Computer Architecture

IN BA3 - Paolo IENNE

Notes by Ali EL AZDI

September 11, 2024

Introduction

This document is designed to offer a LaTeX-styled overview of the Computer Architecture course, emphasizing brevity and clarity. Should there be any inaccuracies or areas for improvement, please reach out at ali.elazdi@epfl.ch for corrections. For the latest version, check my GitHub repository.
`https://github.com/elazdi-al/comparch/blob/main/main.pdf`

Contents

Chapter 1

Part I(a) - ISA Reminder, Assembly Language, Compiler - W 1.1

hum...welcome back
In the first part of the course, professor introduced (for motivational purposes) how computer architecture, specifically processors, have become essential to our lives, and how the field is growing exponentially. (didn't think it was essential to mention here...)

1.1 From High Level Languages to Assembly Language

1.1.1 High Level Languages

When talking about programming we usually think of programs that look like this...

```
int data = 0x00123456;
int result = 0;
int mask = 1;
int count = 0;
int temp = 0;
int limit = 32;
do {
    temp = data & mask;
    result = result + temp;
    data = data >> 1;
    count = count + 1;
} while (count != limit);
```

name	value
data	0x00123456
result	0
mask	1
count	...
temp	
limit	
...	
my_float	3.141529
a_string	Hello world!

1.1.2 Assembly Language

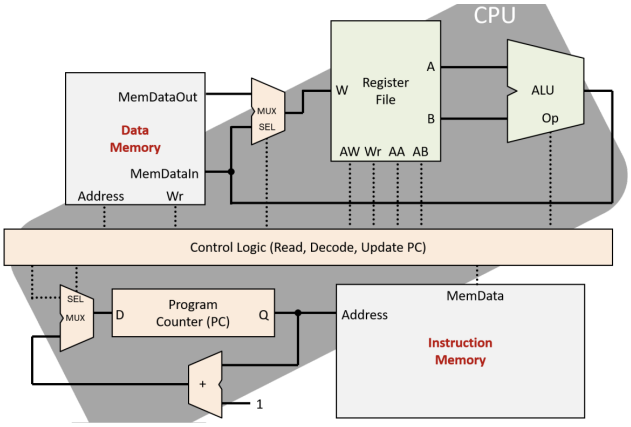
We use this code because it enables us to build a *Finite State Machine*, which isn't feasible with C code. This language provides a more rigid format with a sequence of numbered instructions, an *opcode*, predefined variable names, and the ability to **jump between lines**.

```
    li x1, 0x00123456
    li x2, 0
    li x3, 1
    li x4, 0
    li x5, 0
    li x6, 32
loop: and x5, x1, x3
    add x2, x2, x5
    srli x1, x1, 1
    addi x4, x4, 1
    bne x4, x6, loop
```

1.2 Processors

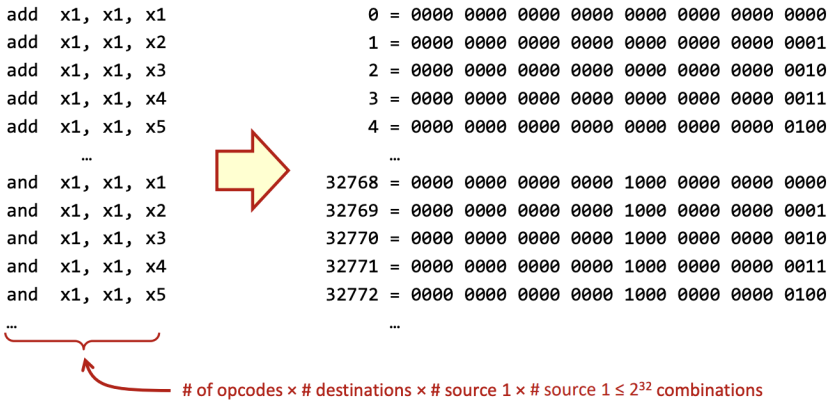
Remember, a processor can be decomposed into five components:

- **ALU (Arithmetic and Logic Unit)**: Performs arithmetic and logical operations.
- **Register File**: Stores data temporarily for quick access during processing.
- **Memory**: Holds data and instructions needed by the processor.
- **Control Logic**: Directs the operation of the processor by coordinating the other components.
- **PC (Program Counter)**: Keeps track of the address of the next instruction to be executed.
- **Instruction Memory**: Stores the program instructions that the processor will execute.

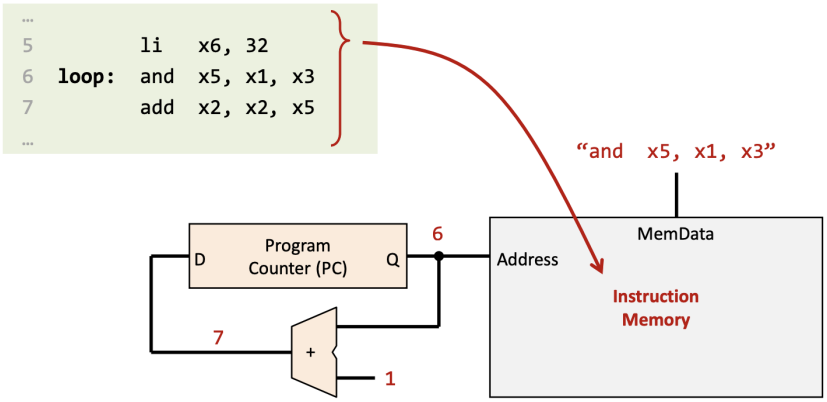


We may distinguish three types of general operations made by the processor:

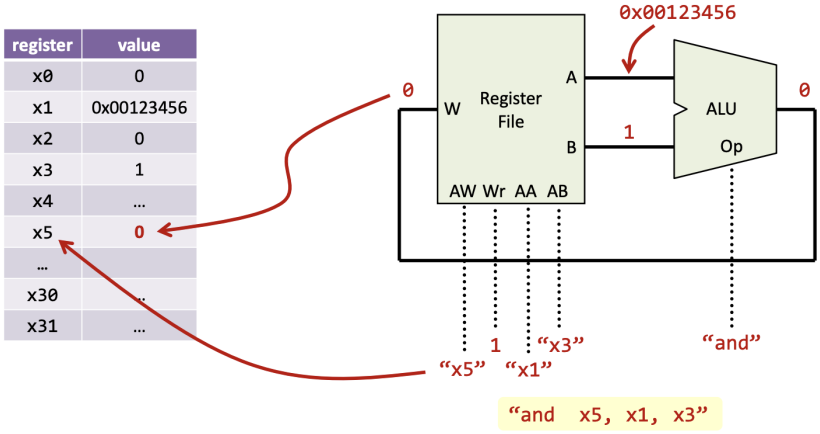
Encoding



Fetching



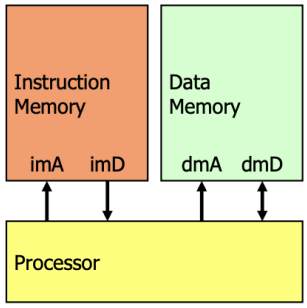
Executing



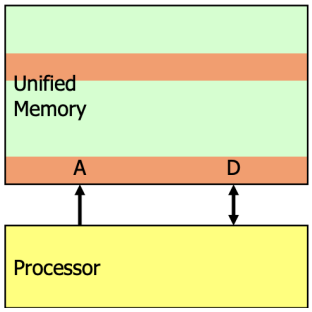
1.3 Joint or Disjoint Program and Data Memories

There are two main types of architectures one called the Harvard Architecture (Where the data and the memory are seperate) and pne called Unified Architecture (where data is shared with the program memory)

Harvard Architecture



Unified Architecture

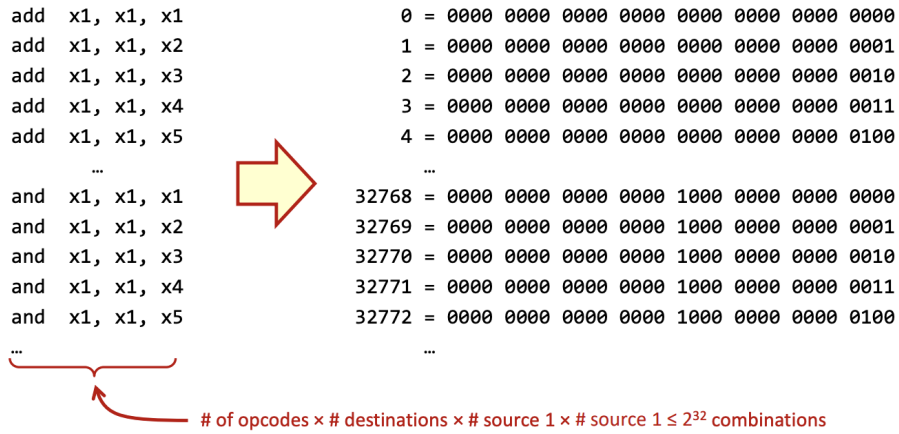


1.4 The Encoding problem

We may ask ourselves how we encode assembly written instructions into actual 0s and 1s.

1.4.1 The Stupid Solution

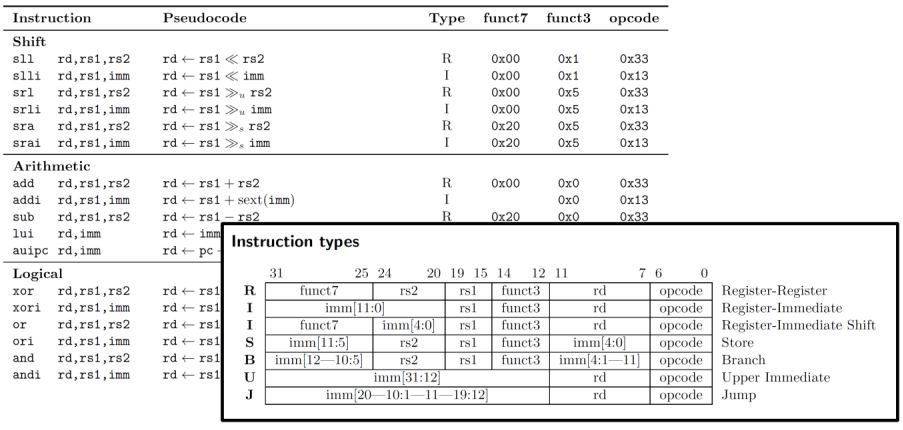
Now, the professor throws out the "stupid idea"(his words) of just counting all possible instructions, assigning a number to each one, and writing the numbers in binary. The problem with such a method is that the number of instructions could grow exponentially, requiring an unmanageable number of bits to represent each one, leading to inefficiency.



"stupid solution"

1.4.2 RISC-V Encoding (The Solution)

Instead, the chosen solution is to use an instruction set encoding where instructions are grouped into classes, each with a fixed format optimizing both memory usage and processing speed by limiting the number of bits required to represent instructions.



Instruction		Pseudocode	Type	funct7	funct3	opcode
Shift						
sll	rd,rs1,rs2	rd ← rs1 ≪ rs2	R	0x00	0x1	0x33
slli	rd,rs1,imm	rd ← rs1 ≪ imm	I	0x00	0x1	0x13
srl	rd,rs1,rs2	rd ← rs1 $\gg_u$ rs2	R	0x00	0x5	0x33
srli	rd,rs1,imm	rd ← rs1 $\gg_u$ imm	I	0x00	0x5	0x13
sra	rd,rs1,rs2	rd ← rs1 $\gg_s$ rs2	R	0x20	0x5	0x33
srai	rd,rs1,imm	rd ← rs1 $\gg_s$ imm	I	0x20	0x5	0x13
Arithmetic						
add	rd,rs1,rs2	rd ← rs1 + rs2	R	0x00	0x0	0x33
addi	rd,rs1,imm	rd ← rs1 + sext(imm)	I		0x0	0x13
sub	rd,rs1,rs2	rd ← rs1 − rs2	R	0x20	0x0	0x33
lui	rd,imm	rd ← imm				
auipc	rd,imm	rd ← pc +				
Logical						
xor	rd,rs1,rs2	rd ← rs1				
xori	rd,rs1,imm	rd ← rs1				
or	rd,rs1,rs2	rd ← rs1				
ori	rd,rs1,imm	rd ← rs1				
and	rd,rs1,rs2	rd ← rs1				
andi	rd,rs1,imm	rd ← rs1				

Instruction types

	31–25	24–20	19–15	14–12	11–7	6–0	
R	funct7	rs2	rs1	funct3	rd	opcode	Register-Register
I	imm[11:0]		rs1	funct3	rd	opcode	Register-Immediate
I	funct7	imm[4:0]	rs1	funct3	rd	opcode	Register-Immediate Shift
S	imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode	Store
B	imm[12—10:5]	rs2	rs1	funct3	imm[4:1—11]	opcode	Branch
U	imm[31:12]				rd	opcode	Upper Immediate
J	imm[20—10:1—11—19:12]				rd	opcode	Jump

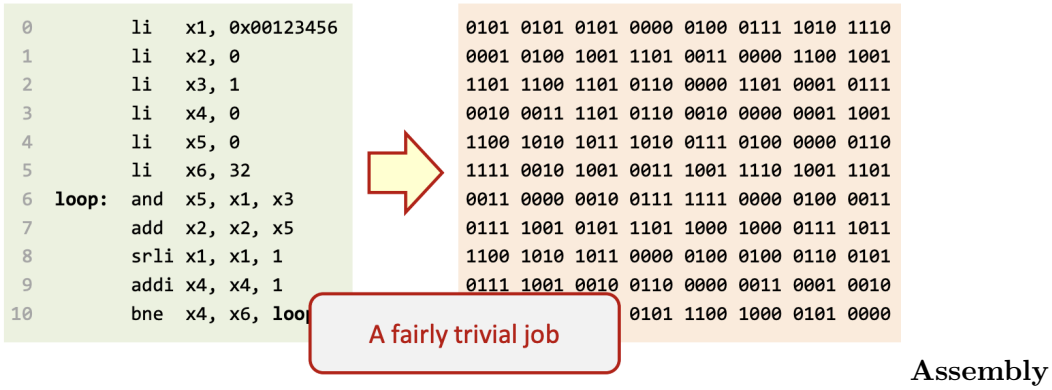
RISC-V encoding

1.4.3 Automating this process

Now to automate the processes of decoding assembler code into machine code we use an ***Assembler****, and to automate the process of decoding a higher level language to assembler we use a* ***Compiler****.*

Assembler

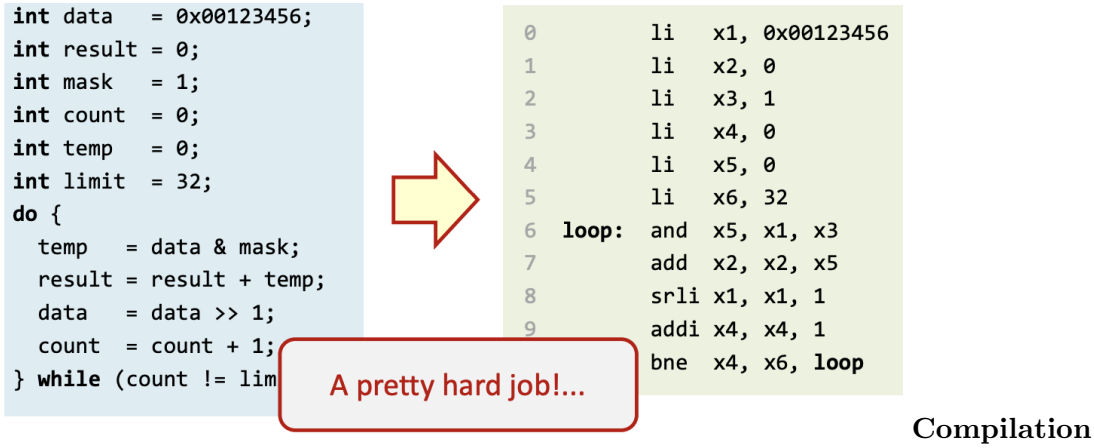
The program that does this is called an assembler. It takes the assembly code and converts it into machine code.



Assembly

Compiler

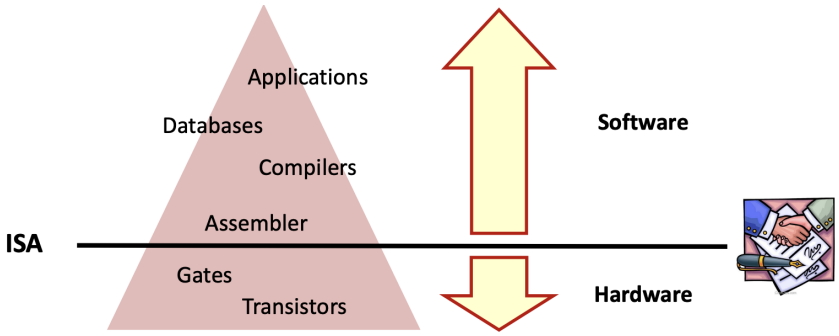
A compiler is a program that translates high-level source code written in languages like C or Java into machine code or an intermediate representation.



Compilation

1.5 ISA (Instruction Set Architecture)

The ISA is the interface between the hardware and the software. It defines the instructions that a processor can execute, as well as the format of those instructions.



Chapter 2

Part I(b) - ISA, Functions, and Stack - W 1.2

2.1 Arithmetic and Logic Instructions in RISCV

Bellow some examples of RISCV instructions:
Two Operands Instructions

```
1 sll  x5, x5, x9
2 add  x6, x5, x7
3 xor  x6, x6, x8
4 slt  x8, x6, x7
```

Shift x5 left by x9 positions → x5
Add x5 and x7 → x6
Logic XOR bitwise x6 and x8 → x6
Set x8 to 1 if x6 is lower than x7, otherwise to 0

Arithmetic Instructions

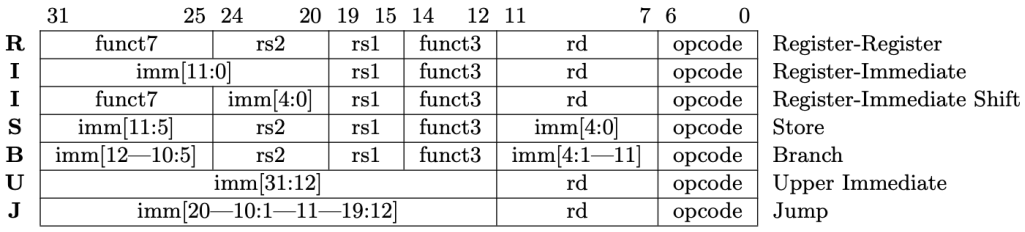
```
1 slli x5, x5, 3
2 addi x6, x5, 72
3 xori x6, x6, -1
4 slti x8, x6, 321
```

Shift x5 left of 3 positions → x5
Add 72 to x5 → x6
Logic XOR bitwise x6 and 0xFFFFFFFF → x6
Set x8 to 1 if x6 is lower than 321, to 0 otherwise

Here, you may ask yourself, why are all immediates (constants) writtent on a maximum of 12bits?

2.1.1 Constants must be encoded on 12 bits

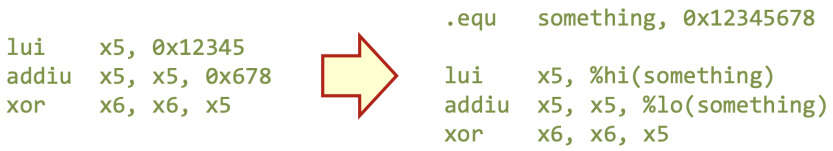
As you may see here, all instructions encode immediates on 12 bits.



	31–25	24–20	19–15	14–12	11–7	6–0	
R	funct7	rs2	rs1	funct3	rd	opcode	Register-Register
I	imm[11:0]		rs1	funct3	rd	opcode	Register-Immediate
I	funct7	imm[4:0]	rs1	funct3	rd	opcode	Register-Immediate Shift
S	imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode	Store
B	imm[12—10:5]	rs2	rs1	funct3	imm[4:1—11]	opcode	Branch
U	imm[31:12]				rd	opcode	Upper Immediate
J	imm[20—10:1—11—19:12]				rd	opcode	Jump

2.1.2 Assembler Directives

Assembler directives help write cleaner and more readable code. The code snippets on the left and right below are equivalent.



The left-hand side code snippet shows an assembly sequence where a 32-bit constant value (`0x12345678`) is loaded into a register (`x5`). Since immediate values are 16-bit limited, this requires splitting the 32-bit value into two instructions:

- The first instruction, `lui`, loads the upper 16 bits (`0x12345`) into the register `x5`.
- The second instruction, `addiu`, adds the lower 16 bits (`0x678`) to `x5`, completing the full 32-bit value in the register.

This approach, while functional, can become cumbersome when dealing with multiple constants, making the code less readable and harder to maintain.

The right-hand side shows the same functionality but makes use of assembler directives, specifically the `.equ` directive to define a label (`something`) for the constant `0x12345678`. Using the `%hi()` and `%lo()` pseudo-instructions, the assembler automatically splits the constant into its upper and lower parts:

- The `%hi(something)` loads the upper 16 bits into `x5`.
- The `%lo(something)` adds the lower 16 bits to `x5`.

This method enhances code clarity and maintainability, especially when working with multiple constants, by using human-readable labels instead of raw numeric values. The assembler handles the details of splitting the 32-bit constant into its upper and lower parts.

Directive	Effect
`.text`	Store subsequent instructions at next available address in *text* segment
`.data`	Store subsequent items at next available address in *data* segment
`.asciiz`	Store string followed by null-terminator in `.data` segment
`.byte`	Store listed values as 8-bit bytes
`.word`	Store listed values as 32-bit words
`.equ`	Define constants

2.1.3 The `x0` Register

The `x0` register is hardwired to 0 and cannot be changed. Any attempt to write into `x0` will have no effect.

`Why is this useful?`

One common application is in introducing wait delays during program execution. By leveraging the fixed nature of `x0`, it simplifies certain instructions that require an immediate zero value.

2.2 PseudoInstructions

PseudoInstructions simplify commands involving the `x0` register by creating easier-to-use alternatives.

Pseudoinstruction	Base Instruction(s)	Meaning
`nop`	`addi x0, x0, 0`	No operation
`li rd, immediate`	Myriad sequences	Load immediate
`mv rd, rs`	Myriad sequences	Copy register
`not rd, rs`	`xori rd, rs, -1`	One's complement
`neg rd, rs`	`sub rd, x0, rs`	Two's complement
`seqz rd, rs`	`sltiu rd, rs, 1`	Set if = zero
`snez rd, rs`	`sltu rd, x0, rs`	Set if ≠ zero
`sltz rd, rs`	`slt rd, rs, x0`	Set if ¡ zero
`sgtz rd, rs`	`slt rd, x0, rs`	Set if ¿ zero

The term *myriad sequences* refers to a series of instructions that together achieve the functionality of a single pseudoinstruction, such as using `lui` and `addi` to implement `li rd, immediate`. **According to the professor li should be called `mvi` (as move immediate).**

2.2.1 Control flow instructions

Control flow instructions are used to change the order of execution of instructions are a kind of pseudo-instructions.

```
    li x1, 0x00123456
    li x2, 0
    li x3, 1
    li x4, 0
    li x5, 0
    li x6, 32
loop: and x5, x1, x3
    add x2, x2, x5
    srli x1, x1, 1
    addi x4, x4, 1
    bne x4, x6, loop
```

2.2.2 If-Then-Else

```
if (x5 == 72) {
    x6 = x6 + 1;
    } else {
    x6 = x6 - 1;
}
```

```
.text
    li x7, 72
    beq x5, x7, then_clause
else_clause:
    addi x6, x6, -1
    j end_if
then_clause:
    addi x6, x6, 1
end_if:
```

As seen here, beqi does not exist in RISCV, instead we use `beq` and `li` to achieve the same result.

2.2.3 Jumps and Branches

A common but not universal distinction exists between *jumps* and *branches*. In RISC-V (inherited from MIPS and used by SPARC, Alpha, etc.), jumps refer to unconditional control transfer instructions, while branches refer to conditional control transfer instructions. However, not all architectures follow this convention. For instance, in x86, all control transfer instructions are considered jumps, such as `JMP`, `JZ`, `JC`, and `JNO`.

2.2.4 Comparaisions

The processor implements only $<$ and $>$, and the assembler "creates" $\leq$ and $\geq$.

Pseudoinstruction	Base Instruction(s)	Meaning
`beqz rs, offset`	`beq rs, x0, offset`	Branch if = zero
`bnez rs, offset`	`bne rs, x0, offset`	Branch if $\neq$ zero
`blez rs, offset`	`bge x0, rs, offset`	Branch if $\leq$ zero
`bgez rs, offset`	`bge rs, x0, offset`	Branch if $\geq$ zero
`bltz rs, offset`	`blt rs, x0, offset`	Branch if $<$ zero
`bgtz rs, offset`	`blt x0, rs, offset`	Branch if $>$ zero
`bgt rs, rt, offset`	`blt rt, rs, offset`	Branch if $>$
`ble rs, rt, offset`	`bge rt, rs, offset`	Branch if $\leq$
`bgtu rs, rt, offset`	`bltu rt, rs, offset`	Branch if $>$, unsigned
`bleu rs, rt, offset`	`bgeu rt, rs, offset`	Branch if $\leq$, unsigned

2.2.5 Do-While

Do-while loops look like this (we obviously use control flow instructions here).

```
do {
    x5 = x5 >> 1;
    x6 = x6 + 1;
} while (x5 != 0);
```

```
.text
loop:
    srli x5, x5, 1
    addi x6, x6, 1
    bnez x5, loop
```

2.3 Functions

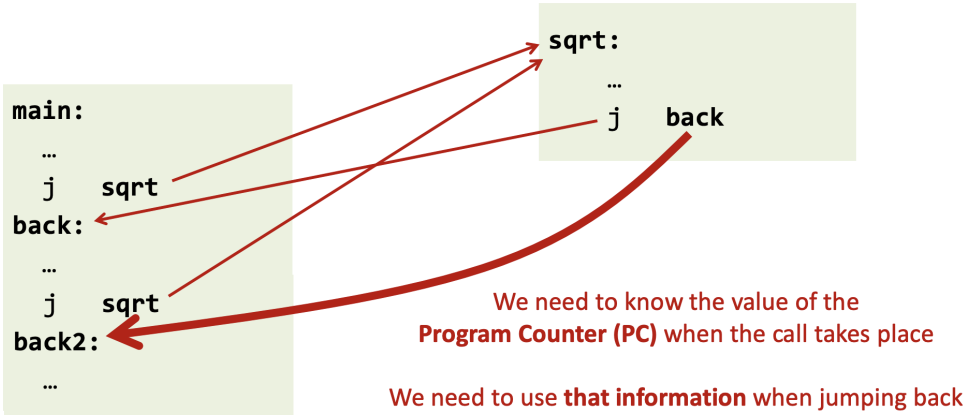
In higher-level programming languages, functions (routines, subroutines, procedures, methods, etc.) are used to encapsulate code and make it reusable.
Calling a function involves these steps:

1. Place arguments where the called function can access them.
2. Jump to the function.
3. Acquire storage resources the function needs.
4. Perform the desired task of the function.
5. Communicate the result value back to the calling program.
6. Release any local storage resources.
7. Return control to the calling program.

2.3.1 Jump to the Function/Retun control to the calling program

The too simple not working approach

A simple (not working) approach for creating functions would be to do this:



With this approach the function doesn't know where to return to after being called (back2 or back) **For the next part, remember, the Program Counter is distinct from general-purpose registers. It is dedicated to managing the flow of instruction execution, while general registers are used for data manipulation.**

The Good Approach

The right approach involves using the Jump and Link instruction `jal`*, here loading PC + 4 (remember 4 bytes per Instruction) into x1 as a way to come back from the function.*

```
main:
    ...
    jal x1, sqrt
    ...
    ...
    jal x1, sqrt
```

```
sqrt:
    ...
    ...
    jr x1
```

Both times x1 was used to store the return adress, and there is a reason for that (Register Conventions Sections).

2.3.2 Jump Instructions

There are only two core real jump instructions in RISCV, `jal` *(jump and link) and* `jalr` *(jump and link register), the rest are pseudo instructions using them.*

Pseudoinstr.	Base Instruction(s)	Meaning
`j offset`	`jal x0, offset`	Jump
`jal offset`	`jal x1, offset`	Jump and link
`jr rs`	`jalr x0, 0(rs)`	Jump register
`jalr rs`	`jalr x1, 0(rs)`	Jump and link register
`ret`	`jalr x0, 0(x1)`	Return from subroutine

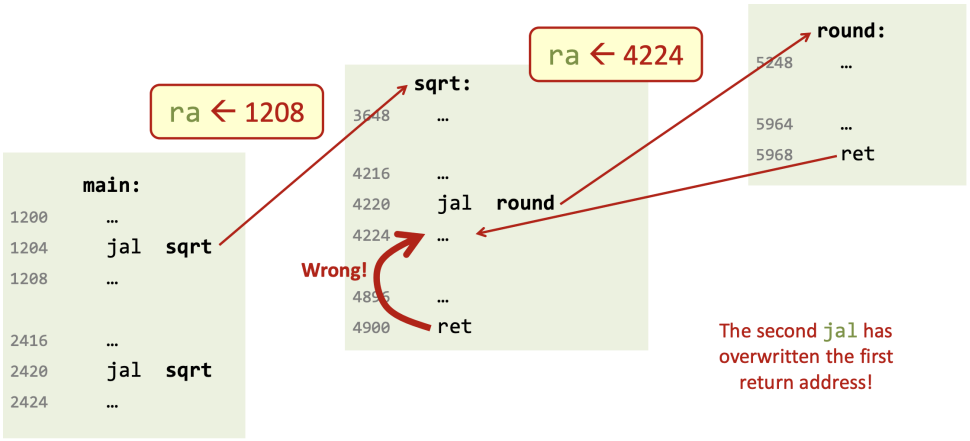
2.3.3 Register Conventions

Register conventions are rules that dictate how registers are used in a program, here are the ones we've seen for now

Register	Mnemonic	Description
x0	zero	Hard-wired zero
x1	ra	Return Address

2.3.4 Back to the good (not so good) approach

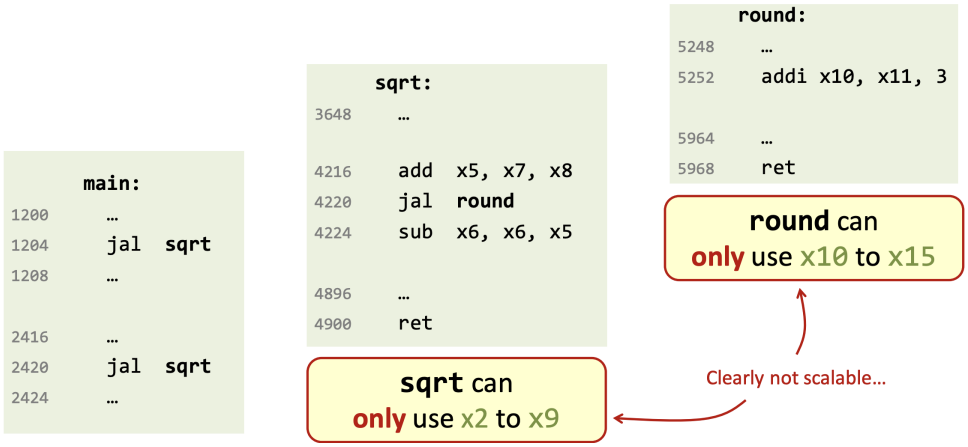
There's still a problem with the previous approach, say for example you want to call a function from another function.



Here the allocated space for the return address is overwritten by the second function call, and the first function can't return to the right place.

2.3.5 One simple solution (still not good)

One solution would be to say that a range of registers are used for certain functions and that they can't be used by other functions.



The problem here is that it's still not very scalable.

2.3.6 Acquire storage resources the function needs (still not it)

One simple solution to our problem would be to allocate memory for the function at in the data section of the program.

```
.data
sqrt_save_ra: .word 0
sqrt_save_x5: .word 0
```

```
.text
sqrt:
...
add x5, x7, x8
sw ra, sqrt_save_ra
sw x5, sqrt_save_x5
jal round
lw ra, sqrt_save_ra
lw x5, sqrt_save_x5
sub x6, x6, x5
...
ret
```

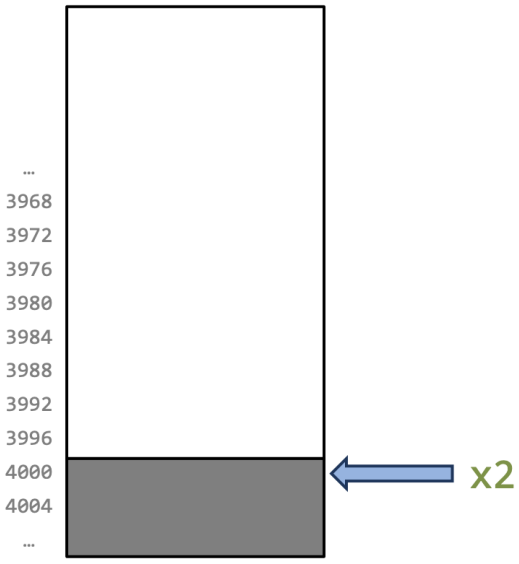
Problem: Recursive Functions

The problem here is that the return address is overwritten by the recursive call.

```
.data
    find_child_save_ra: .word 0
.text
    find_child:
    ...
    sw ra, find_child_save_ra
    jal find_child
    lw ra, find_child_save_ra
    ...
    ret
```

2.3.7 The Stack

The Solution to our Problem is this, the Stack.
The Stack is a region of memory that grows and shrinks as needed.
We may use a register (e.g `x2`) to point to the first used word after the end of the used region.

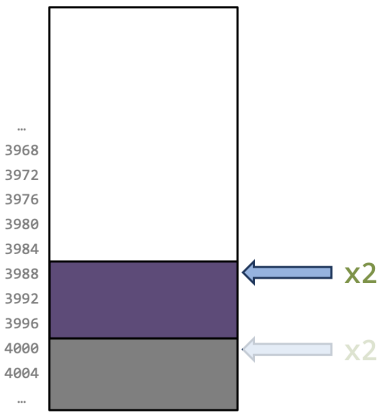


Dynamic Memory Allocation

The Stack, contrary to the Data Section, is dynamic and can be used to allocate memory when needed. This means that during program execution, variables or temporary data can be stored in the stack, which grows or shrinks depending on the operations performed.
The `stack pointer`, typically register x2, is used to manage the allocation and deallocation of memory.

In this instruction, for example, we allocate 12 bytes in the stack. We achieve this by decrementing the stack pointer (x2) by 12. This ensures that the new memory space is available for temporary storage.

```
1 addi x2, x2, -12
```

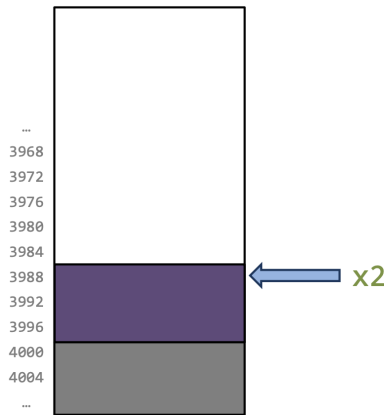


Retrieving Data from the Stack

Once memory has been allocated on the stack, we can store or retrieve data from it. In this case, we are retrieving data that was previously saved in the stack. The lw (load word) instruction is used to load the values stored at different offsets in the stack.

In this case, we retrieve three different values from the stack using the lw instruction, which loads a 4-byte value into the specified registers (ra, x5, and x6). The offsets (0, 4, and 8) refer to different positions in the 12 bytes we allocated earlier.

```
1 lw ra, 0(x2)
2 lw x5, 4(x2)
3 lw x6, 8(x2)
```

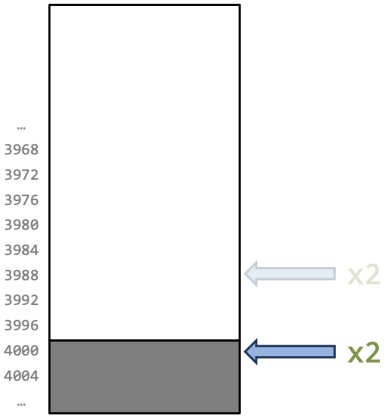


Memory Deallocation

After the data has been used or is no longer needed, it is good practice to deallocate the memory to ensure proper management of the stack. We deallocate memory by adjusting the stack pointer (x2) back to its original position.

In this instruction, we restore the stack to its previous state by adding 12 back to the stack pointer (x2).
This effectively "frees" the 12 bytes of memory we had allocated earlier.

```
addi x2, x2, 12
```



The Stack Pointer

The Stack Pointer is a register that points to the top of the stack, by convention it corresponds to the x2 register

Register	ABI Name	Description	Preserved across call?
x2	sp	Stack pointer	Yes

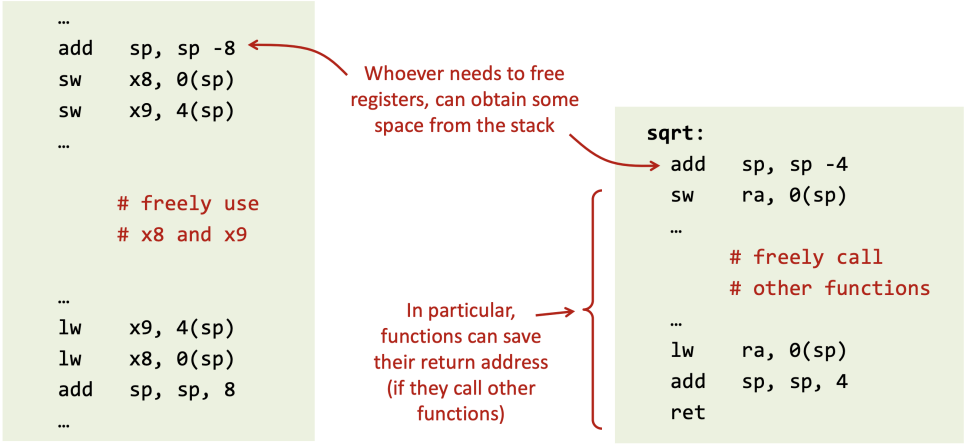
Other architectures have special instructions to place stuff on the stack (push) and to retrieve it (pop)

PUSH AX

```
add sp, sp, -4
sw x5, 0(sp)
```

2.3.8 Spilling Registers to Memory

Spilling registers to memory involves saving register values to the stack when more registers are needed or to prevent overwriting important data, allowing the registers to be reused. This technique is also used in function calls to save the return address, ensuring the program can correctly return control after the function finishes.

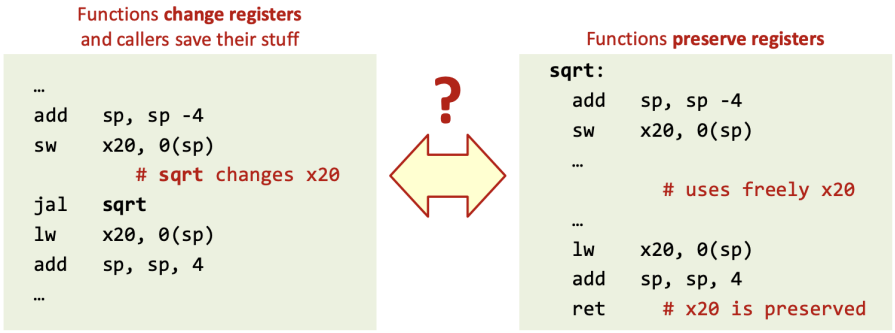


2.3.9 Register across functions

In assembly programming, handling registers across functions can be managed in two main ways: either functions **change registers** and expect the caller to save their values, or functions **preserve registers** and ensure that the register values remain the same across function calls.

- On the left, the function `sqrt` changes the value of register `x20`, requiring the caller to save and restore its value.
- On the right, the function `sqrt` preserves the value of `x20`, ensuring that the caller does not need to manage the saving and restoring.

This distinction is important, but it does not cause issues as long as there is agreement on how registers are handled.
In case it's still not clear, we're looking at the `sw` instruction



2.3.10 Preserving Registers

In RISC-V, register preservation is managed through a combination of callee-saved and caller-saved registers. Callee-saved registers (such as `s0`, `s1`, and `s2-11`) are preserved by the called function, ensuring that their values remain unchanged after the function call.
Caller-saved registers (such as `t0`, `t1-2`, and `t3-6`) are temporary and do not need to be preserved by the called function, meaning the caller must save them if their values are important.

Register	ABI Name	Description	Preserved across call?
x0	zero	Hard-wired zero	—
x1	ra	Return address	No
x2	sp	Stack pointer	Yes
x5	t0	Temporary/alternate link register	No
x6–7	t1–2	Temporaries	No
x8	s0/fp	Saved register/frame pointer	Yes
x9	s1	Saved register	Yes
x18–27	s2–11	Saved registers	Yes
x28–31	t3–6	Temporaries	No

2.4 Passing Arguments in RISC-V

In RISC-V, there are two main ways to pass arguments to functions:

2.4.1 Option 1: Using Registers

- Specific registers are used to pass arguments and return results.

- This can be done in a straightforward way, where each function uses different registers (e.g., passing an argument in `x5` and returning the result in `x6`).

- A more structured approach is to follow a convention where arguments are passed in registers `x10` to `x17`, with results returned in `x10`.

- The limitation: if there are more arguments than available registers (e.g., more than 8 arguments), this approach is insufficient.

2.4.2 Option 2: Using the Stack

- When registers are not enough, extra arguments are placed on the stack.

- The stack offers a universal solution because it has no practical limit on size.

- However, using the stack is more complex and requires additional work compared to using registers.

2.4.3 The RISC-V Approach

- RISC-V uses a combination of both methods.

- Registers `x10` to `x17` are used to pass arguments, with `x10` and `x11` also handling return values.

- If more arguments are needed beyond what these registers can handle, they are passed via the stack.

Register	ABI Name	Description	Preserved across call?
x10–11	a0–1	Function arguments/return values	No
x12–17	a2–7	Function arguments	No

Register reserved for arguments and return values in RISC-V.

2.5 Summary of RISC-V Register Conventions

Not covered in CS-200

Register	ABI Name	Description	Preserved across call?
x0	zero	Hard-wired zero	—
x1	ra	Return address	No
x2	sp	Stack pointer	Yes
x3	gp	Global pointer	—
x4	tp	Thread pointer	—
x5	t0	Temporary/~~alternate link register~~	No
x6–7	t1–2	Temporaries	No
x8	s0/fp	Saved register/~~frame pointer~~	Yes
x9	s1	Saved register	Yes
x10–11	a0–1	Function arguments/return values	No
x12–17	a2–7	Function arguments	No
x18–27	s2–11	Saved registers	Yes
x28–31	t3–6	Temporaries	No

Chapter 3

Part I(c) - ISA Memory and Addressing Modes - W 2.1

3.1 Memory

Memory is a really important component of a computing system, we store our programs in it, we store our data in it, and it's through memory that we receive and send data.

Though memory is very useful it has three main drawbacks:

- It's **slow** → Caches
- It's **finite** → Virtual Memory
- It can make an ISA **too complex** → Pipelining

no worries we'll cover each one of these in this chapter.

3.1.1 Address and Data

Data in Memory can be accessed by an adress, meaning i's a Random Access (it can access a memory value without going through the preceding ones).

Professor Remark: "There's not anything random about this memory, we'd better call it and abitrary access memory. (!not and official name)"

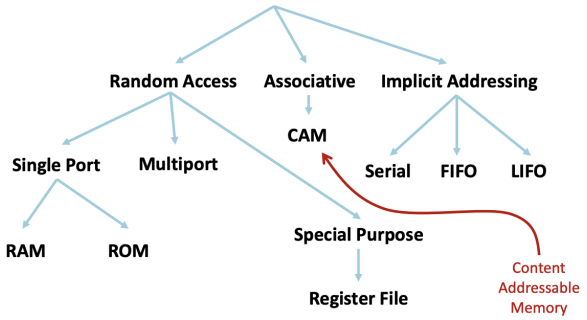
Address	Value
0	12
1	6
2	4
3	1
4	0
5	3
6	1
7	13
8	15
9	9
10	3
11	5
12	0
13	0
14	0
15	0

Write	Read
`Memory[5] = 3`	`Memory[5]?`

3.2 Many Types of Memories

We may distinguish between different types of memories based on their **technology**, such as SRAM, DRAM, EPROM, and Flash, and their **capabilities**, including **speed**, **capacity**, **density**, **writability** (whether they are writable, permanent, or reprogrammable), as well as their **size**, **volatility**, and **cost**.

3.2.1 Functional Taxonomy of Memories

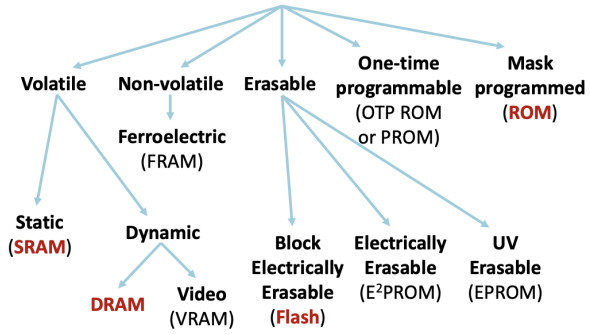


Multiport memory allows simultaneous access by multiple processors, while **single-port** memory supports only one at a time.

Non-Random Access memories

- **Adsociative** memories enable fast data retrieval by content rather than address, making it useful for cache memory, pattern recognition, and efficient lookups in large datasets.
- In **Implicit addressing** the address of the data to be operated on is inferred directly by the operation code (opcode), without explicitly specifying the address in the instruction.

3.2.2 Taxonomy of Random Access Memories

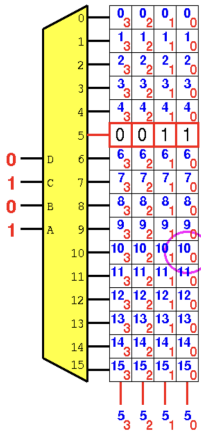


3.2.3 Basic Structure

Remember, a Data Flip Flop, stores a 1 bit value by updating the output value to the input value at the rising edge of the clock signal.

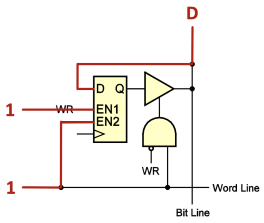
Address	Value
0	12
1	6
2	4
3	1
4	0
5	3
6	1
7	13
8	15
9	9
10	3
11	5
12	0
13	0
14	0
15	0

16 x 4 Memory Cells (Special DFFs (Data Flip-Flops))



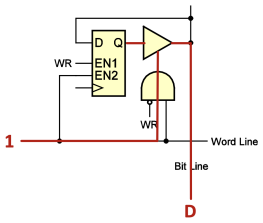
3.2.4 Write Operations

The D is connected to the Data outside of the system and at the risiing edge it updates the value of the DFF. **The AND gate ensures that the write signal is high when the clock signal is high.**



3.2.5 Read Operations

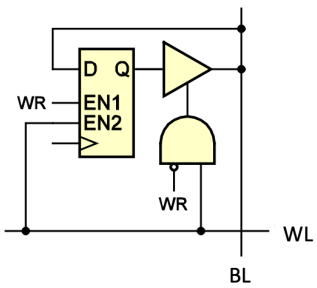
D is still connected to the Data, remember the tri-state driver is active when it's enable signal is active (so when the wr is off and the operation signal is sent.).

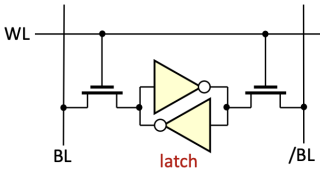


3.2.6 Practical SRAMs

DISCLAIMER !!: Combinational loops are prohibited as they can lead to unstable behavior, unpredictable timing, simulation and synthesis issues, excessive power consumption, and lack of a defined reset state, making them unsuitable for reliable digital circuit design.

While the type of memory we've juste seen is small, and very fast, SRAM memories uses 6 transitors per cell (less than the previous design). We've also seen (in Taxonomy) that SRAM is ***static*** *meaning it doesn't require periodic refresh.*



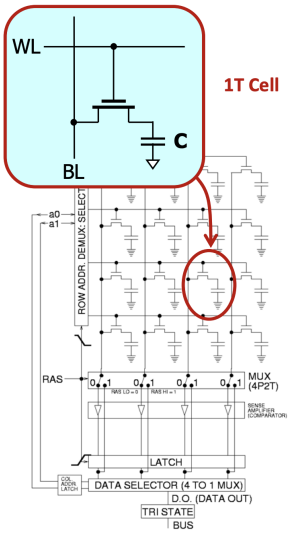


3.2.7 DRAMs

Dynamic RAMS(DRAMs) are the densest and cheapest type of RAM memory, it stores information as charge in small capacitors. This makes the DRAM need periodic refresh otherwise the charge might leak off (60ms) the capacitor due to parasitic resistances and the information lost

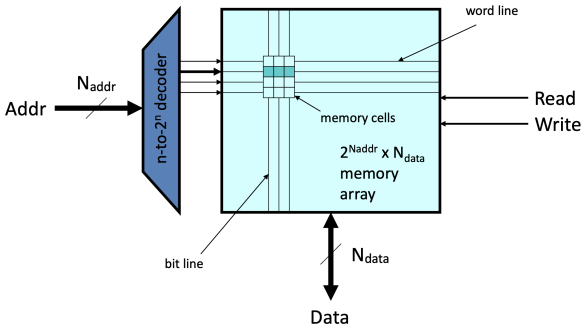
Refresh means, we come back before the end of a charge (60ms) and we rewrite the value, if there is still some charge, we add charge, if there's no charge and we keep as is.

Personal Remark: Dynamic = Bad, data dissapears and needs refresh

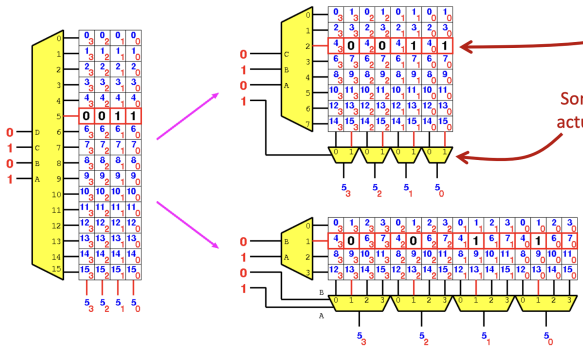


3.2.8 Ideal Random Access Memory

A memory array uses an n-to-2^n decoder to select a word line based on the input address, enabling data to be read or written through the bit lines.



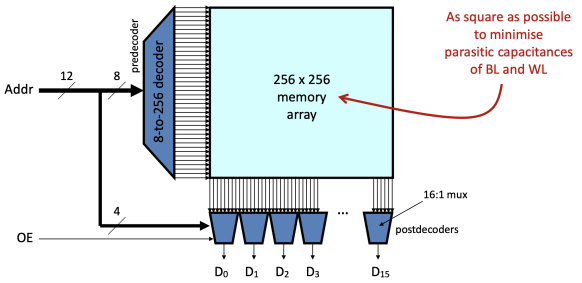
3.2.9 Physical Organisation



Out of all physical organizations, the squared one is the best one as it has the best performance. This layout facilitates faster access times and simplified wiring, resulting in improved computational efficiency and system scalability.

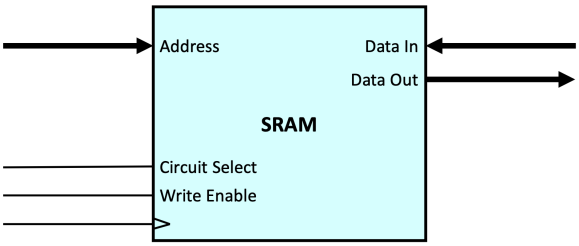
3.2.10 Realistic ROM Array

ROMs are Read-Only Memories, they are used to store the program of the computer, they are non-volatile and can't be written to.



3.2.11 Static Ram Typical Interface

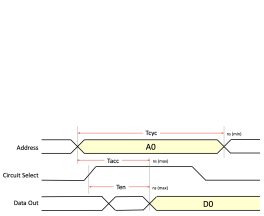
This a typical interface of a SRAM, it has a 16-bit data input/output, a 16-bit address input, a write enable signal, and a circuit select signal.

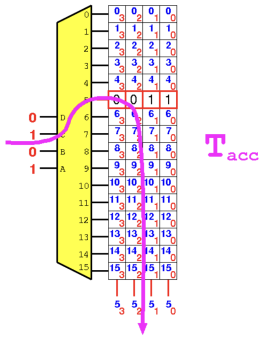


3.3 Typical Asynchronous SRAM Read Cycle

The read cycle of an asynchronous SRAM is initiated by the address input, which is decoded to select the word line, enabling the data to be read from the memory array and output to the data bus.

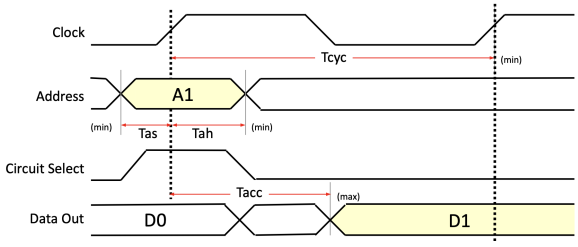
Here, Tcyc is the cycle time, Tacc is the access time, and Ten is the enable time.





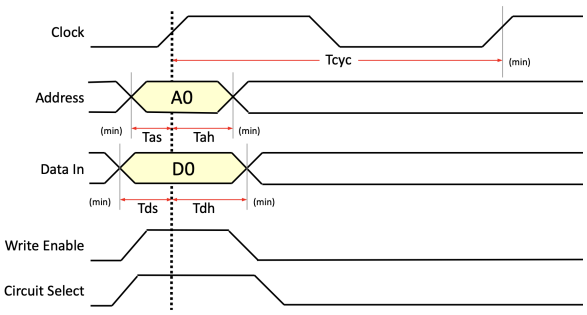
Read Cycle

Latency defined as the number of cycles between the address asserted and data available



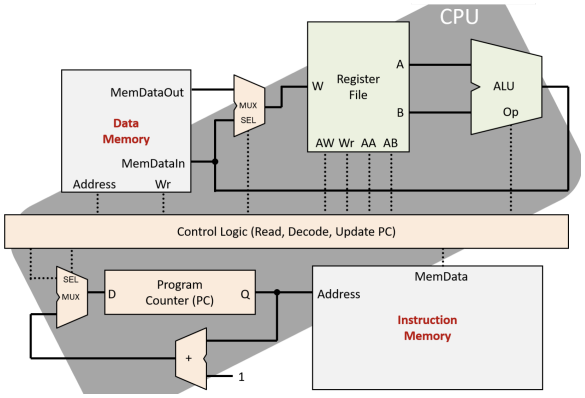
Write Cycle

Writes on the edge of the clock signal, as a DFF



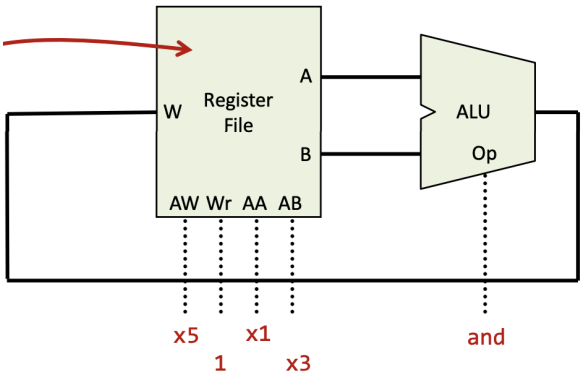
3.4 Where is Memory in the Processor?

In the processor we have memory in the Data memory component and in the Instruction memory component.

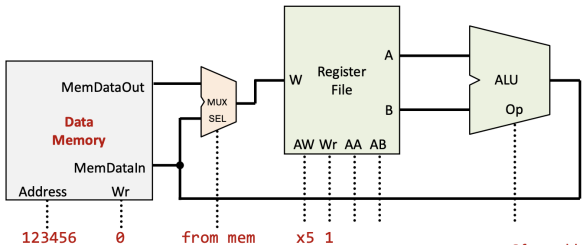


3.4.1 Arithmetic and Logic Instructions

The register file can only contain a limited number of registers making it difficult to handle more complex computations and managing data input/output efficiently.

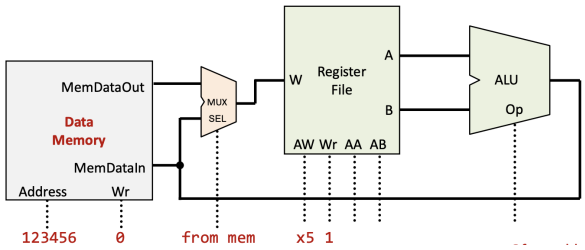


Load Instructions



Load and Store: The RiSC-V Way

This instruction would never work for example because the adress is too big to be sent as an immediate value : `lw x5, (x7)`



A Load/Store Architecture

A feature of RISC-V is that it's a Load/Store architecture, meaning that the only way to access memory is through load and store instructions. Also, instructions reading and writing in memory do exactly that and nothing else, contrary to more complex instruction set architectures (CISC), where instructions may combine memory access with other operations like arithmetic or logic. This simplicity in RISC-V's instruction set helps with streamlining the pipeline and improving performance efficiency.

Load		**I**	0x2	0x03
`lw`	`rd,imm(rs1)`	`rd ← mem[rs1 + sext(imm)]`		
Store		**S**	0x2	0x23
`sw`	`rs2,imm(rs1)`	`mem[rs1 + sext(imm)] ← rs2`		

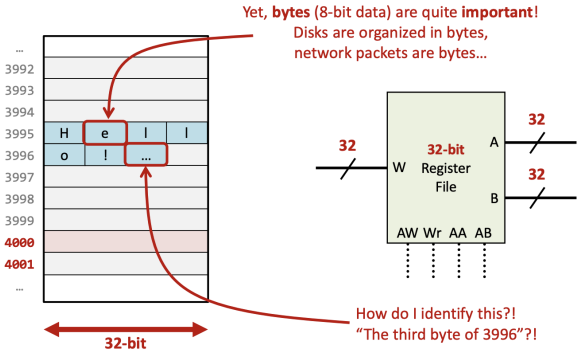
3.5 More Addressing Modes? Not in RISC-V!

Addressing Mode	**Instruction**	**Description**
Register	`add x0, x1, x2`	Adds the value of `x1` and `x2`, stores the result in `x0`.
Immediate	`add x0, x1, 123`	Adds the value of `x1` and the immediate constant 123, stores the result in `x0`.
Direct or Absolute	`add x0, x1, (1234)`	Adds the value of `x1` and the value at memory address 1234, stores the result in `x0`.
Register Indirect	`add x0, x1, (x2)`	Adds the value of `x1` and the value in memory at the address held in `x2`, stores in `x0`.
Displacement or Relative	`add x0, x1, 123(x2)`	Adds the value of `x1` and the value in memory at `x2` plus the displacement 123, stores in `x0`.
Base or Indexed	`add x0, x1, i5(x2)`	Adds the value of `x1` and the value in memory at `x2` plus index `i5`, stores in `x0`.
Auto-increment/-decrement	`add x0, x1, (x2+)`	Adds the value of `x1` and the value in memory at the address in `x2`, then increments `x2`, stores in `x0`.
PC-Relative	`add x0, x1, 123(pc)`	Adds the value of `x1` and the value in memory at `pc` plus 123, stores in `x0`.

Syntax here looks like RISC-V but most of these instructions do not exist in RISC-V.

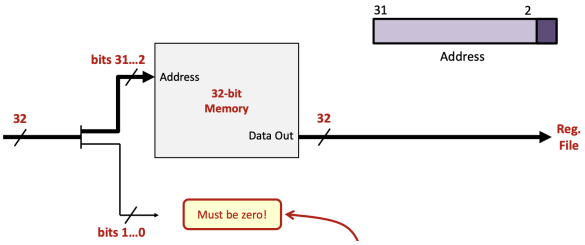
3.5.1 Word Adressed Memory

In a word addressed memory, the address is the index of the word in the memory.
The letters inside the word are identified as eg. for Hello World, H:3980, E:3981, L:3982, ...



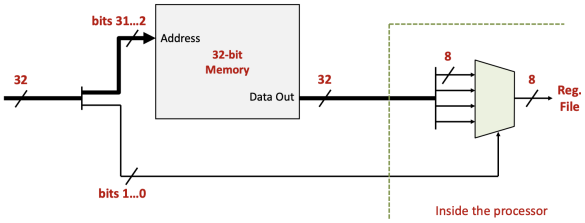
3.5.2 Loading Words (lw) and Instructions

The lw instruction is used to load a word from memory into a register.
The adress of such words would necessarly be a multiple of 4 meaning the two least significant bits must be 0s.(to the ensure the data is word aligned...)



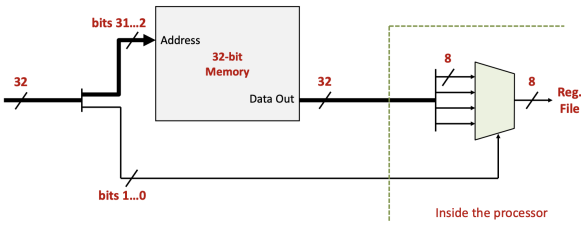
3.5.3 Loading Bytes (lb)

The `lb` (Load Byte) instruction doesn't require alignment because it only loads 1 byte (8 bits), which can be accessed at any memory address, unlike `lw` which requires word alignment to efficiently load 4 bytes (32 bits).
The lb instruction is used to load a byte from memory into a register.



3.5.4 A Few More Load/Store Instructions

Access bytes (and half-words) as if memory were made of bytes

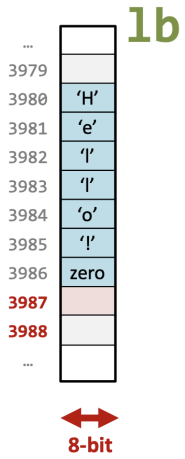


3.5.5 Access as it is more suitable

For example storing the "Hello!"zero value in the memory would like this:

Counting Characters in a String

As an example, for counting the number of characters in a string, the load byte instruction would be more suitable as seeing the string as a sequence of bytes makes use of the memory as a sort of array.




```
strlen:
    mv t0, a0 # Copy the pointer (a0) into t0 to traverse the string
    li t1, 0 # t1 will hold the length (initialized to 0)
loop:
    lbu t2, 0(t0) # Load byte at address t0 into t2
    beq t2, zero, end # If t2 is 0 (null byte), we are done
    addi t1, t1, 1 # Increment the length counter (t1)
    addi t0, t0, 1 # Point to the next character in the string
j loop # Repeat the loop
end:
    mv a0, t1 # Move the length (t1) into a0 as the return value
    ret # Return to caller
```

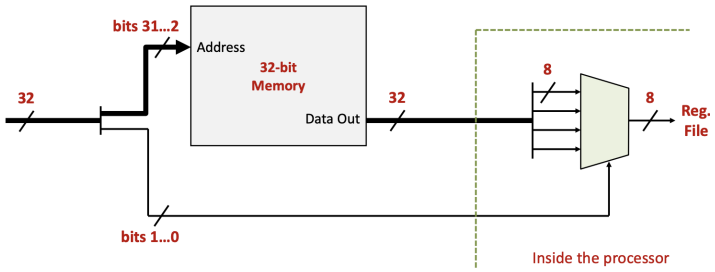
`lbu` is used here to ensure that the byte is treated as an unsigned value, which is the correct approach for processing characters in a string.

In a word addressed memory view, the code would look like such:

```
strlen:
    li t1, 0            # t1 will hold the length (initialized to 0)
next_word:
    li t2, 4            # t2 will count the bytes in a loaded word (four)
    lw t3, 0(t0)        # Load four bytes at address t0 into t3
next_byte:
    andi t4, t3, 0xff   # Move the "little-end" in t4
    beq t4, zero, end   # If t4 is 0 (null byte), we are done
    addi t1, t1, 1      # Increment the length counter (t1)
    srli t3, t3, 8      # Prepare the next byte of the word in the "little-end" (t3)
    addi t2, t2, -1     # One byte left in the loaded word
    bnez t2, next_byte  # If more bytes in t3, check the next
    addi a0, a0, 4      # Else point to the next word of characters in the string
    j next_word         # Repeat the loop
end:
    mv a0, t1           # Move the length (t1) into a0 as the return value
    ret                 # Return to caller
```

3.5.6 Loading Bytes (lb)

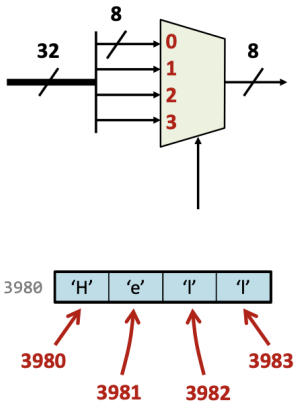
Now, one may wonder in what ordering the bytes are stored in memory.



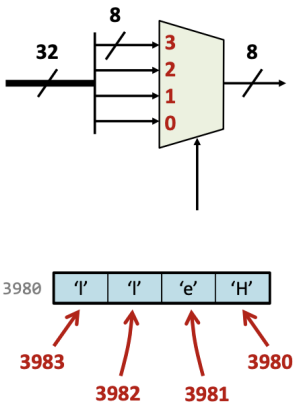
Which Byte Where?

Both ordering of bytes are valid the only thing we have to do is stick to one, the most generally used is little-endian as it's the RISCV default and the Intel x86/x64 default.

Little Endian



Big Endian



Personal Remark : Mnemotechnic - Little Endian = Little End (The ending memory index takes the smallest(starting) data adress), Big Endian = Big End.

Chapter 4

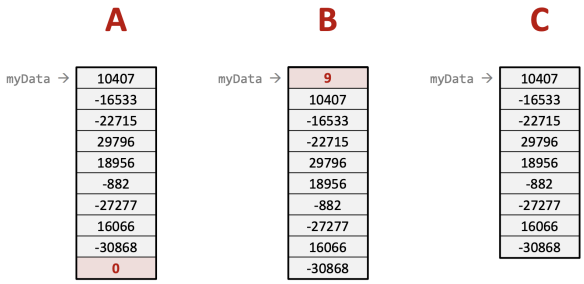
Part I(d) - ISA Arrays and Data Structures - W 2.2

4.1 Arrays

In higher level languages, are written like follows :

```
short\[\] myData = \{10407, -16533, -22715, 29796, 18956, \dots\}:
```

4.1.1 Different Ways to Store Arrays

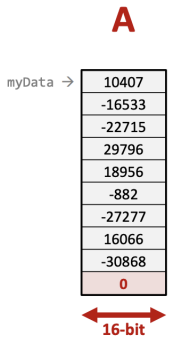


- **A: Storing Arrays with a Null Terminator**
  - In this method, the array is stored with each element represented using 16-bit integers.
  - A null terminator (the value 0) is used at the end of the array to indicate its termination.
  - This method is common when the array size is unknown in advance, and the null terminator acts as a signal to stop reading the data.
- **B: Storing Arrays with a Length Prefix**
  - Here, the first element of the array contains the length of the array, stored as a 16-bit integer (in this case, the length is 9).
  - The rest of the array is stored in consecutive memory locations, similar to method A.
  - This method allows the array size to be known before reading all the data, making it more efficient for some use cases.
- **C: Storing Arrays without a Terminator or Length Prefix**
  - In this case, the array is stored without a length prefix or a null terminator.
  - The size of the array must be known externally, either through the code or an external mechanism.
  - This method is the most compact but requires prior knowledge of the array's size.

4.1.2 Adding Positive Elements

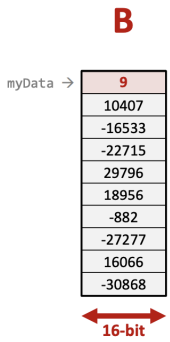
Here we'll write the same program for the different ways of storing arrays.
The program will add all positive elements of an array of signed 16-bit integers. At call time, a0 points to the array, at return time, a0 contains the result.

A: Storing Arrays with a Null Terminator



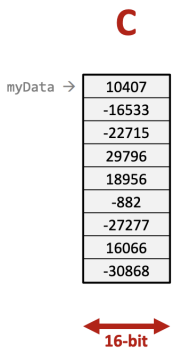

```
add_pos:li t0, 0          # Initialize t0 to 0
        lh t1, 0(a0)      # Load halfword from memory address a0 into t1
        beqz t1, end      # Branch to 'end' if t1 equals zero
        blez t1, donothing  # Branch to 'donothing' if t1 is less than or equal to
            zero
        add t0, t0, t1    # Add t1 to t0 (only if t1 is positive)
donothing:                # This block does nothing for negative or zero values
# You can put other operations here if needed
        j add_pos         # Jump back to the beginning of add_pos to check the next
            value
end:                      # Label 'end' for the program termination
```

B: Storing Arrays with a Length Prefix




```
add_pos_b:
    lh t2, 0(a0)          # Load the length of the array into t2
    addi a0, a0, 2        # Move to the first element of the array (skip the length
        prefix)
    li t0, 0              # Initialize t0 to 0 for storing the sum
loop_b:
    beqz t2, end_b        # If the length (t2) is zero, branch to 'end_b'
    lh t1, 0(a0)          # Load the current array element into t1
    blez t1, skip_b       # If t1 is less than or equal to zero, skip the addition
    add t0, t0, t1        # Add t1 to t0 (only if t1 is positive)
skip_b:
    addi a0, a0, 2        # Move to the next element in the array
    addi t2, t2, -1       # Decrease the length counter
    j loop_b              # Jump back to loop_b
end_b:                    # End label
```

C: Storing Arrays without a Terminator or Length Prefix




```
add_pos_c:
    li t0, 0          # Initialize t0 to 0 for storing the sum
loop_c:
    beqz t2, end_c    # If the array size (t2) is zero, branch to 'end_c'
    lh t1, 0(a0)      # Load the current array element into t1
    blez t1, skip_c   # If t1 is less than or equal to zero, skip the addition
    add t0, t0, t1    # Add t1 to t0 (only if t1 is positive)
skip_c:
    addi a0, a0, 2    # Move to the next element in the array
    addi t2, t2, -1   # Decrease the array size counter
    j loop_c          # Jump back to loop_c
end_c:                # End label
```

4.1.3 Pointer to Memory vs Index in Array

Now we're wodnering which one of these two ways of accessing the array is better.
Obviously the less instructions the better (not always true actually but well), Pointer to Memory.

```
add_positive:
    li t0, 0
    mv t1, a1
next_short:
    beq t1, zero, end
    lh t2, 0(a0)
    bltz t2, negative
    add t0, t0, t2
negative:
    addi a0, a0, 2
    addi t1, t1, -1
    j next_short
end:
    mv a0, t0
ret
```

```
add_positive:
    li t0, 0
    li t1, 0
next_index:
    bge t1, a1, end
    slli t2, t1, 1
    add t2, a0, t2
    lh t3, 0(t2)
    bltz t3, negative
    add t0, t0, t3
negative:
    addi t1, t1, 1
    j next_index
end:
    mv a0, t0
ret
```

In C

Writing this in C for better understanding. Again, which one is better?

Obviously the less instructions the better (again not always true but ah), Index in array

Pointer to memory

```c
short sum = 0;
short *ptr = myData;
short *end = myData + N;
while (ptr < end) {
    if (*ptr > 0) {
    sum += *ptr;
}
    ptr++;
}
```

Index in array

```c
short sum = 0;
int i;
for (i = 0; i < N; i++) {
    if (myData[i] > 0) {
    sum += myData[i];
}
}
```

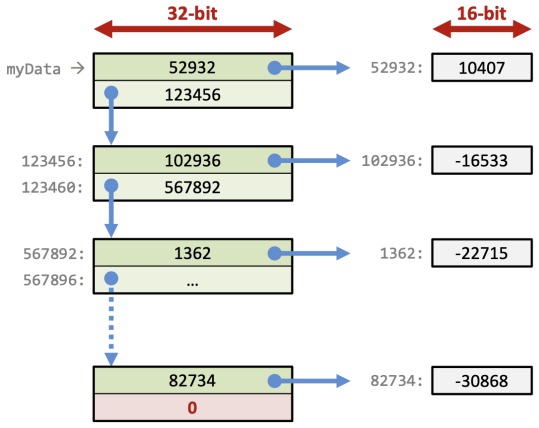
We need a good compiler

Seeing this, the idea would be to have a sufficiently good ***compiler*** *(check I.4.3.2 if needed) such that we write our C code in Index in array, and we get Pointer to memory code in assembly. Thus writing better code but also getting better performance.*

Another type of collection we could've used to store the data is a *Linked List.*

Linked lists are useful for efficiently inserting and deleting elements, especially in the middle of the list.

Each 32-bit element in a linked list contains 16 bits for the value and 16 bits for the address of the next element, enabling efficient insertions but slower sequential access compared to arrays.



Chapter 5

Part I(e) - ISA Arithmetic - W 3.1

5.1 Notation

Before we start, let's define some notation:

- **Number representation (with a fixed number of digits/bits):**

$$A = A^{(n)} = A^{(m)}$$

- **Number in binary or decimal:**

$$A = A_{10} = A_2 = A_{2c}$$

With A_{2c} being the 2's complement representation.
And A_2 being the binary representation.

- **Individual digits or bits:**

$$a_{n-1}, a_{n-2}, \ldots, a_2, a_1, a_0$$

- **Digit string representation:**

$$\langle a_{n-1}a_{n-2}\ldots a_2a_1a_0\rangle$$

5.2 Numbers

Numbers in computing can be represented in different forms, each with specific use cases.

Integers can be either signed or unsigned, representing positive and negative values, or only non-negative values. Examples include:

$$0, 1, 2, 3, 4294967295, -2147483648$$

Fixed-point numbers are essentially integers with an implicit scaling factor (e.g., 10^k or 2^k) to handle fractional values. Common in applications like signal processing. Examples include:

$$0.12, 3.14, 1073741823.75$$

Floating-point numbers represent a wide range of values using a base and exponent, providing flexibility in precision. Examples include:

$$3.14E3, -2.5E1, 1.0E0, 4.2E-2, -1.5E-3$$

5.2.1 Unsigned Integers

Unsigned integers are:

- *Weighted*: Each digit has a positional value.
- *Nonredundant*: Every number has a unique representation.
- *Based on a fixed-radix system*: Typically radix-10 (decimal) or radix-2 (binary).

- *Canonical*: Follows a standard form for representation.

Definition:

$$A = \langle a_{n-1}a_{n-2}\dots a_2a_1a_0\rangle = \sum_{i=0}^{n-1} a_i R^i$$

where A is the unsigned integer, a_i are the digits, and R is the radix.

5.2.2 Signed Integers

We may distinguish between three methods for representing signed integers:

- **Sign-and-Magnitude (SM)**: Uses the most significant bit (MSB) to represent the sign (0 for positive, 1 for negative), with the remaining bits representing the magnitude. This method has the drawback of two zeros (+0 and -0) (Redundant).
- **Two's Complement**(Specific True-and-Complement): The most common way to represent signed integers. It avoids the two-zero problem and simplifies arithmetic operations. Negative numbers are represented by flipping the bits and adding 1.
- **Biased Representation**: Primarily used in floating-point numbers, especially for the exponent part. A fixed bias is added to the actual value to avoid negative exponents. It's rarely used for integers but is another method for handling signed numbers.

Sign and Magnitude

In the sign-and-magnitude representation, the most significant bit (MSB) is used to represent the sign of the number. The remaining bits represent the magnitude.

Definition

$$A = \langle sa_{n-2}a_{n-3}\dots a_2a_1a_0\rangle = (-1)^s \cdot \sum_{i=0}^{n-1} a_i R^i$$

where A is the signed integer, s the most significant bit of A representing the sign of the number, a_i the digits, and R the radix.

Example (Signed 4-bit integer):

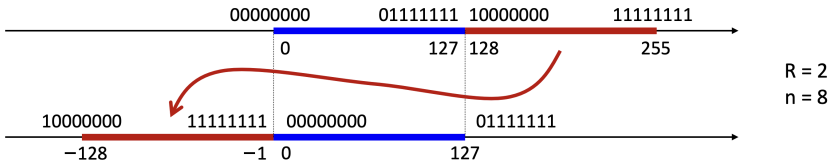
Consider the 4-bit signed binary number 1011_2. In this case:

1. The MSB $s = 1$, indicating the number is negative.
2. The magnitude bits are $011_2 = 3_{10}$.
3. Therefore, the value of the number is -3.

Thus, 1011_2 represents -3_{10} in sign-and-magnitude representation.

5.2.3 Radix's Complement

Radix's complement is a method used to represent signed numbers in different number systems.
It is a special form of *true-and-complement* where the complement $C = R^n$, with R being the radix (base) and n the number of digits.



Definition

A number A in radix's complement is represented as:

$$A = \langle a_{n-1}a_{n-2}\dots a_1a_0\rangle = -a_{n-1}R^{n-1} + \sum_{i=0}^{n-2} a_iR^i$$

where a_{n-1} is the most significant bit, which also indicates the sign (negative for $a_{n-1} = 1$).

For binary numbers, radix's complement is known as **two's complement**, which is the most commonly used method for representing signed numbers in digital systems.

Binary (2's Complement) Representation

Two's complement uses base $R = 2$ and has a fixed word length n.
Here is an example for an 8-bit number system:

Binary	Decimal	Range
00000000	0	Positive range
01111111	127	
10000000	-128	Negative range
11111111	-1	

The two's complement system enables representation of both positive and negative numbers within a fixed bit length.

Decimal (10's Complement) Representation

In a decimal system with radix $R = 10$,
We use 10's complement to represent signed numbers. For instance:

$$5,678^{10c}_{(5)} = 05,678_{10c} = +5,678_{10}$$

This is a positive number representation in 10's complement. For a negative number:

$$9,999,999^{10c}_{(7)} = -1_{10}$$

Here, $9,999,999$ in 7 digits represents -1 in decimal form.

Examples of Binary (2's Complement)

Below are several examples of numbers in binary (2's complement) and their corresponding decimal values:
This is a positive binary number.

$$0100,1101,0010^{2c}_{(12)} = 100,1101,0010_2 = +1,234_{10}$$

This is a negative binary number in 8-bit representation.

$$1111,1111^{2c}_{(8)} = -1_{10}$$

This is a negative binary number in 12-bit representation.

$$1011,0000,1110^{2c}_{(12)} = -1,234_{10}$$

5.2.4 Two's Complement Subtraction

Consider the binary subtraction using the standard paper-and-pencil method:

Borrow:	-1	-1	-1				-1		
	0	0	0	0	1	0	1	0	(10_{10})
$-$	0	0	0	1	0	0	0	1	(17_{10})
	1	1	1	1	1	0	0	1	

Since we had to borrow beyond the most significant bit, the result is negative. The binary result is:

$$-1\ 1\ 1\ 1\ 1\ 0\ 0\ 1_2$$

To find its decimal value:

$$-2^7 + 2^6 + 2^5 + 2^4 + 2^3 + 2^0 = -128 + 64 + 32 + 16 + 8 + 1 = -7$$

and

$$10_{10} - 17_{10} = -7_{10}$$

5.2.5 Addition Is Unchanged from Unsigned

In arithmetic operations, addition remains consistent whether using signed or unsigned numbers. The following instructions are available for basic arithmetic operations:

Arithmetic						
add	rd,rs1,rs2	rd ← rs1 + rs2	R	0x00	0x0	0x33
addi	rd,rs1,imm	rd ← rs1 + sext(imm)	I		0x0	0x13
sub	rd,rs1,rs2	rd ← rs1 − rs2	R	0x20	0x0	0x33

- `add rd, rs1, rs2`: Adds the values in `rs1` and `rs2`, and stores the result in `rd`.
- `addi rd, rs1, imm`: Adds the value in `rs1` with the sign-extended immediate value `imm`, and stores the result in `rd`.
- `sub rd, rs1, rs2`: Subtracts the value in `rs2` from `rs1`, and stores the result in `rd`.

Note that older architectures (e.g., MIPS) had distinct instructions for signed (`add`) and unsigned (`addu`) addition. However, this distinction is unnecessary as the hardware handles both identically.

Sign-and-magnitude addition presents unique challenges, making **two's complement** the standard for signed integers in modern architectures.

5.2.6 Sign Extension

In digital systems, sign extension is a technique used to increase the bit width of a binary number while preserving its value and sign. It is commonly used when converting a number from a smaller to a larger bit width in a way that maintains its original meaning, whether it's unsigned or in two's complement format.

Example: 4-bit to 8-bit Conversion

Consider the 4-bit two's complement number 1110_2, which represents -2_{10}.
When extending this number to 8 bits, we replicate the MSB (which is 1 in this case) to fill the additional bits, as shown below:

$$5_{10} = 0101_2 \quad (4 \text{ bits}) \rightarrow \quad 00000101_2 \quad (8 \text{ bits}).$$

while

$$-2_{10} = 1110_2 \quad (4 \text{ bits}) \quad \rightarrow \quad 11111110_2 \quad (8 \text{ bits}).$$

This ensures that the number remains -2_{10} even after increasing the bit width.

Truncation *is allowed when reducing bit width, but only if the truncated bits are redundant (i.e., copies of the sign bit). For example, going from 8 bits back to 4 bits would result in* 1110_2*, preserving the value* -2_{10}*.*

5.2.7 Signed and Unsigned Instructions

In RISC-V, instructions differentiate between signed (s) and unsigned (u) operations:

Shift						
srl	rd,rs1,rs2	rd ← rs1 $\gg_u$ rs2	R	0x00	0x5	0x33
srli	rd,rs1,imm	rd ← rs1 $\gg_u$ imm	I	0x00	0x5	0x13
sra	rd,rs1,rs2	rd ← rs1 $\gg_s$ rs2	R	0x20	0x5	0x33
srai	rd,rs1,imm	rd ← rs1 $\gg_s$ imm	I	0x20	0x5	0x13
Compare						
slt	rd,rs1,rs2	rd ← rs1 $<_s$ rs2	R	0x00	0x2	0x33
slti	rd,rs1,imm	rd ← rs1 $<_s$ sext(imm)	I		0x2	0x13
sltu	rd,rs1,rs2	rd ← rs1 $<_u$ rs2	R	0x00	0x3	0x33
sltiu	rd,rs1,imm	rd ← rs1 $<_u$ sext(imm)	I		0x3	0x13
Branch						
blt	rs1,rs2,imm	pc ← pc + sext(imm $\ll$ 1), if rs1 $<_s$ rs2	B		0x4	0x63
bge	rs1,rs2,imm	pc ← pc + sext(imm $\ll$ 1), if rs1 $\geq_s$ rs2	B		0x5	0x63
bltu	rs1,rs2,imm	pc ← pc + sext(imm $\ll$ 1), if rs1 $<_u$ rs2	B		0x6	0x63
bgeu	rs1,rs2,imm	pc ← pc + sext(imm $\ll$ 1), if rs1 $\geq_u$ rs2	B		0x7	0x63
Load						
lb	rd,imm(rs1)	rd ← sext(mem[rs1 + sext(imm)][7 : 0])	I		0x0	0x03
lbu	rd,imm(rs1)	rd ← zext(mem[rs1 + sext(imm)][7 : 0])	I		0x4	0x03
lh	rd,imm(rs1)	rd ← sext(mem[rs1 + sext(imm)][15 : 0])	I		0x1	0x03
lhu	rd,imm(rs1)	rd ← zext(mem[rs1 + sext(imm)][15 : 0])	I		0x5	0x03

- **Shift:** `sra`, `srai` (s) vs. `srl`, `srli` (u).
  - Signed shifts preserve the sign bit, while unsigned shifts insert zeroes.
- **Compare:** `slt`, `slti` (s) vs. `sltu`, `sltiu` (u).
  - Signed comparisons use two's complement, unsigned comparisons ignore sign.
- **Branch:** `blt`, `bge` (s) vs. `bltu`, `bgeu` (u).
  - Signed branches use two's complement; unsigned branches do not consider sign.
- **Load:** `lb`, `lh` (s) vs. `lbu`, `lhu` (u).
  - Signed loads extend the sign bit, while unsigned loads extend with zeroes.

5.3 Overflow

Overflow occurs when the result of an arithmetic operation exceeds the range of values that can be represented with a fixed number of bits. This can happen in both unsigned and signed arithmetic, though the detection method differs. In general, overflow results in an incorrect outcome that needs to be detected and handled.

5.3.1 Overflow in 2's Complement

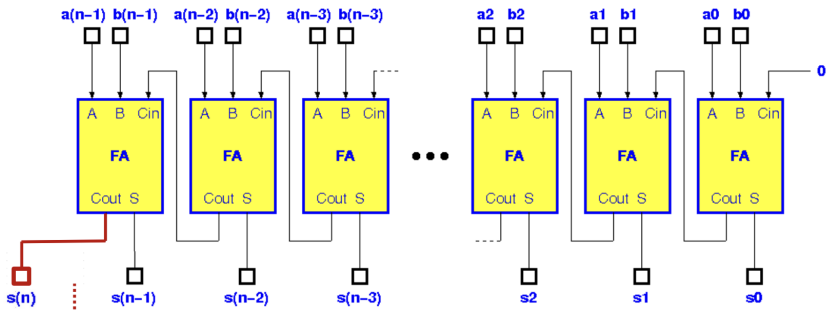
In 2's complement arithmetic, overflow occurs when the result of an addition or subtraction operation falls outside the representable range for the number of bits. For an n-bit 2's complement system, the representable range is -2^{n-1} to $2^{n-1} - 1$.
Overflow is detected by examining the carry into and out of the most significant bit (MSB). Specifically, overflow occurs if:

$$\text{Overflow} = \text{Cout}_{n-1} \oplus \text{Cout}_n$$

Where:

- Cout_{n-1} is the carry into the MSB.
- Cout_n is the carry out of the MSB.

An overflow occurs when these two carry bits differ. This is because the sign of the result is incorrect if there is a mismatch, leading to an incorrect outcome.

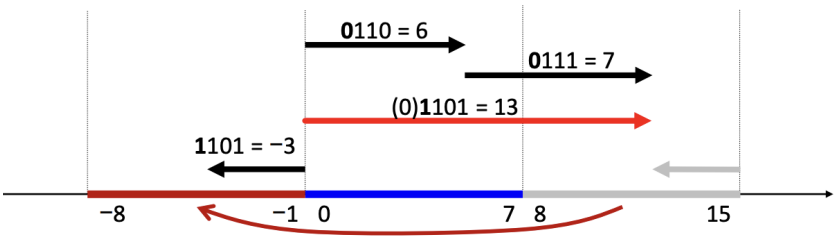


For example, if two large positive numbers are added and result in a negative value (or two negative numbers added result in a positive value), this indicates an overflow in 2's complement addition.

5.3.2 Overflow in Software

In many architectures, detecting overflow during arithmetic operations is a critical aspect of software implementation. Overflow occurs when the result of an addition or subtraction exceeds the capacity of the register used to store it. Detection methods vary depending on the type of architecture:

- **Traditional architectures (e.g., x86):** These systems provide a *carry bit* in a special register, known as a flag, that is set when an overflow occurs. Thus, overflow detection operates similarly to hardware-based overflow detection.
- **Modern architectures (e.g., RISC-V):** These architectures typically provide only the result of the addition or subtraction without a carry bit. Overflow detection must be handled in software, based on analyzing the sign and magnitude of the result.



Overflow detection can be based on the following observations:

- **Addition of opposite sign numbers:** The magnitude of the result decreases, making overflow impossible.
- **Addition of same sign numbers:** Overflow is possible if the result exceeds the range representable by the register, leading to an incorrect sign in the result.

5.3.3 Detect Addition Overflow in Software

- Add two 32-bit signed integers and detect overflow
  - At call time, a0 and a1 contain the two integers
  - On return, a0 contains the result and a1 must be nonzero in case of overflow

5.3.4 Detect Addition Overflow in Software

- Add two 32-bit signed integers and detect overflow
  - At call time, `a0` and `a1` contain the two integers.
  - On return, `a0` contains the result and `a1` must be nonzero in case of overflow.

```
srai a2, a0, 31        # a2 = sign of a0 (0 or -1)
srai a3, a1, 31        # a3 = sign of a1 (0 or -1)
xor  a4, a2, a3        # a4 = 0 if signs are same, -1 if different
add  a0, a0, a1        # compute sum in a0
srai a5, a0, 31        # a5 = sign of sum (0 or -1)
xor  a6, a2, a5        # a6 = 0 if sign of sum same as a0, -1 if different
and  a1, a4, a6        # a1 = -1 if overflow occurred, else 0
srli a1, a1, 31        # a1 = 1 if overflow occurred, else 0
```

5.4 A Strange but Useful Property

Personal Remark: don't mistake A and $\overline{A}$ as sets of elements which might confuse you. They are binary numbers.
In binary arithmetic, there is a particularly useful property that can be expressed as follows:

$$A + \overline{A} = -1$$

or equivalently,

$$-A = \overline{A} + 1$$

Proof: Consider a binary number $A = a_{n-1}2^{n-1} + \sum_{i=0}^{n-2} a_i 2^i$, where $a_i \in \{0, 1\}$ represents the binary digits of A. The complement of A, denoted $\overline{A}$, is given by replacing each a_i with its complement $\overline{a_i}$.

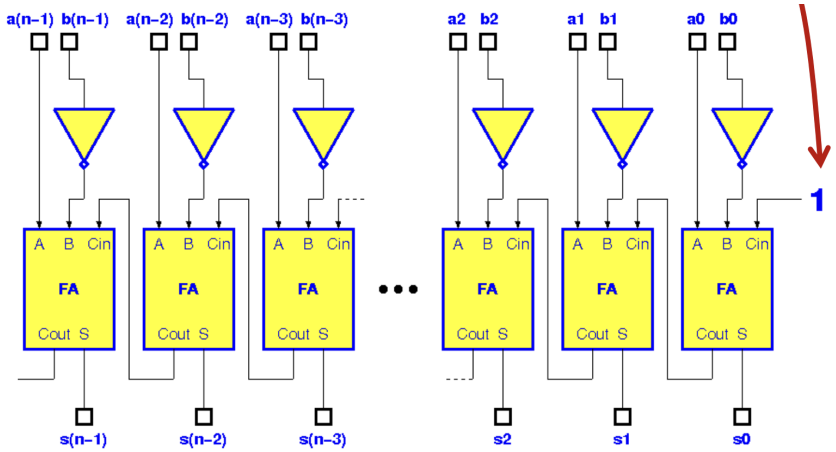
$$\begin{aligned} A + \overline{A} &= \left(-a_{n-1}2^{n-1} + \sum_{i=0}^{n-2} a_i 2^i\right) + \left(-\overline{a_{n-1}}2^{n-1} + \sum_{i=0}^{n-2} \overline{a_i} 2^i\right) \\ &= -(a_{n-1} + \overline{a_{n-1}}) \cdot 2^{n-1} + \sum_{i=0}^{n-2} (a_i + \overline{a_i}) \cdot 2^i \\ &= -2^{n-1} + \sum_{i=0}^{n-2} 2^i = -1 \end{aligned}$$

Where $\overline{A}$ is the two's complement of A.
Intuition: For each binary digit, adding a_i and its complement $\overline{a_i}$ results in 1. Therefore, $A + \overline{A}$ consists entirely of 1s, representing -1 in two's complement.

5.4.1 Two's Complement Subtractor

Using the property of two's complement, we can create a subtractor circuit. The subtractor is implemented using an adder, where the number to be subtracted is inverted and incremented by 1.

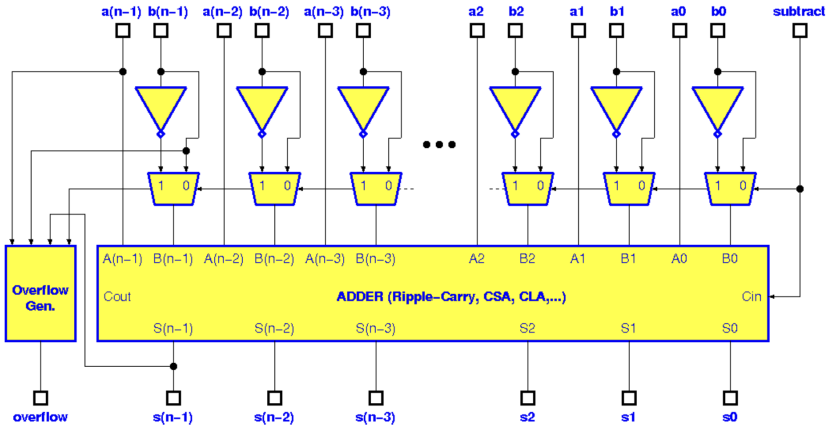
- **Step 1: Inversion of Subtrahend (B)**
  The subtrahend B is inverted using NOT gates, as shown in the diagram. This converts B into its one's complement.
- **Step 2: Addition of A and Inverted B**
  The full adders (FA) add each bit of the minuend A to the inverted bits of B. The full adders also handle any carry-over from the previous addition.
- **Step 3: Add 1 (Two's Complement)**
  To complete the two's complement operation, a carry-in of 1 is added to the least significant bit (LSB), which effectively adds 1 to the inverted B.
- **Output:**
  The sum outputs S $(s_0, s_1, s_2, ...)$ represent the result of the subtraction $A - B$, while the final carry-out can be used to detect overflow.



5.4.2 Two's Complement Add/Subtract Unit

This circuit performs both addition and subtraction using two's complement arithmetic. The operation is selected based on the control input signal for subtraction. The unit consists of several key components:

- **Input Inversion:** Each bit of the subtrahend B is passed through a XOR gate controlled by the 'subtract' signal. When the 'subtract' signal is high (logic 1), the bits of B are inverted to form the two's complement of B, effectively switching the operation to subtraction.
- **Addition:** The ripple-carry adder, represented by the ADDER block, performs binary addition of the bits from A and B. The carry-in (Cin) to the least significant bit is used to add 1 when performing subtraction, completing the two's complement process.
- **Overflow Detection:** The overflow generator block detects if the result of the addition/subtraction operation has exceeded the range representable by the fixed number of bits. The 'overflow' output is asserted in such cases.
- **Output:** The result of the operation is provided as the sum output (S), representing either the sum $A + B$ or the result of $A - B$, depending on the control signal.



5.5 Bounds Check Optimization

Very, very, very useful. When working with signed integers (e.g., array indices), a common task is to ensure that the index remains within a valid range, typically $0 \leq t0 < N$, where N is some predefined boundary. This can be achieved efficiently using a single branch check that combines both lower and upper bound constraints.

Single Branch Bound Check

The instruction `bgeu` (branch if greater than or equal, unsigned) can perform two checks at once:

$$\texttt{bgeu}\ t0, t1, \texttt{out\_of\_bound}$$

Here, $t0$ is the signed number to be checked, and $t1 = N$ is the boundary.

Explanation

- If $t0 \geq 0$, the behavior of `bgeu` mimics that of `bge` (branch if greater than or equal) for signed integers, thus effectively performing an upper bound check.
- If $t0 < 0$, since the comparison is unsigned, $t0$ will appear as a very large positive value, hence automatically triggering the out-of-bound case.

This approach efficiently checks both the lower and upper bounds in one instruction, streamlining the bounds checking process.

5.6 Floating Point Representation

Floating point numbers are widely used in computing to represent real numbers in a way that supports a wide dynamic range.

They are composed of a *significand* (or *mantissa*) and an *exponent* of the base. This representation allows for the approximation of very large and very small values, similar to the way scientific notation is used in everyday practices.

Such as

$$\begin{aligned} 0.18\ \mu\text{m} &\rightarrow 0.18 \cdot 10^{-6}\ \text{m} \rightarrow 1.8 \cdot 10^{-7}\ \text{m} \\ 75\ \text{km} &\rightarrow 75 \cdot 10^{3}\ \text{m} \rightarrow 7.5 \cdot 10^{4}\ \text{m} \end{aligned}$$

In floating point representation, a number X is expressed as:

$$X = (-1)^s \cdot \left(\sum_{i=0}^{n-1} a_i \cdot 2^i \right) \cdot 2^{\left(-e_{m-1}2^{m-1} + \sum_{j=0}^{m-2} e_j 2^j\right)}$$

where:

- s is the sign bit,
- a_i represents the bits of the significand (in sign-and-magnitude form),
- e_j represents the bits of the exponent (in 2's complement form).

Properties of Floating Point Numbers

- **Large dynamic range**, but *variable accuracy*.
- Numbers are **redundant** unless *normalized*.
- Floating point operations are **not associative**, unlike real numbers.
- Exponents are typically stored in a **biased signed representation**, making zero easier to represent and simplifying comparisons in hardware.
- The **mantissa** (significand) is usually normalized such that $1 \leq m < 2$, with a *hidden bit* to store the leading 1.

Standardization and Hardware Support

Floating point representation is standardized by the IEEE 754 standard, which is widely adopted in modern computing systems:

- **x86/x64** architectures have supported floating point operations through SSE/AVX extensions since 1999.
- **RISC-V** also includes support for floating point through ISA extensions.

Example: Decimal to IEEE 754 Simple Precision (32 Bits) Conversion

Convert -7.75 to IEEE 754 single-precision:

Step 1: Sign Bit (1 Bit)

$s = 1$ (negative number).

Step 2: Binary Conversion

$7_{10} = 111_2$, $0.75_{10} = 0.11_2$, so $7.75_{10} = 111.11_2$.

Step 3: Normalize

$111.11_2 = 1.1111_2 \times 2^2$.

Step 4: Exponent (8 Bits)

$E = 2 + 127 = 129$, $129_{10} = 10000001_2$.

Step 5: Mantissa (23 Bits)

Take the fractional part after the leading 1 and pad with zeros to make 23 bits:

1111 0000 0000 0000 0000 000
(fractional part after the leading 1).

Step 6: IEEE 754 Representation

$$\underbrace{1}_{\text{Sign bit}}\ \underbrace{10000001}_{\text{Exponent bits}}\ \underbrace{1111\ 0000\ 0000\ 0000\ 0000\ 000}_{\text{Mantissa}}$$

5.6.1 Sign-and-Magnitude Addition

(Assembly)
In this exercise, we aim to write a function in RISC-V assembler to sum two 32-bit signed numbers represented in sign-and-magnitude (S&M) format. The result should also be produced in the sign-and-magnitude format.

- The two operands are stored in registers `a0` and `a1` on entry.
- The result should be placed in register `a0`.
- Overflow cases should be ignored.

```
# Load the operands from a0 and a1
# Mask the sign bits
andi t0, a0, 0x7FFFFFFF   # Extract magnitude of a0
andi t1, a1, 0x7FFFFFFF   # Extract magnitude of a1

# Extract the signs (MSB) of both operands
srai t2, a0, 31           # Sign of a0
srai t3, a1, 31           # Sign of a1

# Compare the signs
beq t2, t3, same_sign     # If signs are equal, sum the magnitudes
sub t0, t0, t1            # If signs differ, subtract the magnitudes
j check_result            # Jump to check the result

same_sign:
    add t0, t0, t1            # Add magnitudes if signs are the same

check_result:
    # Restore the sign in the result
    slli t0, t2, 31           # Shift the sign bit into position
    or a0, t0, t1             # Combine sign and magnitude into a0
```