	100	101	102	103	104	105	106	107	108	109	10A	10B	10C	10D	10E	10F

Load Word

Example: `lw x10 12(x5)` if x5 is 0x100

- 0x100+12 = 0x10C
- Bytes at 0x10C-0x10F are 0x53, 0x42, 0x56, 0x00
- Since RISC-V is little-endian, this is the 32-bit value 0x0056 4253
- So register x10 will now store 0x0056 4253

Byte (0x)	EF	BE	AD	DE	43	53	36	31	43	20	52	49	53	42	56	00
Address (0x)	100	101	102	103	104	105	106	107	108	109	10A	10B	10C	10D	10E	10F

Exercise …

- Given following memory contents (in hexadecimal):

Address	Data
0	FF
1	12
2	6D
3	12
4	45
5	23
6	00
7	1E

- Let's assume x6 = 3
  - What will be in x7 after executing the following instruction?
    - lw x7, 0(x6)
      - x7 = 0x00234512
      - 12 (in hex) is the least-significant byte!

Store Word

Store Word syntax:

- `sw rs2 imm(rs1)`

Means: Compute `imm+rs1`, then store the 4 bytes of rs2 into that address.

Byte (0x)	EF	BE	AD	DE	43	53	36	31	43	20	52	49	53	42	56	00
Address (0x)	100	101	102	103	104	105	106	107	108	109	10A	10B	10C	10D	10E	10F

Exercise …

- Given following memory contents (in hexadecimal):
- x6 = 2 and x7 = 0x67854309

Address	Data
0	FF
1	12
2	6D
3	12
4	45
5	23
6	00
7	1E

Data is stored in little-endian order!

- What will be the memory contents after executing the following instruction?
  - sw x7, 0(x6)

Address	Data
0	FF
1	12
2	09
3	43
4	85
5	67
6	00
7	1E

Loading and Storing Bytes

Example: `sw x10 0(x5)` if x5 is 0x100, x10 is 0x1234 5678

- 0x100+0 = 0x100
- Since RISC-V is little-endian, the 32-bit value 0x1234 5678 gets split into bytes 0x78 0x56 0x34 0x12
- So the bytes in memory 0x100, 0x101, 0x102, and 0x103 get set to 0x78, 0x56, x34, and 0x12, respectively.

Byte (0x)	78	56	34	12	43	53	36	31	43	20	52	49	53	42	56	00
Address (0x)	100	101	102	103	104	105	106	107	108	109	10A	10B	10C	10D	10E	10F

Loading and Storing Bytes

In addition to word data transfers (lw, sw), RISC-V has byte data transfers:

- `lb rd imm(rs1)`
- `sb rs2 imm(rs1)`

Load and store one byte instead of a full word

Problem: If registers contain 4 bytes, how do we load/store only 1 byte?

Byte (0x)	EF	BE	AD	DE	43	53	36	31	43	20	52	49	53	42	56	00
Address (0x)	100	101	102	103	104	105	106	107	108	109	10A	10B	10C	10D	10E	10F

Store Byte

Example: `sb x10 0(x5)` if x5 is 0x100, x10 is 0x1234 5678

For `sb`, we store the *least significant byte*.

In the above example, 0x78 is the LSB.

So x100 gets set to 0x78

Byte (0x)	78	BE	AD	DE	43	53	36	31	43	20	52	49	53	42	56	00
Address (0x)	100	101	102	103	104	105	106	107	108	109	10A	10B	10C	10D	10E	10F

Load Byte

Example: `lb x10 0(x5)` if x5 is 0x100

For `lb`, we *extend* the numeric value to a full 32 bits.

In the above example, we load the number 0xEF. What 32-bit number has the same numeric value as 0xEF?

Answer: It depends on your representation scheme

Byte (0x)	EF	BE	AD	DE	43	53	36	31	43	20	52	49	53	42	56	00
Address (0x)	100	101	102	103	104	105	106	107	108	109	10A	10B	10C	10D	10E	10F

Sign- and Zero-extending

There are two main representation schemes used: unsigned numbers, and 2's complement

For unsigned numbers: 8-bit 0xEF = 239. 239 -> 32 bits is 0x000000EF. General rule: Fill the top bits with 0s (Zero-extension).

For signed numbers: 8-bit 0xEF = -17. -17 -> 32 bits is 0xFFFF FFEF.

General rule: Fill the top bits with the *most significant bit* of the number (Sign-extension).

Loading and Storing Bytes

In addition to word data transfers (lw, sw), RISC-V has byte data transfers:

- `lb rd imm(rs1)  -> sign extend the byte`
- `lbu rd imm(rs1) -> zero extend the byte`
- `sb rs2 imm(rs1) -> store least significant byte only`

Load and store one byte instead of a full word

Byte (0x)	EF	BE	AD	DE	43	53	36	31	43	20	52	49	53	42	56	00
Address (0x)	100	101	102	103	104	105	106	107	108	109	10A	10B	10C	10D	10E	10F

Example: What is in x12?

```
li x11,0x93F5

sw x11,0(x5)

lb x12,1(x5)
```

Example: What is in x12?

Current Line

```
li x11,0x93F5

sw x11,0(x5)

lb x12,1(x5)
```

Register	Value
x11	0x00000000
x12	0x00000000
x5	0x00000100

Random garbage since data isn't set

Byte (0x)	EF	BE	AD	DE	43	53	36	31	43	20	52	49	53	42	56	00
Address (0x)	100	101	102	103	104	105	106	107	108	109	10A	10B	10C	10D	10E	10F

Example: What is in x12?

```
li x11,0x93F5

sw x11,0(x5)

lb x12,1(x5)
```

Current Line

Register	Value
x11	0x000093F5
x12	0x00000000
x5	0x00000100

Byte (0x)	EF	BE	AD	DE	43	53	36	31	43	20	52	49	53	42	56	00
Address (0x)	100	101	102	103	104	105	106	107	108	109	10A	10B	10C	10D	10E	10F

Example: What is in x12?

```
li x11,0x93F5
sw x11,0(x5)
lb x12,1(x5)
```

Current Line

Register	Value
x11	0x000093F5
x12	0x00000000
x5	0x00000100

Byte (0x)	EF	BE	AD	DE	43	53	36	31	43	20	52	49	53	42	56	00
Address (0x)	100	101	102	103	104	105	106	107	108	109	10A	10B	10C	10D	10E	10F

Example: What is in x12?

```
li x11,0x93F5

sw x11,0(x5)

lb x12,1(x5)
```

Current Line

Register	Value
x11	0x000093F5
x12	0x00000000
x5	0x00000100

Little-Endian order

Byte (0x)	F5	93	00	00	43	53	36	31	43	20	52	49	53	42	56	00
Address (0x)	100	101	102	103	104	105	106	107	108	109	10A	10B	10C	10D	10E	10F

Example: What is in x12?

```
li x11,0x93F5

sw x11,0(x5)

lb x12,1(x5)
```

Current Line

Register	Value
x11	0x000093F5
x12	0x00000000
x5	0x00000100

Byte (0x)	F5	93	00	00	43	53	36	31	43	20	52	49	53	42	56	00
Address (0x)	100	101	102	103	104	105	106	107	108	109	10A	10B	10C	10D	10E	10F

Example: What is in x12?

```
li x11,0x93F5

sw x11,0(x5)

lb x12,1(x5)
```

Register	Value
x11	0x000093F5
x12	0xFFFFFF93
x5	0x00000100

Sign-extend: Top bit of 0x93 is 1, so fill with 1s

Byte (0x)	F5	93	00	00	43	53	36	31	43	20	52	49	53	42	56	00
Address (0x)	100	101	102	103	104	105	106	107	108	109	10A	10B	10C	10D	10E	10F

Similar instructions …

- lh – load halfword
- lhu – load halfword unsigned
- There's no lwu in 32-bit implementation
- There's lwu in 64-bit implementation and no ldu!

Offset vs Element Number

- Offset is based on number of bytes!
- If each element of an array is word sized, then to load the element at index 3, offset = 3x4 = 12!!

Exercise …

- Convert following C-language statement into RISC-V assembly
  - A[12] = h + A[8]
  - Assume x21 = h, x22 = base address of A
  - Use x9 as the temporary register!
  - Assume that array elements are word sized!
- Solution

```
lw x9, 32(x22)          # Temporary reg x9 gets A[8]
add x9, x21, x9         # Temporary reg x9 gets h + A[8]
sw x9, 48(x22)          # Stores h + A[8] back into A[12]
```

Another Exercise …

- Compile following C-language statement into RISC-V assembly
  - A[30] = h + A[30] + 1;
  - Assume x21 = h, x10 = base address of A
  - Use x9 as the temporary register!
  - Assume that array elements are word sized!
- Solution

```
lw x9, 120(x10)          # Temporary reg x9 gets A[30]
add x9, x21, x9          # Temporary reg x9 gets h+A[30]
addi x9, x9, 1                  # Temporary reg x9 gets h+A[30]+1
sw x9, 120(x10)          # Stores h+A[30]+1 back into A[30]
```

Memory Alignment

- Data is accessed in terms of multiples of bytes from physical memory chip!
  - Example
    - Word-aligned memory may provide data at (byte) address 0, 4, 8 and so on!
    - What if instruction is lw x10, 3(x0)?
      - That's alignment problem!
    - Solution
      1. Do not allow (in implementation) unaligned memory access!
         - Compiler will convert unaligned access into multiple aligned accesses!
      2. Allow (in implementation) unaligned memory access!
         - Memory hardware needs to deal with it internally (taking more access time)!

Load Instruction – Machine Language

- I-type Encoding Format
  - Immediate field is the offset address!

Type	Field						Operations
(field size)	7 bits	5 bits	5 bits	3 bits	5 bits	7 bits	Arithmetic, Logical, Shift and Loads!
I-type	immediate[11:0]		rs1	func3	rd	opcode	

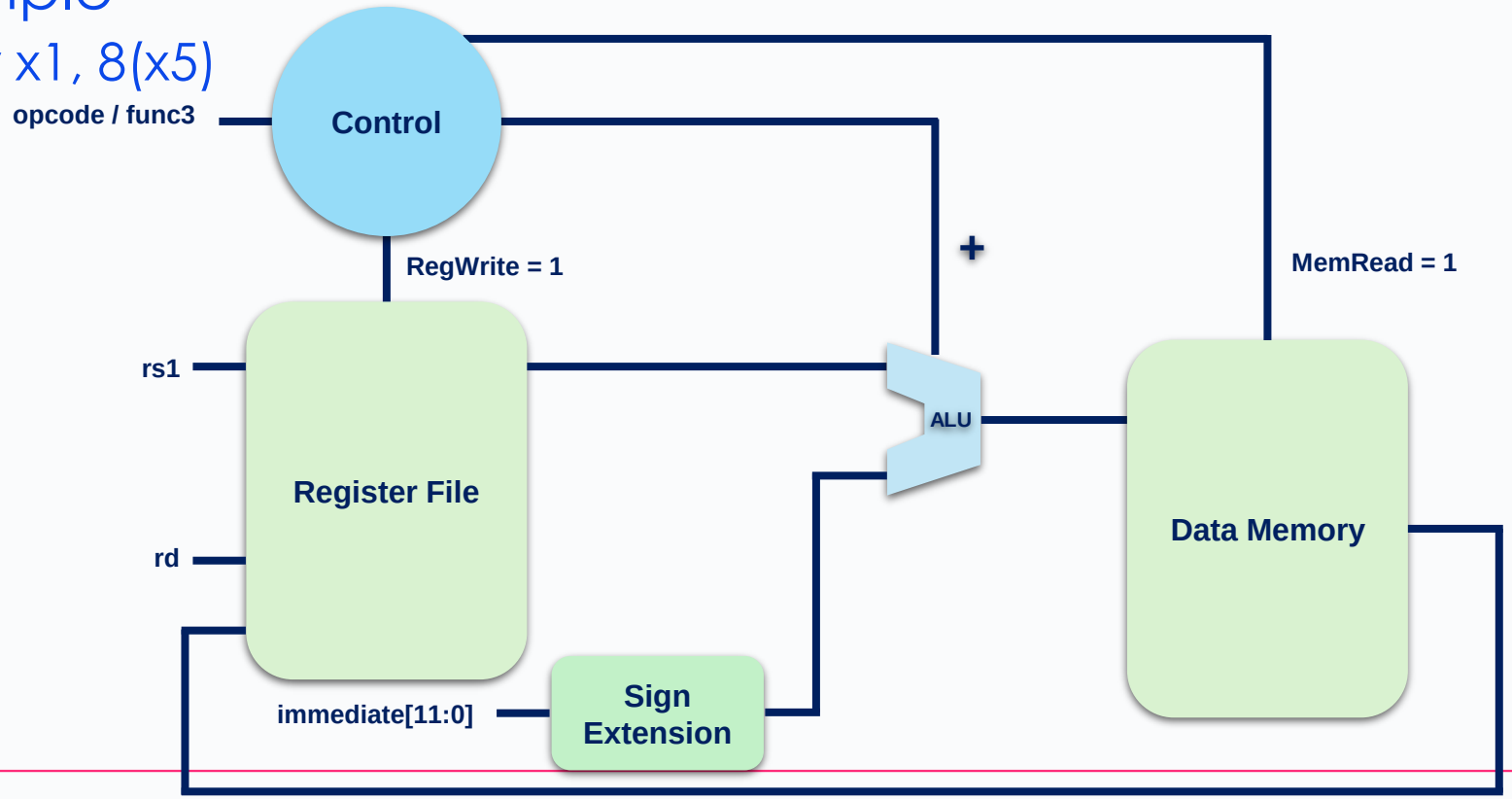
  - Example
    - lw x9, 240(x10)

31 immediate[11:0]	rs1	funct3	rd	opcode 0
12 bits	5 bits	3 bits	5 bits	7 bits
240	10	**2**	9	**3**
000011110000	01010	010	01001	0000011
0000 1111 0000 0101 0010 0100 1000 0011				
0F052483				

Load Instruction – Microarchitecture

I-type	immediate[11:0]	rs1	func3	rd	opcode

- How a load instruction can be implemented?
  - Example
    - lw x1, 8(x5)



Store Instruction – Machine Language

- Should it be I-type Encoding Format just like load?
  - No, think why?
    - Unlike load, store has no destination!
    - Store has two source operands instead of one!
      - One for address just like load!
      - Another containing data to be stored to memory!
  - It uses a unique type (only for stores)!

Store Instruction – Machine Language

- S-type Encoding Format

Type	Field						Operations
(field size)	7 bits	5 bits	5 bits	3 bits	5 bits	7 bits	Stores
S-type	immed[11:5]	rs2	rs1	func3	immed[4:0]	opcode	

- Why is immediate divided into two parts?
  - To keep rs2 and rs1 position fixed in different instructions!
    - Simplifies implementation!
      - Registers can be read in parallel with the instruction decoding!

Store Instruction – Machine Language

- S-type Encoding Format

Type	Field						Operations
(field size)	7 bits	5 bits	5 bits	3 bits	5 bits	7 bits	Stores
S-type	immed[11:5]	rs2	rs1	func3	immed[4:0]	opcode	

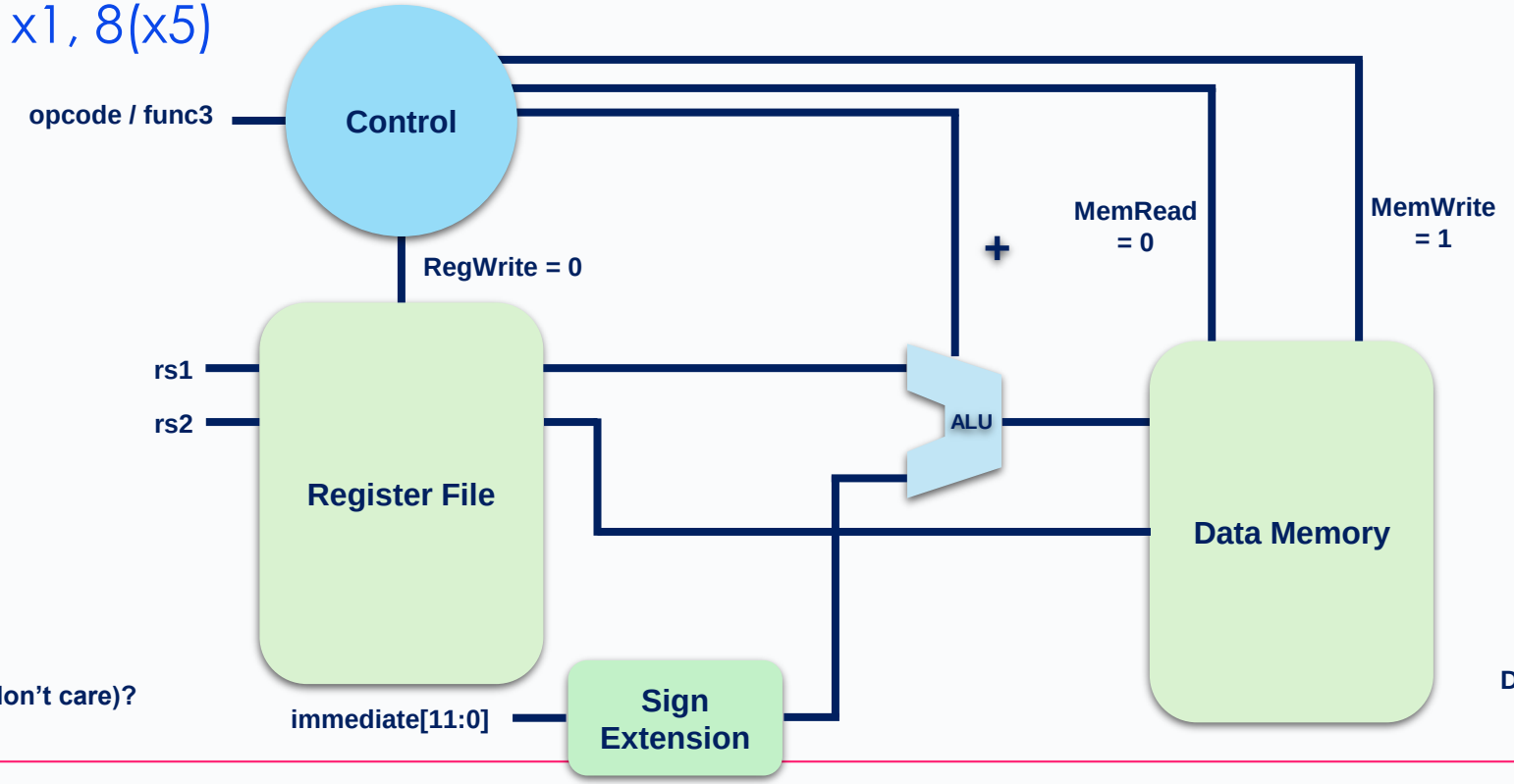
- Example
  - sw x9, 240(x3)
    - Which is rs1 and which is rs2?
      - rs1 is same as in load (base address register)!
    - 240 = 000011110000 = [7 16] (in two parts)

31 immediate[11:5]	rs2	rs1	funct3	immediate[4:0]	opcode 0
7 bits	5 bits	5 bits	3 bits	5 bits	7 bits
7	9	3	**2**	**16**	**35**
0000111	01001	00011	010	10000	0100011
0000 1110 1001 0001 1010 1000 0010 0011					
0E91A823					

Store Instruction – Microarchitecture

- How a store instruction can be implemented?
  - Example
    - sw x1, 8(x5)

S-type	immed[11:5]	rs2	rs1	func3	immed[4:0]	opcode



Can we have RegWrite = x (don't care)?

Do we need MemRead?

Agenda

- Main Memory
- Data Transfer Instructions
- **C Code Example**
- Upper Immediate Instructions

Converting C code to RISC-V

So far, we've assumed that each variable gets stored in one register. What if we have more than 32 variables? Let's translate a program under the following restrictions:

- Only registers x5, x6, and x7 may be modified, and only for intermediate calculations
    - We'll name them "`t0`", "`t1`", and "`t2`", for "temporary register 0-2“
- x2 points to the start of a block of memory that we can use however we want
    - We'll name x2 "`sp`", for "stack pointer"

Converting C code to RISC-V

```
int a = 5;

char b[] = "string"; // Array will get stored on stack

int c[10];

uint8_t d = b[3];

c[4] = a+d;

c[a] = 20;
```

Converting C code to RISC-V

```
int a = 5;
char b[] = "string";
int c[10];
uint8_t d = b[3];
c[4] = a+d;
c[a] = 20;
```

Step 1: Assign each variable to some offset from sp.

- Exact values don't matter as long as we're consistent

a: 0(sp)

b: 4(sp)

c: 12(sp)

d: 52(sp)

Converting C code to RISC-V

```
int a = 5;
char b[] =
"string";
int c[10];
uint8_t d = b[3];
c[4] = a+d;
c[a] = 20;
```

a: 0(sp)

b: 4(sp)

c: 12(sp)

d: 52(sp)

```
li t0 5

sw t0 0(sp)
```

Converting C code to RISC-V

```
int a = 5;
char b[] =
"string";
int c[10];
uint8_t d = b[3];
c[4] = a+d;
c[a] = 20;
```

a: 0(sp)

b: 4(sp)

c: 12(sp)

d: 52(sp)

```
li t0 0x73
sb t0 4(sp)
li t0 0x74
sb t0 5(sp)
li t0 0x72
sb t0 6(sp)
li t0 0x69
sb t0 7(sp)
li t0 0x6E
sb t0 8(sp)
li t0 0x67
sb t0 9(sp)
sb x0 10(sp)
```

Converting C code to RISC-V (Better Approach)

```
int a = 5;
char b[] =
"string";
int c[10];
uint8_t d = b[3];
c[4] = a+d;
c[a] = 20;
```

a: 0(sp)

b: 4(sp)

c: 12(sp)

d: 52(sp)

```
li t0
0x69727473
sw t0 4(sp)
li t0
0x0000676E
sw t0 8(sp)
```

Converting C code to RISC-V

```
int a = 5;
char b[] =
"string";
int c[10];
uint8_t d = b[3];
c[4] = a+d;
c[a] = 20;
```

a: 0(sp)

b: 4(sp)

c: 12(sp)

d: 52(sp)

Nothing

Converting C code to RISC-V

```
int a = 5;
char b[] =
"string";
int c[10];
uint8_t d = b[3];
c[4] = a+d;
c[a] = 20;
```

a: 0(sp)

b: 4(sp)

c: 12(sp)

d: 52(sp)

```
lb t0 7(sp)
sb t0
52(sp)
```

Converting C code to RISC-V

```
int a = 5;
char b[] =
"string";
int c[10];
uint8_t d = b[3];
c[4] = a+d;
c[a] = 20;
```

a: 0(sp)

b: 4(sp)

c: 12(sp)

d: 52(sp)

```
lw t0 0(sp)
lbu t1
52(sp)
add t2 t0
t1
sw t2
28(sp)
```

Converting C code to RISC-V

```
int a = 5;
char b[] =
"string";
int c[10];
uint8_t d = b[3];
c[4] = a+d;
c[a] = 20;
```

a: 0(sp)

b: 4(sp)

c: 12(sp)

d: 52(sp)

```
li t0 20
lw t1 0(sp)
sw t0 t1*4+12(sp)
slli t1 t1 2 #t1*=4
addi t1 t1 12
add t1 t1 sp
sw t0 0(t1)
```

Converting C code to RISC-V

```
int a = 5;
char b[] =
"string";
int c[10];
uint8_t d = b[3];
c[4] = a+d;
c[a] = 20;
```

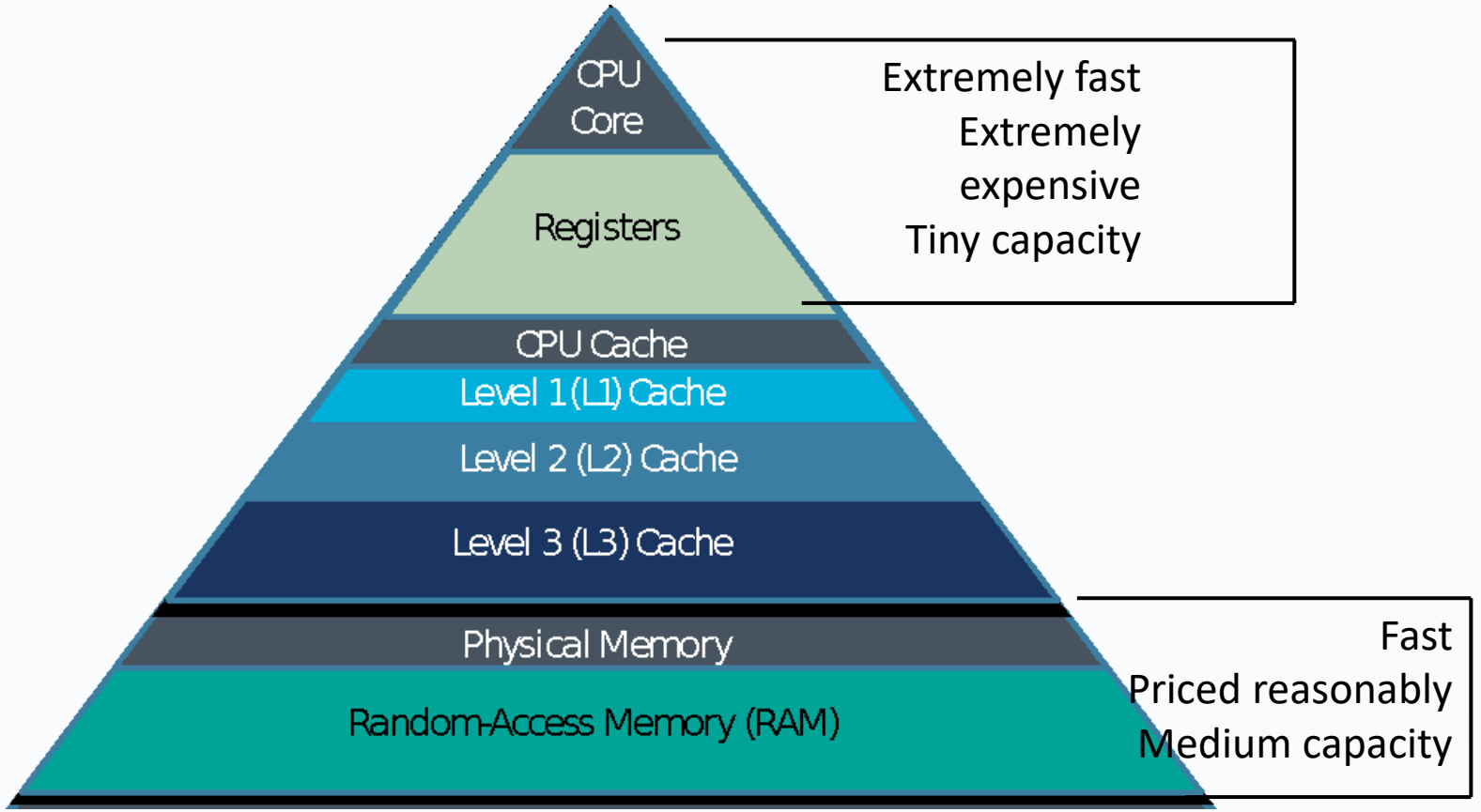
```
li t0 5
sw t0 0(sp)
li t0 0x69727473
sw t0 4(sp)
li t0 0x0000676E
sw t0 8(sp)
lb t0 7(sp)
sb t0 52(sp)
lw t0 0(sp)
lbu t1 52(sp)
add t2 t0 t1
sw t2 28(sp)
li t0 20
lw t1 0(sp)
slli t1 t1 2 #t1*=4
addi t1 t1 12
add t1 t1 sp
sw t0 0(t1)
```

Why we need so many registers

- As the previous example showed, it's possible to write RISC-V with only a sp and three temporary registers

- Why do we have 32 registers?

RISC-V Guiding Philosophy



Speed of Registers vs Memory

- Given that
  - Registers: 32 words (128 Bytes)
  - Memory (DRAM): Billions of bytes
    (2 GB to 96 GB on laptop)

- and physics dictates…
  - Smaller is faster

- How much faster are registers than DRAM??
  - About 50-500 times faster!
    (in terms of latency of one access - tens of ns)
    - But subsequent words come every few ns

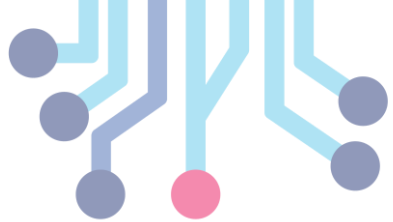
And in Conclusion…

- Memory is byte-addressable, but **`lw`** and **`sw`** access one word at a time.
- A pointer (used by **`lw`** and **`sw`**) is just a memory address, we can add to it or subtract from it (using offset).
- Memory can be used for variables we can't store in registers, but 100x slower than using registers directly
  - Use loads and stores as infrequently as possible!
- New Instructions:

`lw, sw, lb, sb, lbu, lh, sh, lhu`

Agenda

- Main Memory
- Data Transfer Instructions
- C Code Example
- **Upper Immediate Instructions**



Immediate in most ALU instructions is 12-bit, how can we have larger immediate value e.g., 32-bit?

Load Upper Immediate Instruction

- lui instruction allows a mechanism to make larger immediate …
- Example
  - Example
    - Example
      - Load following into 32-bit register, x19!
        - 00000000 00111101 0000**0101 00000000**
      - Solution
        - 00000000 00111101 0000 = 976 in decimal
        - 0101 00000000 = 1280 in decimal
        - lui x19, 976 # load immediate in bits {31-12} of register and make other bits 0!
        - addi x19, x19, 1280

LUI: Corner case

- How would you translate "li t0 0xABCDEFFF" to instructions?
- Initial idea:
  - ```
    lui t0 0xABCDE
    addi t0 t0 0xFFF
    ```
- Problem: 0xFFF isn't 4095; it's -1
  - After the first line, we get t0 = 0xABCDE000
  - After the second line, we get t0 = 0xABCDDFFF instead!
- As such, we need to be careful in this case and do lui t0 0xABCDF instead
  - ```
    lui t0 0xABCDF          #t0 stores 0xABCDF000
    addi t0 t0 0xFFF        #t0 stores 0xABCDEFFF
    ```
- This ends up affecting li instructions only when the offset has its 11th bit set to 1, so it's an easy case to forget about.

AUIPC and Relative Addressing

- auipc similarly gets used primarily as a way to save an arbitrary value when used with an addi
- The main difference is that it adds its result to PC
- Often when writing code, we want to allow multiple programs to be combined (like with libraries), but that would change the addresses of all the labels in our code
- To avoid this issue, many instructions involving labels use relative addressing instead of absolute addressing.
  - Absolute address: "This label is at location 0x000000FC". This fails if we move our code to a different place in memory
  - Relative address: "This label is 48 bytes after the current line of code". This still works if we move both the line of code and the label the same distance.
- As such, auipc often gets used with la instructions.

Add Upper Immediate to PC Instruction

NCDC
NUST CHIP DESIGN CENTRE

- auipc is similar to lui and is helpful in making addresses in PC-relative addressing
  - auipc forms a 32-bit offset from the 20-bit immediate, filling in the lowest 12 bits with zeros, adds this offset to the pc, then places the result in register rd
    - Can be used for both control-flow transfers or data accesses!
    - Used in combination with addi, jalr or load / store …
  - See RISC-V ISA Manual for details …

LUI and AUIPC Instructions – Machine Language

- U-Type Encoding Format

Type	Field						Operations
(field size)	7 bits	5 bits	5 bits	3 bits	5 bits	7 bits	lui and auipc instructions
U-type	immediate[31:12]				rd	opcode	

  - Note
    - immediate[31:12**] doesn't mean that** upper 20 bits of immediate value (as specified in instruction) are used
      - **it means** 20-bit immediate is placed in upper 20 bits (31:12) of register rd!

  - Example
    - lui x19, 976
      - immediate[31:12] = 976 = 00000000 00111101 0000
      - x19 = 00000000 00111101 0000**0000 00000000**

LUI and AUIPC Instructions – Microarchitecture

- How to implement lui and AUIPC?
  - LUI
    - Take 20-bit immediate from instruction, append 12-bit 0's towards right and write to rd!
  - AUIPC
    - Take 20-bit immediate from instruction, append 12-bit 0's towards right, add it to PC and write to rd!
  - Draw the two as an exercise …

Instructions covered so far …

- Arithmetic
  - add, sub, addi
- Logical
  - and, or, xor, andi, ori, xori
- Shift
  - sll, srl, sra, slli, srli, srai
- Data Transfer
  - lw, sw, lh, lhu, sh, lb, lbu, sb
- Upper Immediate
  - lui, auipc

Tasks

- Use only the instructions learned so far to complete following tasks!
- Task 1
  - Convert an array of (5) byte-sized integers, in memory, to word-sized integers while retaining the value!
- Task 2
  - Convert a string of (5) small English letters, defined in memory, to a string of capital English letters!
- Task 3
  - Reverse a string of (5) characters, defined in memory!

Thank You

NCDC
NUST
CHIP
DESIGN
CENTRE