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 of the PDF, you can check the following link: `https://elazdi-al.github.io/comparch/index.html`. Feel free to send a pull request to propose any changes you think might be a useful addition to the course content or a modification.
`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

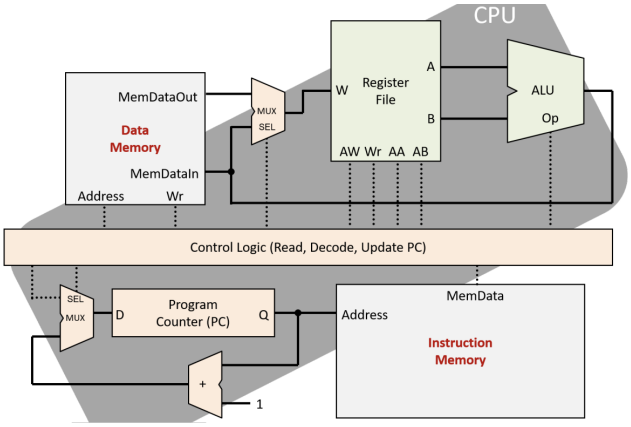
We use this code because it enables us to build a *Finite State Machine*, which isn't feasible with C code. This language provides a more rigid format with a sequence of numbered instructions, an *opcode*, predefined variable names, and the ability to **jump between lines**.

```
    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

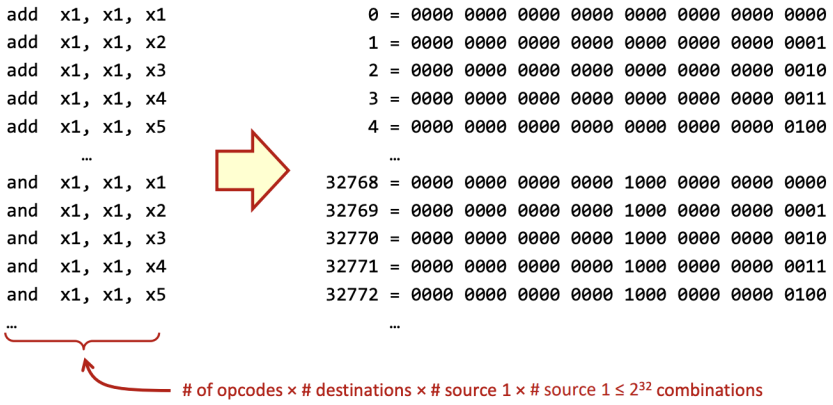
Remember, a processor can be decomposed into five components:

- **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.

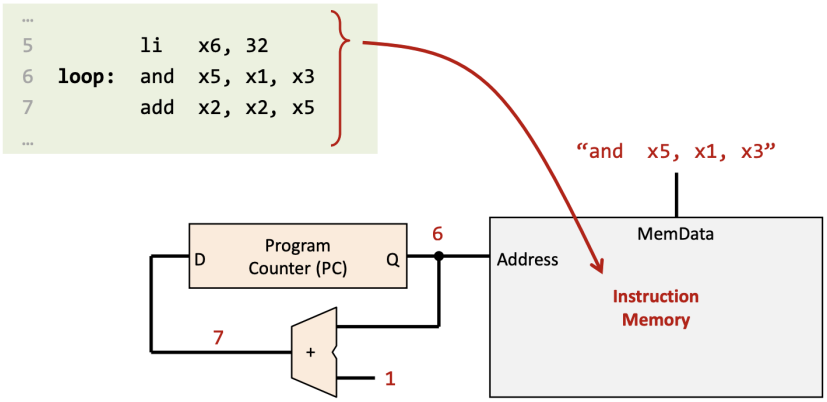


We may distinguish three types of general operations made by the processor:

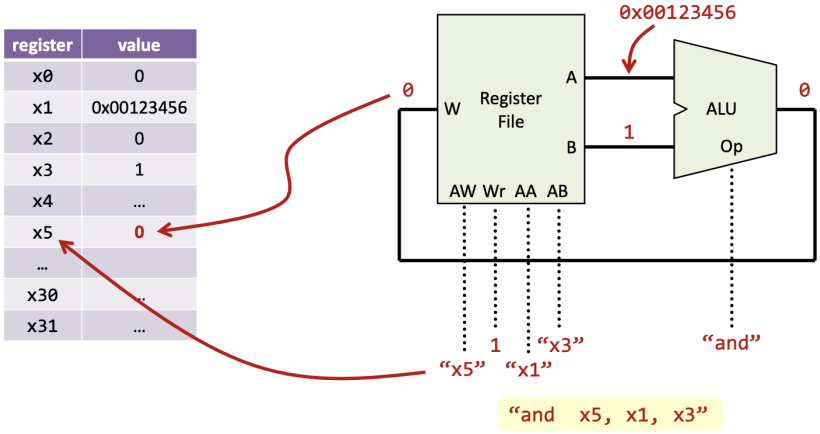
Encoding



Fetching



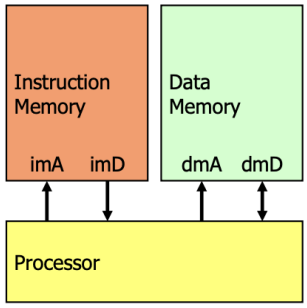
Executing



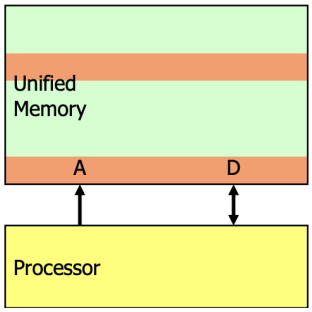
1.3 Joint or Disjoint Program and Data Memories

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

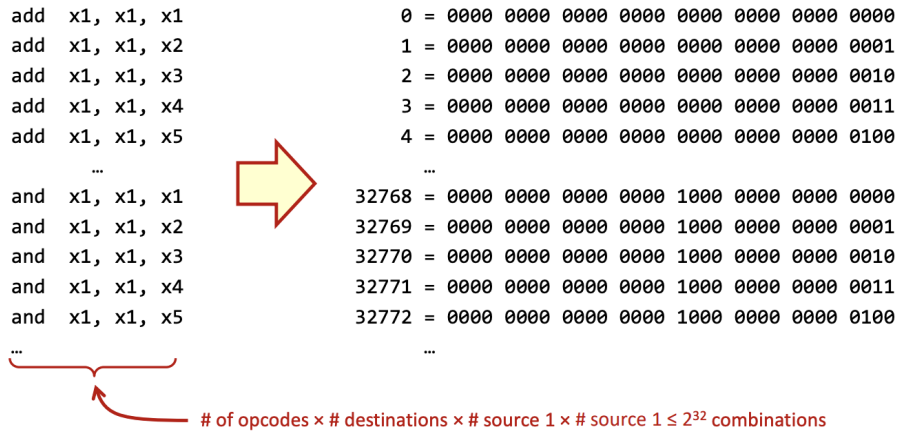


1.4 The Encoding problem

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

1.4.1 The Stupid Solution

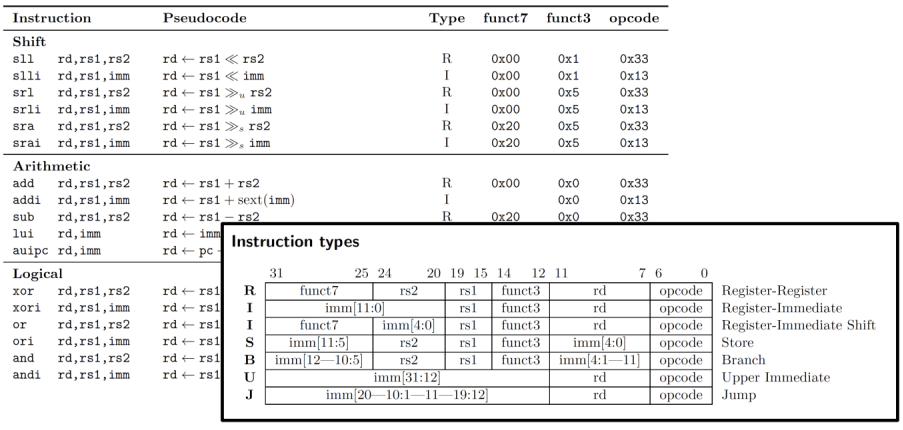
Now, the professor throws out the "stupid idea"(his words) of just counting all possible instructions, assigning a number to each one, and writing the numbers in binary. The problem with such a method is that the number of instructions could grow exponentially, requiring an unmanageable number of bits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

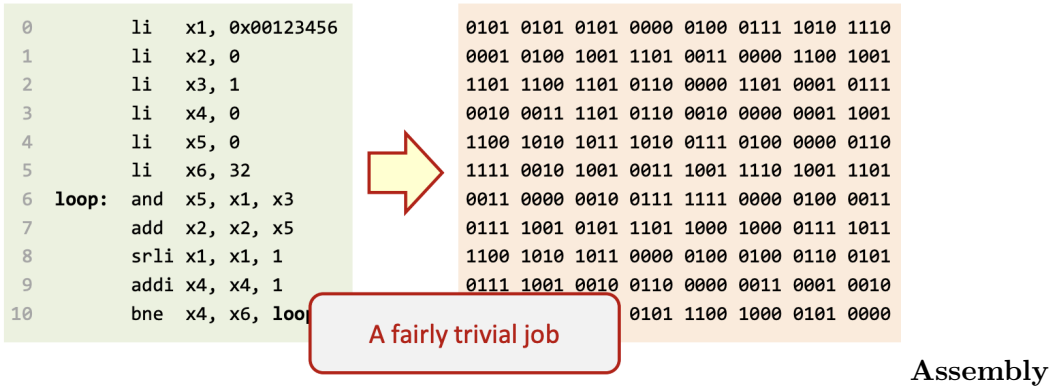
RISC-V encoding

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

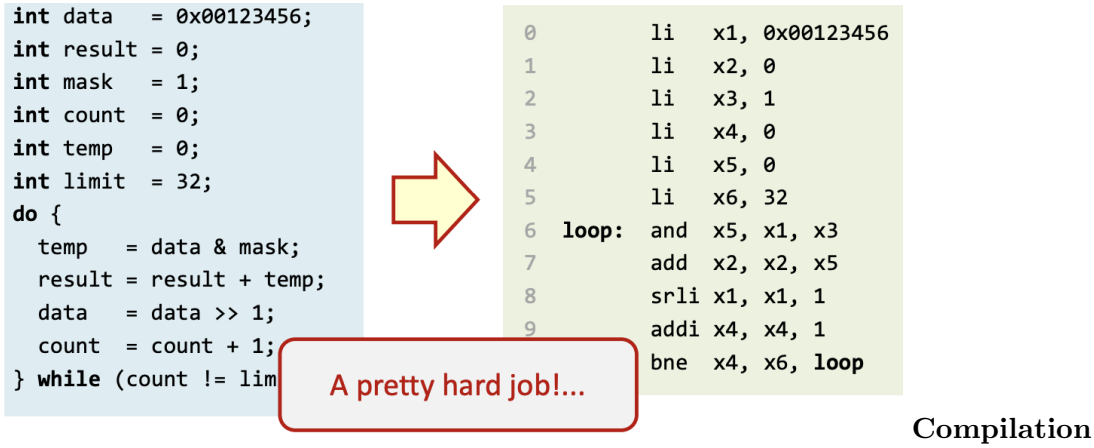
The program that does this is called an assembler. It takes the assembly code and converts it into machine code.



Assembly

Compiler

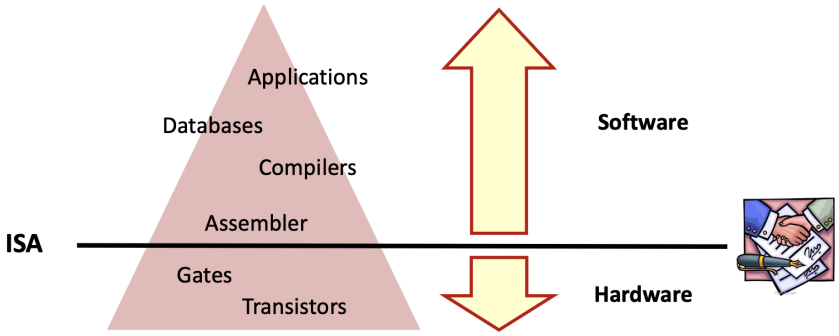
A compiler is a program that translates high-level source code written in languages like C or Java into machine code or an intermediate representation.



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?

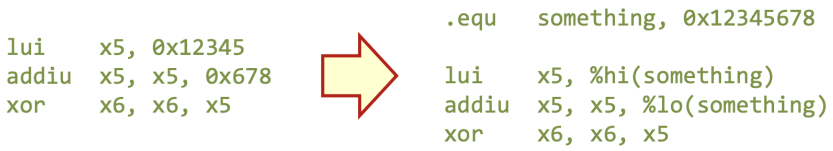
2.1.1 Constants must be encoded on 12 bits

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 20 bits (`0x12345`) into the register `x5`.
- The second instruction, `addiu`, adds the lower 12 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 20 bits into `x5`.
- The `%lo(something)` adds the lower 12 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

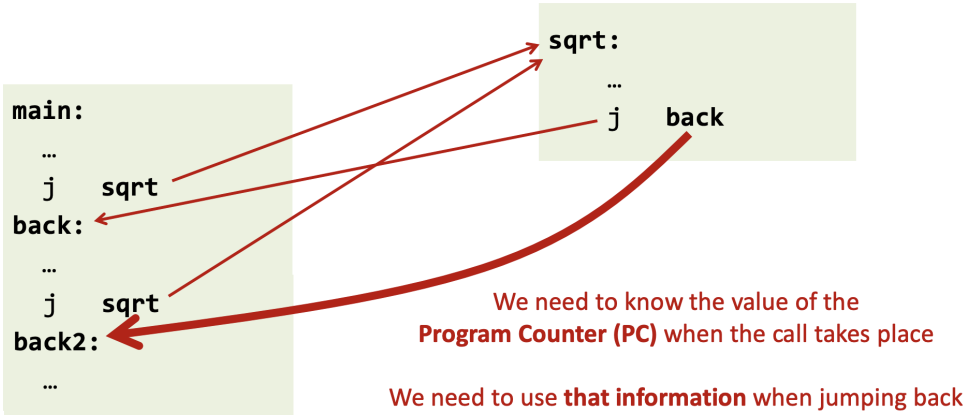
In higher-level programming languages, functions (routines, subroutines, procedures, methods, etc.) are used to encapsulate code and make it reusable.
Calling a function involves these steps:

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

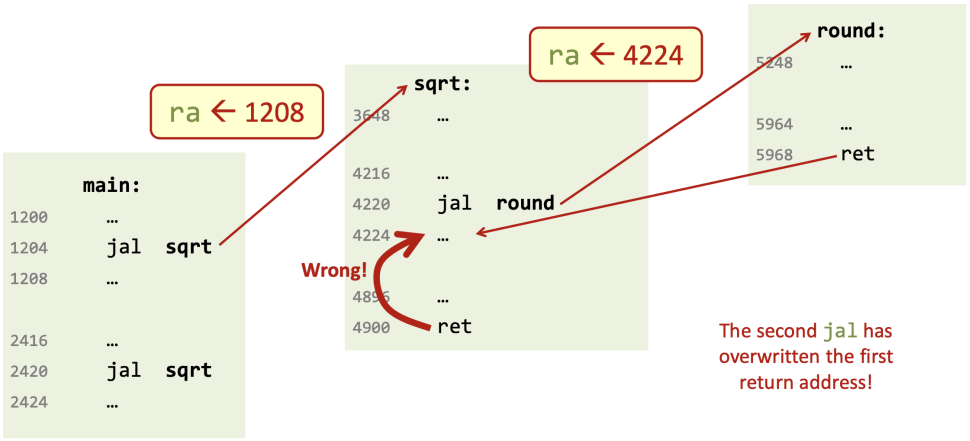
2.3.3 Register Conventions

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

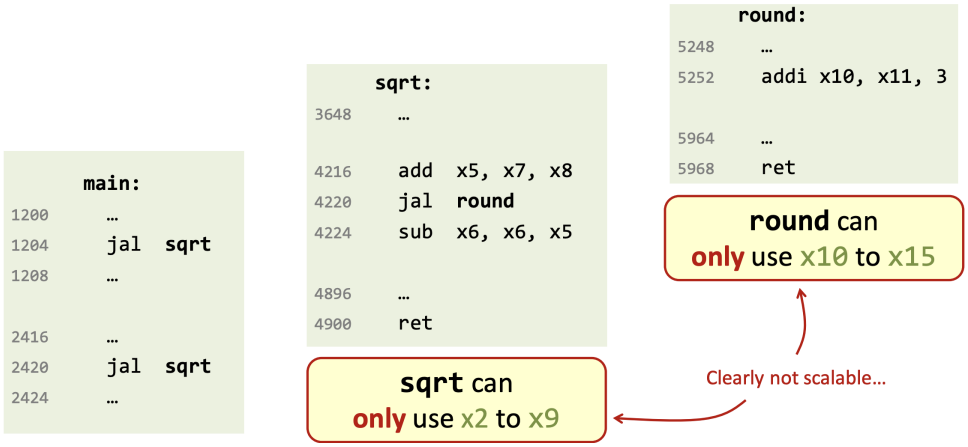
There's still a problem with the previous approach, say for example you want to call a function from another function.



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
```

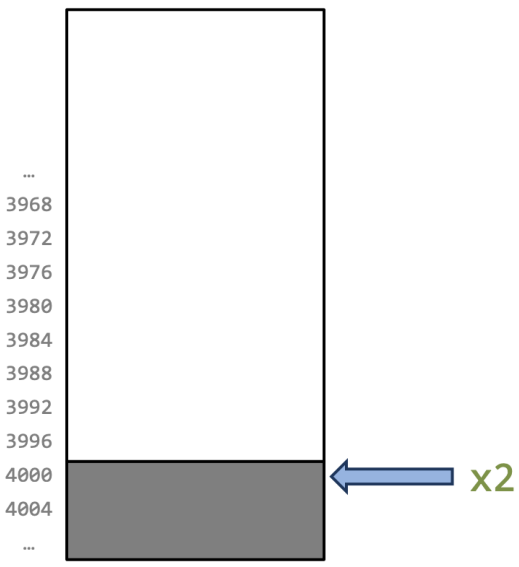
Problem: Recursive Functions

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.

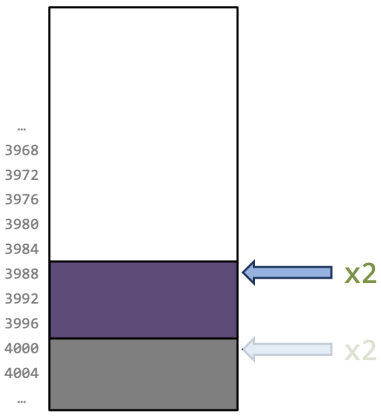


Dynamic Memory Allocation

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
```

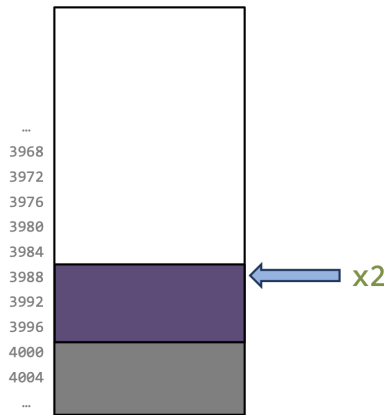


Retrieving Data from the Stack

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)
```

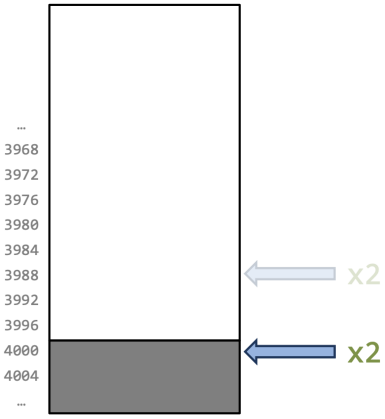


Memory Deallocation

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.

```
1 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

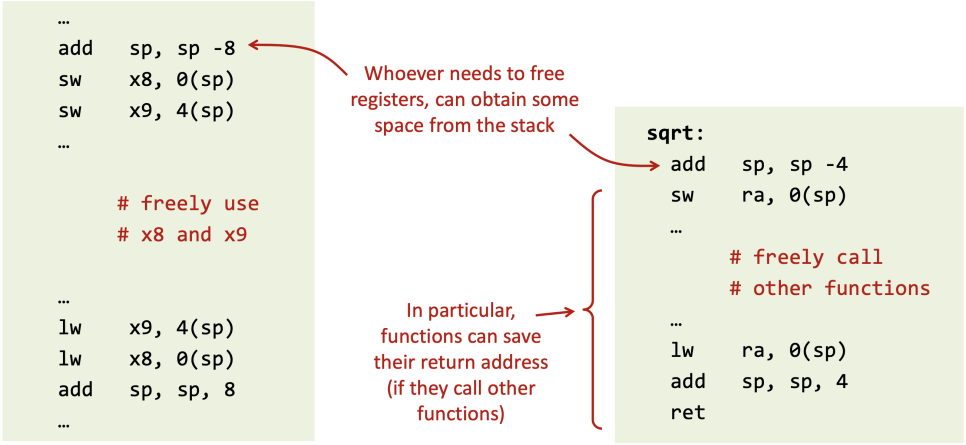
Other architectures have special instructions to place stuff on the stack (push) and to retrieve it (pop)

PUSH AX

```
1 add sp, sp, -4
2 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.

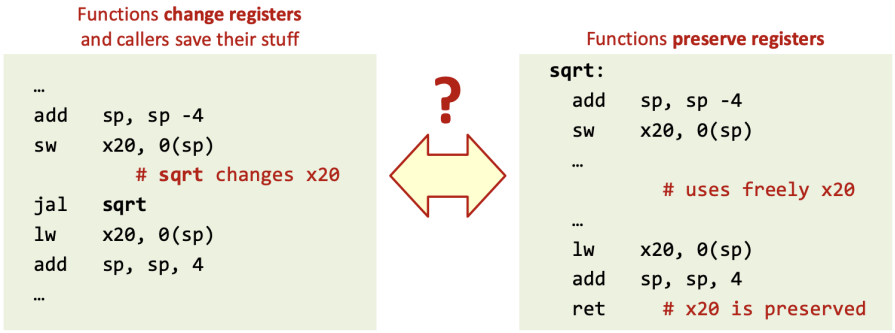


2.3.9 Register across functions

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

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."

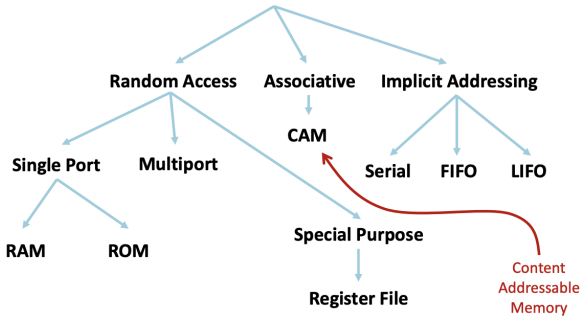
Address	Value
0	12
1	6
2	4
3	1
4	0
5	3
6	1
7	13
8	15
9	9
10	3
11	5
12	0
13	0
14	0
15	0

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

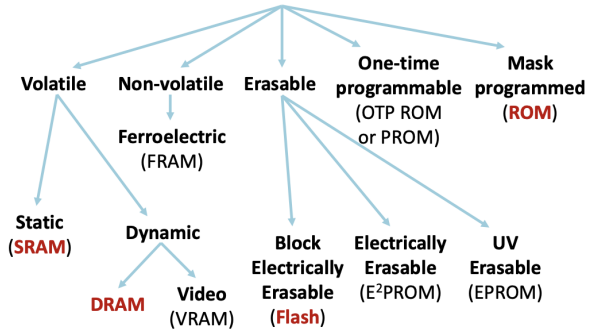


Multiport memory allows simultaneous access by multiple processors, while **single-port** memory supports only one at a time.

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

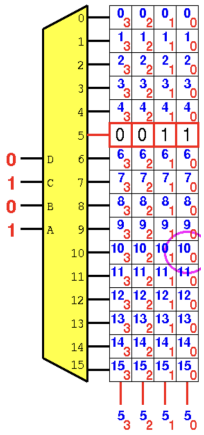


3.2.3 Basic Structure

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.

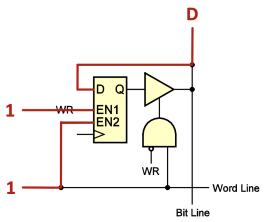
Address	Value
0	12
1	6
2	4
3	1
4	0
5	3
6	1
7	13
8	15
9	9
10	3
11	5
12	0
13	0
14	0
15	0

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



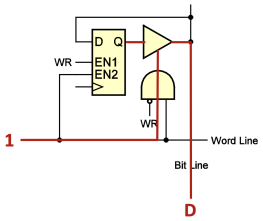
3.2.4 Write Operations

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

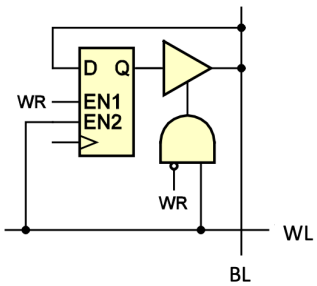
D is still connected to the Data, remember the tri-state driver is active when it's enable signal is active (so when the wr is off and the operation signal is sent.).

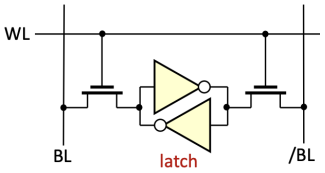


3.2.6 Practical SRAMs

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.*



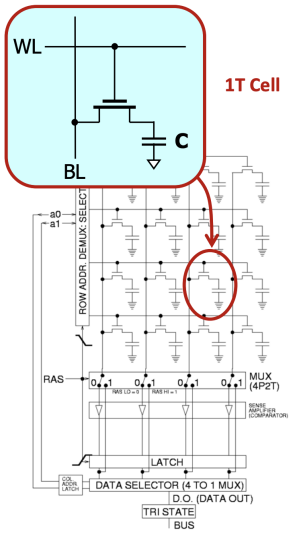


3.2.7 DRAMs

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

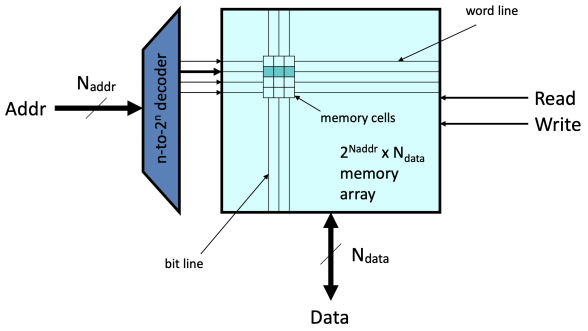
Refresh means, we come back before the end of a charge (60ms) and we rewrite the value, if there is still some charge, we add charge, if there's no charge and we keep as is.

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

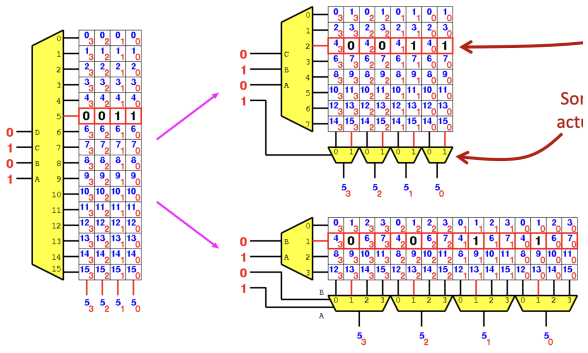


3.2.8 Ideal Random Access Memory

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.



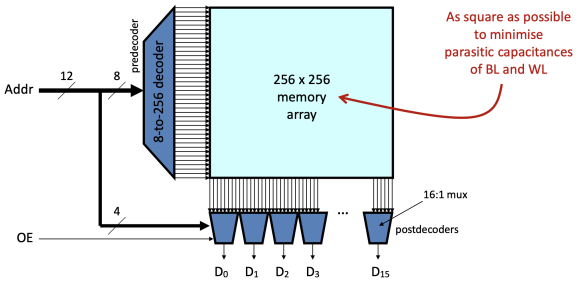
3.2.9 Physical Organisation



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.

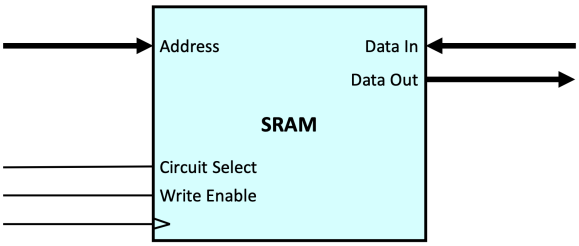
3.2.10 Realistic ROM Array

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

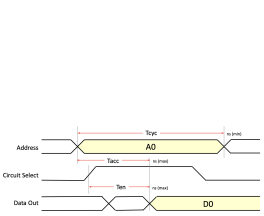
This a typical interface of a SRAM, it has a 16-bit data input/output, a 16-bit address input, a write enable signal, and a circuit select signal.

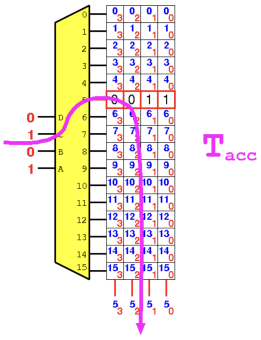


3.3 Typical Asynchronous SRAM Read Cycle

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.

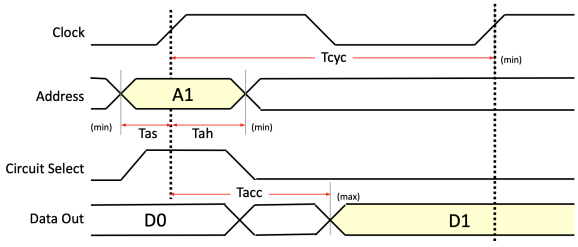
Here, Tcyc is the cycle time, Tacc is the access time, and Ten is the enable time.





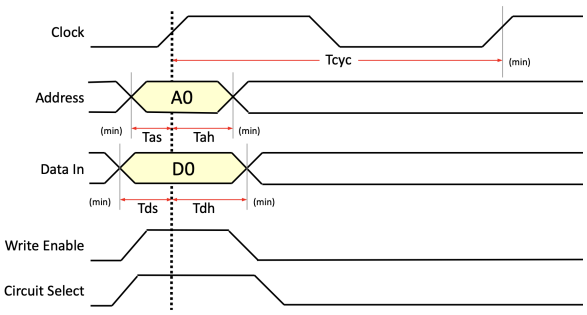
Read Cycle

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



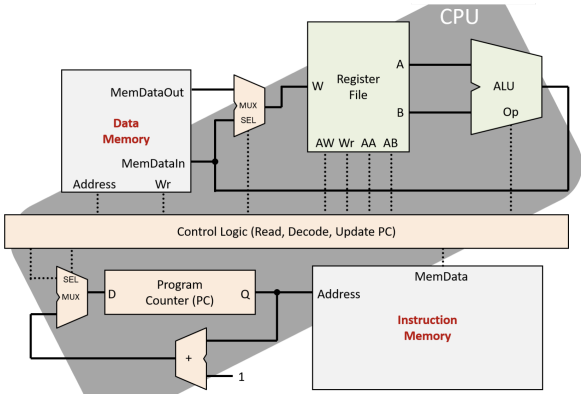
Write Cycle

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



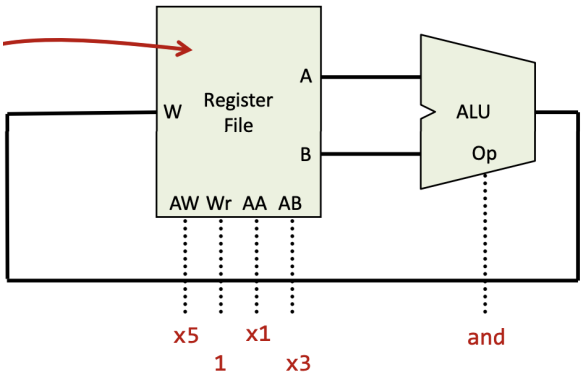
3.4 Where is Memory in the Processor?

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

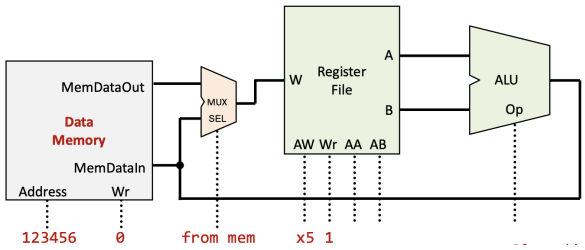


3.4.1 Arithmetic and Logic Instructions

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.

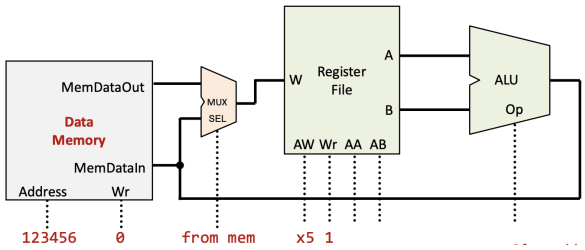


Load Instructions



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`		

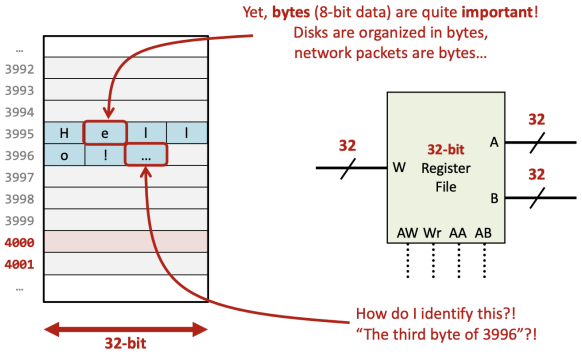
3.5 More Addressing Modes? Not in RISC-V!

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.

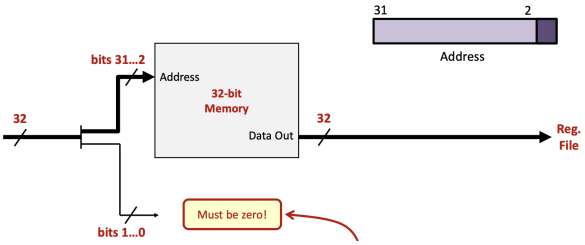
3.5.1 Word Adressed Memory

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, ...



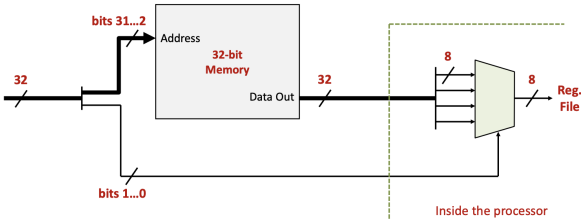
3.5.2 Loading Words (lw) and Instructions

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...)



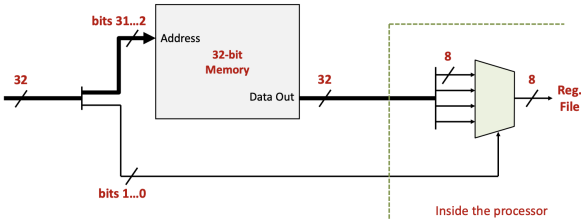
3.5.3 Loading Bytes (lb)

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

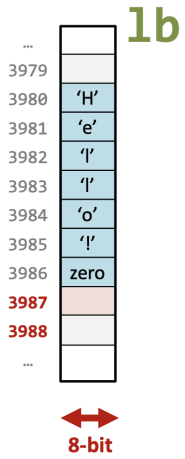


3.5.5 Access as it is more suitable

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
```

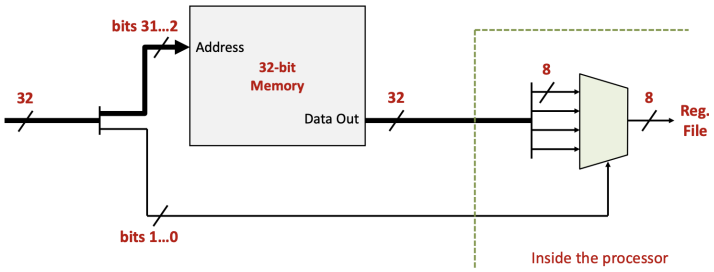
`lbu` is used here to ensure that the byte is treated as an unsigned value, which is the correct approach for processing characters in a string.

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)

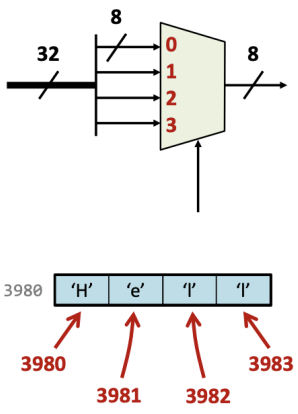
Now, one may wonder in what ordering the bytes are stored in memory.



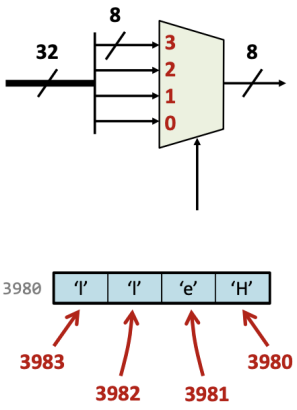
Which Byte Where?

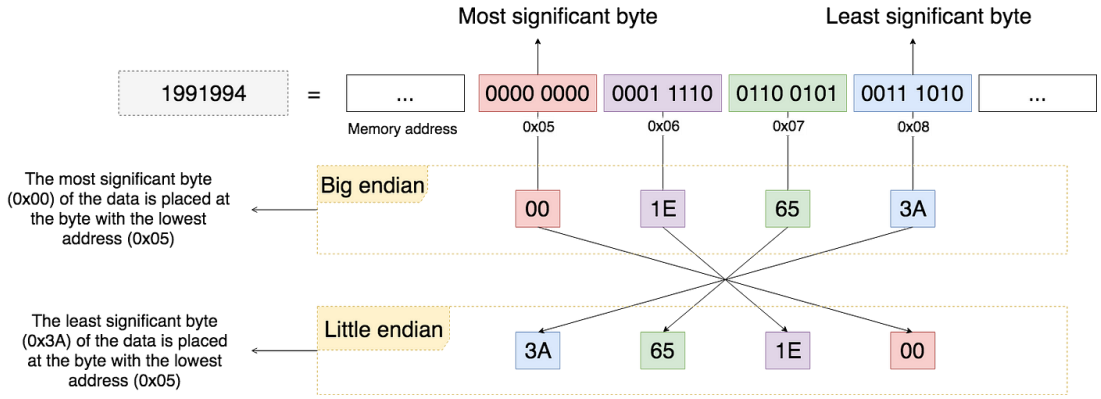
Both ordering of bytes are valid the only thing we have to do is stick to one, the most generally used is little-endian as it's the RISCV default and the Intel x86/x64 default.

Little Endian



Big Endian





source

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

Chapter 4

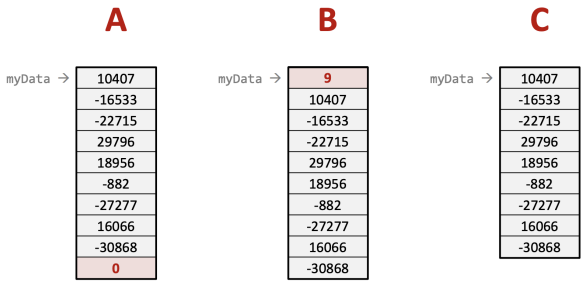
Part I(d) - ISA Arrays and Data Structures - W 2.2

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

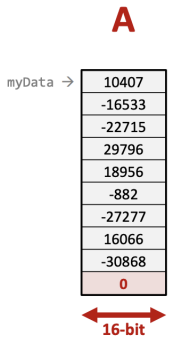


- **A: Storing Arrays with a Null Terminator**
  - In this method, the array is stored with each element represented using 16-bit integers.
  - A null terminator (the value 0) is used at the end of the array to indicate its termination.
  - This method is common when the array size is unknown in advance, 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

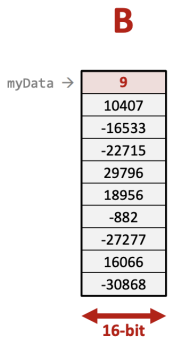
Here we'll write the same program for the different ways of storing arrays.
The program will add all positive elements of an array of signed 16-bit integers. At call time, a0 points to the array, at return time, a0 contains the result.

A: Storing Arrays with a Null Terminator



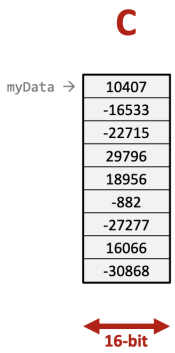

```
add_pos:li t0, 0          # Initialize t0 to 0
        lh t1, 0(a0)      # Load halfword from memory address a0 into t1
        beqz t1, end      # Branch to 'end' if t1 equals zero
        blez t1, donothing  # Branch to 'donothing' if t1 is less than or equal to
            zero
        add t0, 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.

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
```

Index in array

```
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];
}
}
```

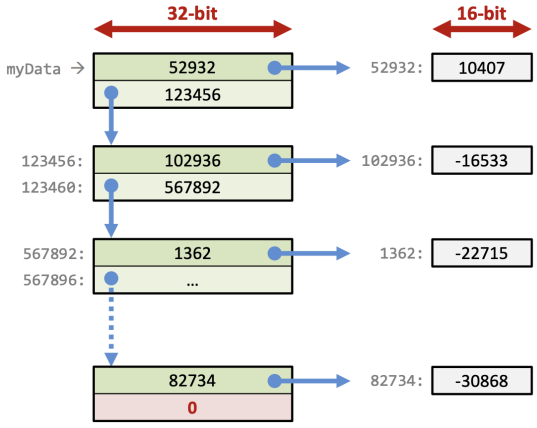
We need a good compiler

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, 3.2

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:

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

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

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

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):

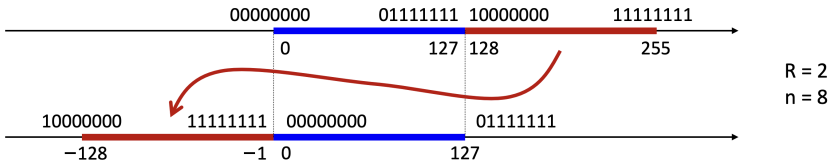
Consider the 4-bit signed binary number 1011_2. In this case:

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

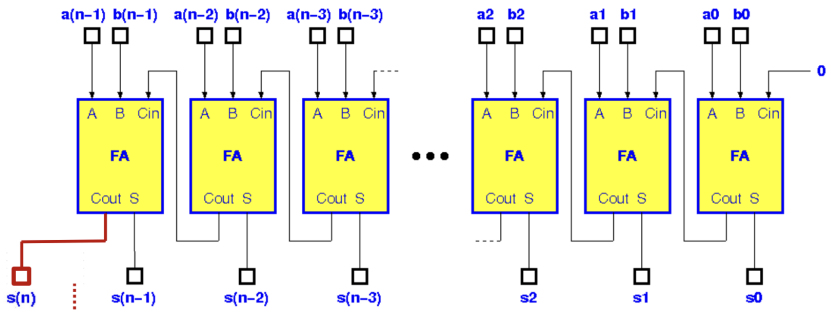
In 2's complement arithmetic, overflow occurs when the result of an addition or subtraction operation falls outside the representable range for the number of bits. For an n-bit 2's complement system, the representable range is -2^{n-1} to $2^{n-1} - 1$.
Overflow is detected by examining the carry into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

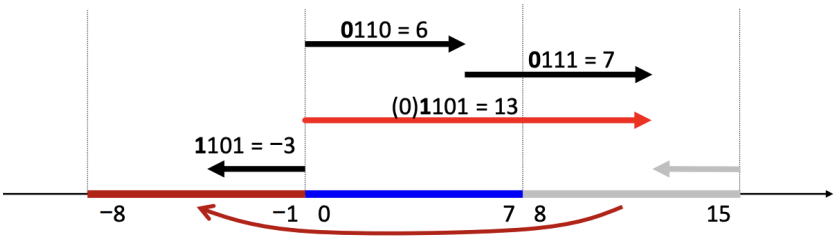


For example, if two large positive numbers are added and result in a negative value (or two negative numbers added result in a positive value), this indicates an overflow in 2's complement addition.

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.

```
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 $\overline{a_i}$.

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

$$= -(a_{n-1} + \overline{a_{n-1}}) \cdot 2^{n-1} + \sum_{i=0}^{n-2} (a_i + \overline{a_i}) \cdot 2^i$$

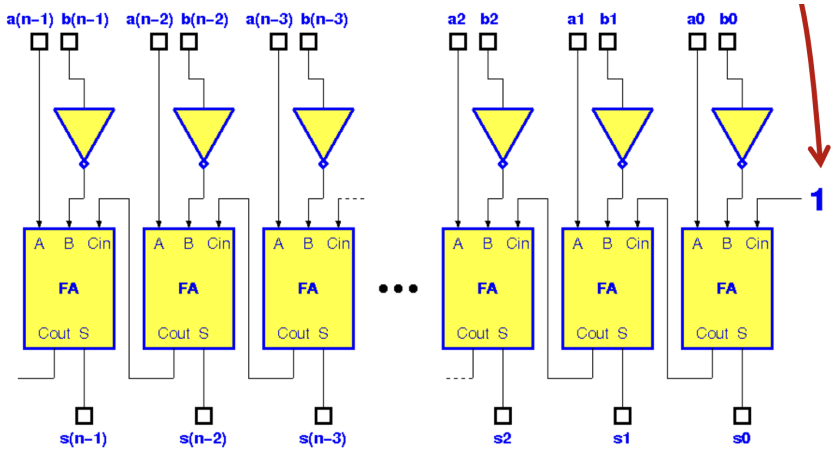
$$= -2^{n-1} + \sum_{i=0}^{n-2} 2^i = -1$$

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.

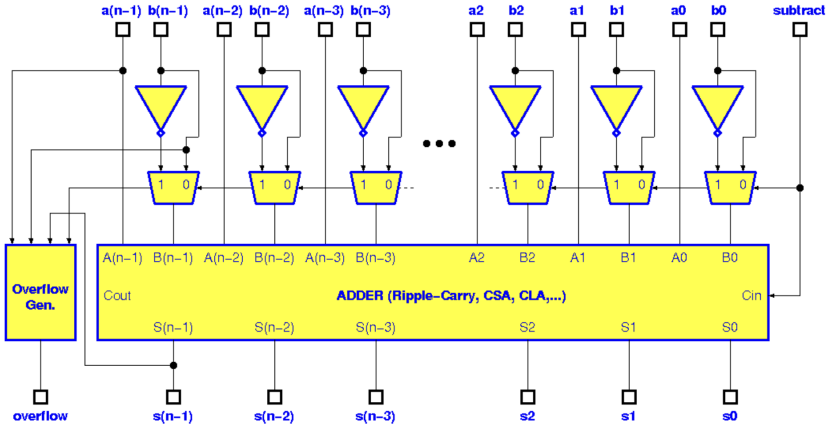


5.4.2 Two's Complement Add/Subtract Unit

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

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

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

$\underbrace{1}$	$\underbrace{10000001}$	$\underbrace{1111\ 0000\ 0000\ 0000\ 0000\ 000}$
Sign bit	Exponent bits	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.

Solution 1

Basic Algorithm:

- If the operands have the same sign:
  - Add the absolute values
  - Attach to the result the same sign as the operands
- If the operands have different signs:
  - Subtract the smallest absolute value from the largest one
  - Attach to the result the sign of the largest value

```
add_sandm:
    lui     t1, 0x80000         # mask for sign bit
    and     t0, a0, t1          # check a0 sign
    beqz    t0, a0_positive     # if positive, skip
    xor     a0, a0, t1          # flip sign bit
    neg     a0, a0              # negate a0

a0_positive:
    and     t0, a1, t1          # check a1 sign
    beqz    t0, a1_positive     # if positive, skip
    xor     a1, a1, t1          # flip sign bit
    neg     a1, a1              # negate a1

a1_positive:
    add     a0, a0, a1          # add values
    and     t0, a0, t1          # check result sign
    beqz    t0, sum_positive    # if positive, skip
    neg     a1, a1              # negate a1
    xor     a0, a0, t1          # flip sign bit

sum_positive:
    ret                         # return result
```

Solution 2

Basic Algorithm:

- Convert the two operands from sign-and-magnitude to 2's complement
- Add the two operands
- Convert the result from 2's complement back to sign-and-magnitude

```
convert_to_twos_comp:
    lui     t1, 0x80000          # mask for sign bit
    and     t0, a0, t1           # check a0 sign
    beqz    t0, a0_twos_comp     # if positive, skip
    xor     a0, a0, t1           # flip sign bit
    neg     a0, a0               # negate a0

a0_twos_comp:
    and     t0, a1, t1           # check a1 sign
    beqz    t0, a1_twos_comp     # if positive, skip
    xor     a1, a1, t1           # flip sign bit
    neg     a1, a1               # negate a1

add_twos_comp:
    add     a0, a0, a1           # add values

convert_to_sign_mag:
    lui     t1, 0x80000          # mask for sign bit
    and     t0, a0, t1           # check result sign
    beqz    t0, result_positive # if positive, skip
    xor     a0, a0, t1           # flip sign bit
    neg     a0, a0               # negate result

result_positive:
    ret                          # return result
```

Chapter 6

Part II(a) - I/O - Exceptions Multicycle Processor W - 3.2, 4.1

In this chapter we will be discussing how we can actually design a processor (subject of our LAB B).

6.1 Processor

(yes, one more time) A processor is composed of several fundamental components that work together to perform computations.

- **Program Counter (PC):** Holds the address of the next instruction to be executed from the instruction memory. It increments after each instruction fetch or is updated based on control logic.

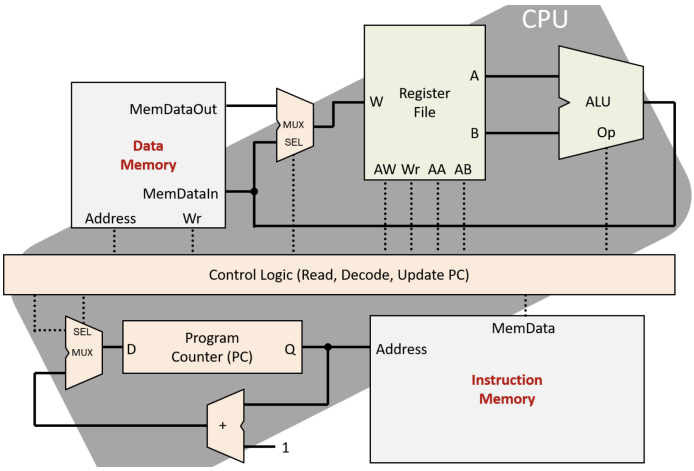
- **Instruction Memory:** Stores the instructions that the processor fetches and executes. Instructions are read sequentially unless altered by control logic.

- **Control Logic:** Manages the flow of data and the sequence of operations, including reading instructions, decoding them, and updating the program counter.

- **Register File:** A set of registers where data is temporarily stored. It allows the processor to access and manipulate values quickly. Each register has read/write capabilities.

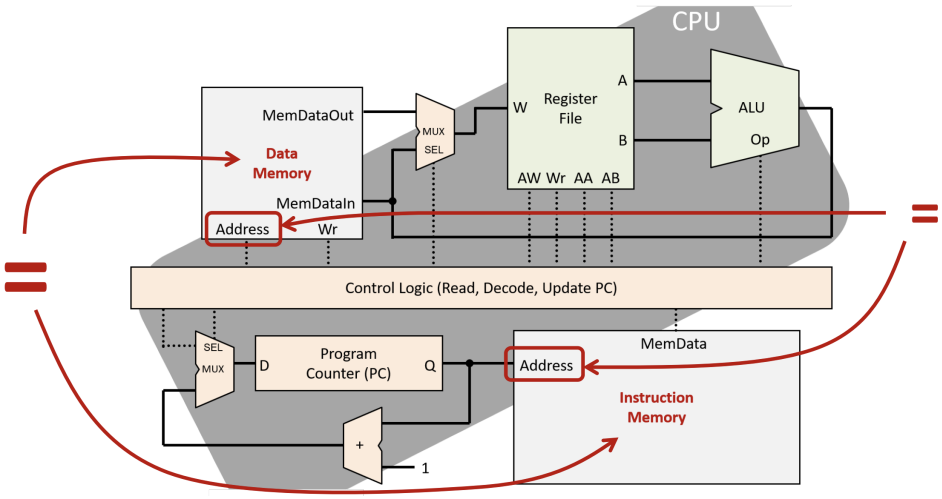
- **Arithmetic Logic Unit (ALU):** Performs arithmetic and logical operations. The inputs are provided by the register file, and the result is stored back into the registers or data memory.

- **Data Memory:** Stores data that can be written to or read from during program execution. It interacts with both the register file and the ALU for storing operands and results.



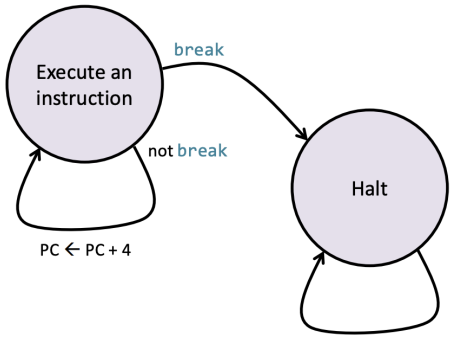
6.1.1 Unified Memory

In the image above, we see that the data memory and instruction memory are separate. However, a choice that is often made is to have a unified memory.



6.1.2 Single-Cycle Processor

At the end, like most circuits, a processor is just another Finite State Machine. The simplified state diagram of a single-cycle processor would like this:



Execute an instruction, move to the next, repeat.
This simplified view doesn't reflect actual CPU design. In reality, instructions take different amounts of time due to complexity and **Propagation Time**—the delay in signal travel through the processor.

6.2 Propagation Time

Remember the difference **(this is absolutely critical to understand the rest of the course)** between combinational circuits and sequential circuits.

As the name suggests, sequential circuits are built like a *sequence*(mnemotechnic), meaning the current output depends on both the current input and the previous state.

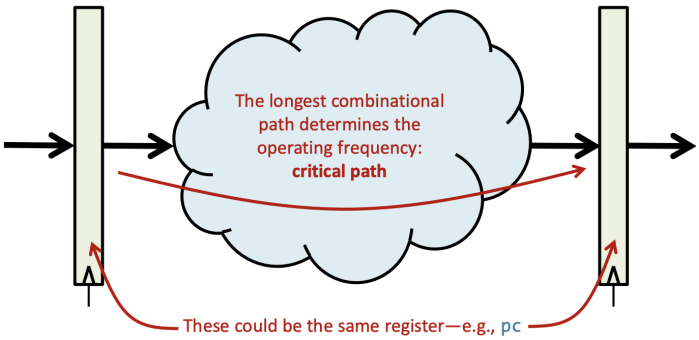
While combinational circuits, don't have a memory, they just take an input and give out an output.

The main thing to understand here is that, for our circuits to function as intended, the **propagation time** must allow the combinational circuits to complete before the next clock cycle (otherwise, it would lead to *obvious bugs*).

This implies that we need to observe the **longest combinational path** and account for it when designing our circuits.

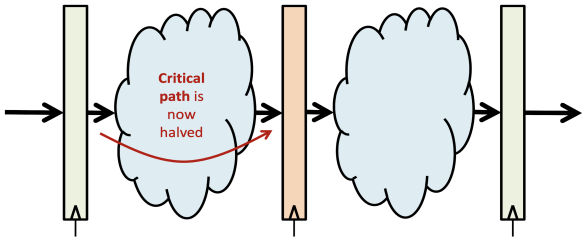
While this is the *efficient approach*, one could, in theory, design a propagation time that is longer than the longest path. However, this would result in a *waste of both time and resources.*

Remember, lower propagation time means higher clock frequency, which means faster processing.



6.2.1 Increasing the Frequency

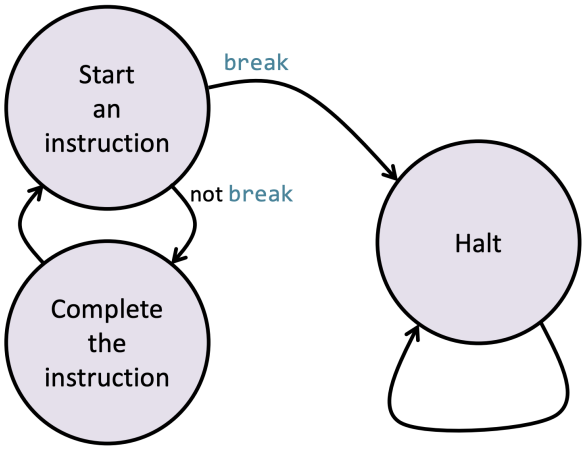
To increase the frequency, we need to decrease the propagation time. This can be achieved by breaking down the combinational path into smaller parts.
For example, consider the ‘lw‘ instruction. This requires adding the offset to the base address (which involves addition, not completely trivial), and then reading the data from memory. This process can be broken down into two stages: first, the addition, and then the memory read.



By doing this, we can operate at twice the *""speed""*(we'll see why this is wrong in a moment).

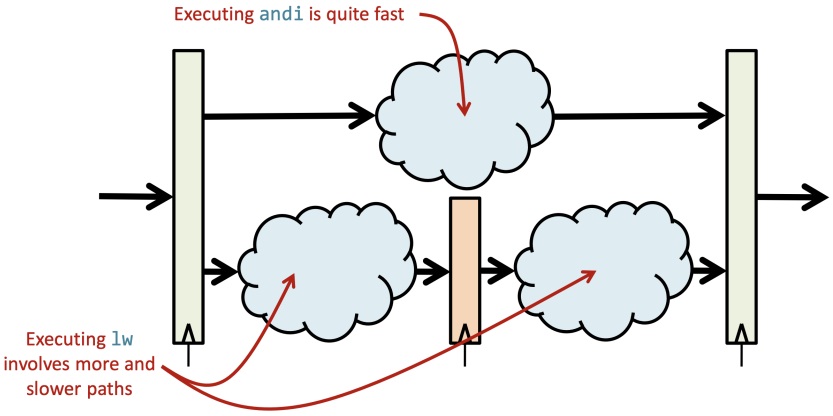
6.2.2 Two-Cycle Processor

However, what we quickly realize is that this approach doesn't result in a real performance gain. While the processor runs at twice the frequency, it also takes twice as long to complete the instruction, leading to no overall improvement. *Historically, Intel often used this strategy to persuade uninformed consumers that their processors were getting faster.*



6.2.3 Not All Paths Are Born Equal

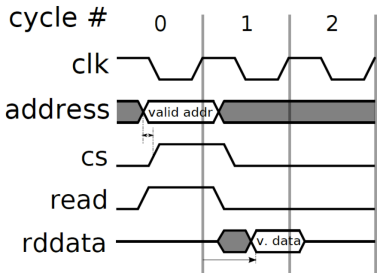
The reason we're discussing this is that not all paths are equal. Some instructions are faster to compute than others. For example, the `andi` instruction is much faster than the `lw` instruction.

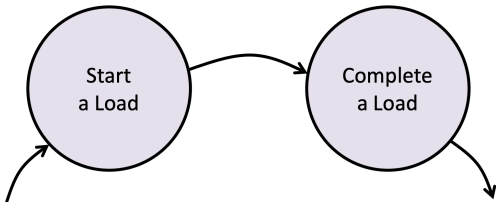


6.2.4 Asynchronous/Synchronous Memories

Another reason why breaking down the combinational path could be beneficial is that certain memories are **Synchronous**, meaning they only read data from a valid memory address on the rising edge of the clock cycle.
On the other hand, **Asynchronous** memories read data as soon as a valid memory address is available, without waiting for the clock cycle.

So, for **Synchronous** memories, breaking down combinational paths into smaller segments allows us to increase the clock frequency, making memory updates faster.





6.3 Multicycle Processor

Now let's try to construct a more convincing representation for our processor.
The processor operates in two cycles: a faster path for simple instructions and a slower path for more complex ones.

- **Fetch1/Fetch2**:

  Simple: Uses only Fetch1 for single-word instructions.

  Complex: Uses Fetch2 to fetch additional data when needed (e.g., multi-word instructions).

- **Decode**:

  Simple: Quick decoding with fewer control signals.

  Complex: More control signals and operands, requiring extra decoding time(and extra Optimization could be to introduce two Decoding stages for simple/complex instructions).

- **Execute**:

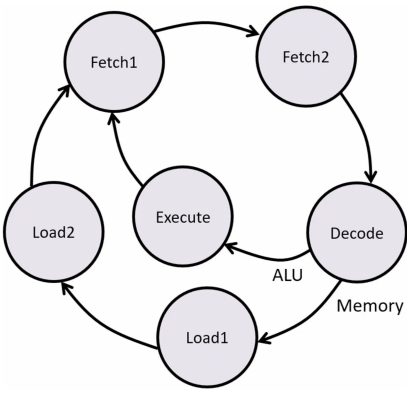
  Simple: Fast ALU operations like additions.

  Complex: Involves branches or complex ALU operations.

- **Load1/Load2**:

  Simple: Skips Load stages if no memory access.

  Complex: Memory operations use Load1 and Load2 to fetch and process data.



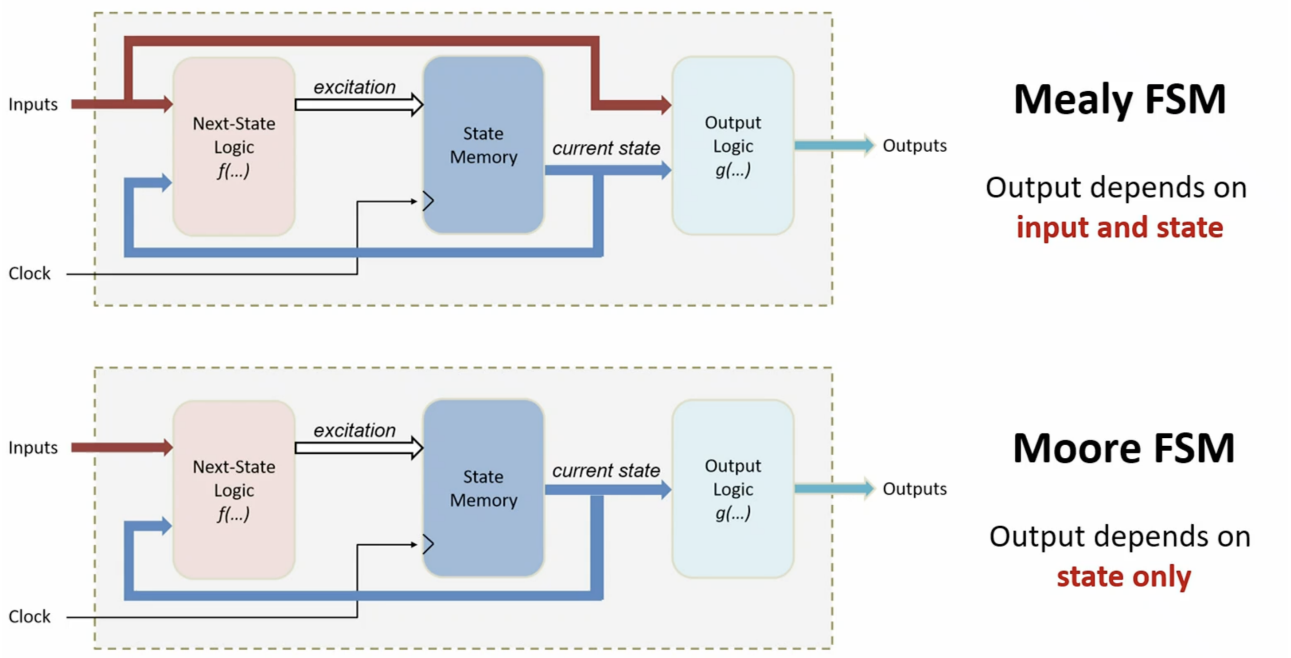
While this is an efficient design, it is not unique. The two things to keep in mind when designing a processor are:

not to have too many stages, *meaning that having an excessive number of stages could increase the complexity and latency of the processor (this we will see later in the course).*

to have paths as balanced as possible, *meaning that the duration of each stage should be similar to avoid bottlenecks that would slow down the overall process. The more balance we have the more we can profit from fast cases.*

6.4 Mealy or Moore?

Personal Remark (mnemotechnic)
Moore - Output Only depends on state (double O like in Moore),
Mealy - Output depends on state and input



It is generally preferable to use Moore state machines because their outputs depend only on the current state, making them simpler to design, debug, and predict, whereas Mealy machines depend on both state and input, introducing complexity and potential glitches. So unless the speicifcations requires us to do otherwise, we'll generally tend to represent our state machines as Moore machines.

6.5 Processor - Building the Circuit

In this part, we will be incrementally adding the components needed to build our processor circuit.
For now, we've added two components to our CPU:

Controller: This component, although empty for now, will eventually manage the flow of data and sequence of operations within the CPU. It will control how data moves and instructions are executed.

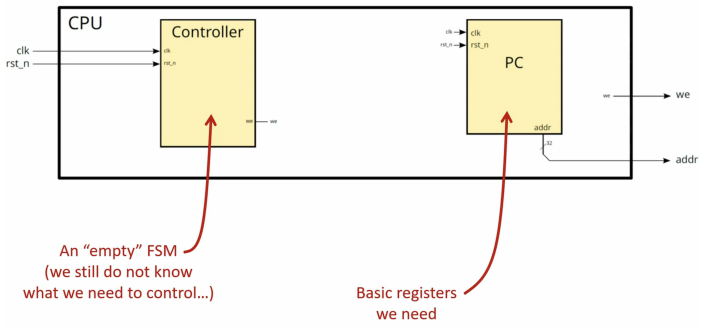
PC (Program Counter): The PC holds the address of the next instruction to be executed from the instruction memory. It increments after each instruction fetch or is updated based on control logic.

Inputs

- `clk`: The clock input ensures the program counter updates synchronously with the system clock.
- `rst_n`: An active-low reset signal that resets the program counter to a default value when low (0).
- `en`: The enable signal controls whether the PC updates its value (controlled by the Controller's FSM).

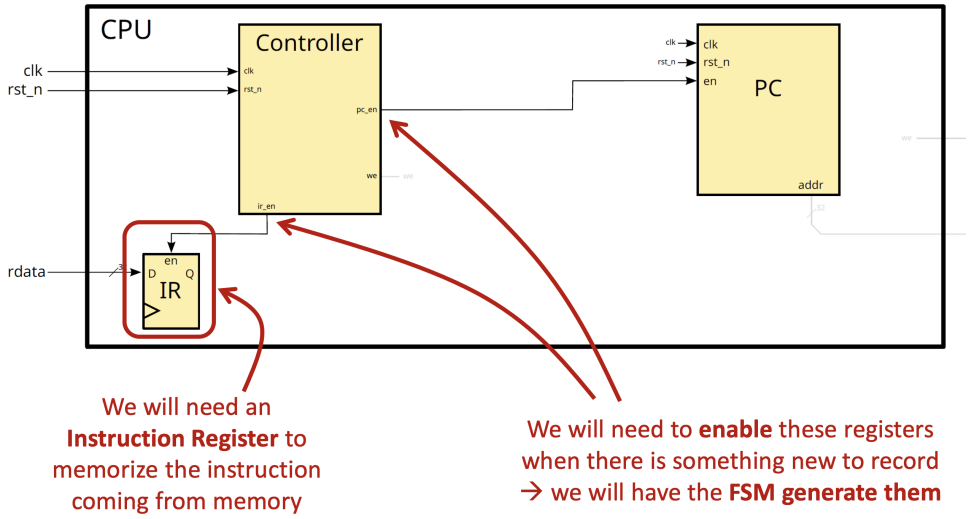
Outputs

- `addr`: The address output representing the next instruction to be fetched from memory.



6.5.1 Adding the Instruction Register

Now we're adding an Instruction Register.



In this step, we introduce the Instruction Register (IR) to our CPU:
Instruction Register (IR): The IR is responsible for storing the instruction fetched from memory. It captures the instruction ready to be decoded and executed. The Controller generates enable signals to control when the PC and IR should update their contents.

PC (Program Counter)

Inputs

- `clk`: The clock input ensures the program counter updates synchronously with the system clock.
- `rst_n`: An active-low reset signal resets the program counter to a default value when low (0).
- `en`: The enable signal controls whether the PC updates its value. It is driven by the FSM in the Controller.

Outputs

- `addr`: The address output representing the next instruction to be fetched from memory.

IR (Instruction Register)

Inputs

- `clk`: Ensures the instruction register captures the instruction at the correct clock edge.
- `rst_n`: Active-low reset to reset the IR to its default state.
- `en`: The enable signal controls whether the IR updates its contents. It is activated when a new instruction is fetched from memory.
- `D`: The data input, which represents the instruction fetched from memory (`rdata`).

Outputs

- `Q`: The output of the instruction register, representing the stored instruction that will be decoded and executed.

Controller

Inputs

- `clk`: The clock signal to ensure synchronization with other components.
- `rst_n`: The active-low reset signal to reset the controller to its initial state.

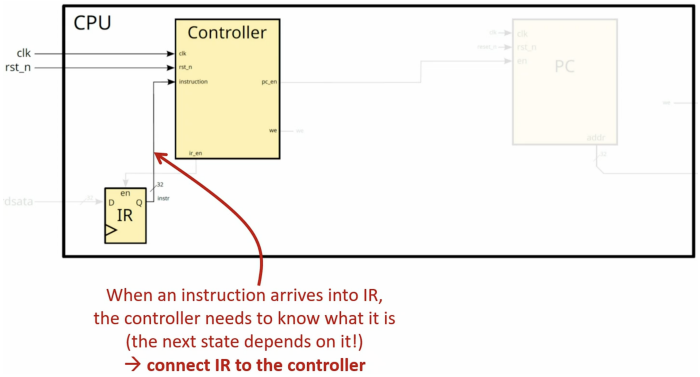
Outputs

- `pc_en`: The enable signal sent to the Program Counter (PC) to control when it should update its value.
- `ir_en`: The enable signal sent to the Instruction Register (IR) to control when it should store a new instruction.

6.5.2 Adding functionality

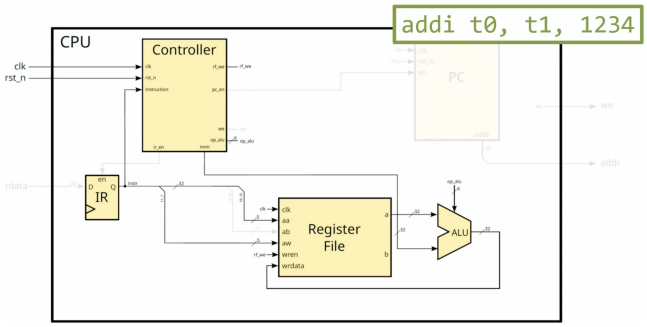
Once an instruction is fetched from memory and stored in the Instruction Register (IR), it is crucial for the Controller to receive this instruction. The Controller needs the instruction to determine the next sequence of operations, as the next state of the system is dependent on the instruction being executed.

The `Q` output of the IR, which holds the stored instruction, is fed directly to the Controller. This connection allows the Controller to decode the instruction and control the subsequent operations of the CPU.
Specifically, the Controller will enable or disable other components, such as the Program Counter (PC), based on the instruction being processed.



6.5.3 I-Type Instructions Need RF and ALU

I-Type instructions such as `addi t0, t1, 1234` require both the register file (RF) and the Arithmetic Logic Unit (ALU) for execution. The operation consists of an addition between a register value and an immediate value, and the result is stored back into a register.



ALU (Arithmetic Logic Unit)

Inputs

- `a`: First operand input from the register file (e.g., `t1`).
- `b`: Second operand input, typically the immediate value for I-type instructions (e.g., `1234`).
- `op_alu`: Control signal from the controller specifying the operation to perform (e.g., addition for the `addi` instruction).

Outputs

- `alu_out`: The result of the operation performed by the ALU (e.g., the sum of `t1` and the immediate value).

Register File

Inputs

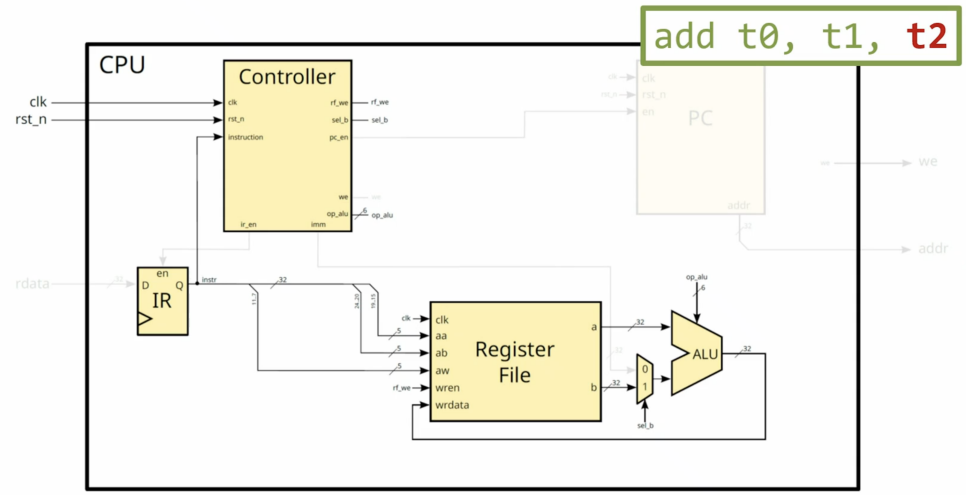
- `clk`: The clock input that ensures register updates are synchronous with the system clock.
- `aa`: The address of the first register (e.g., `t1`) from which data will be read.
- `ab`: The address of the second register (for other instruction types).
- `aw`: The address of the destination register (e.g., `t0`) to which the result will be written.
- `wren`: Write enable signal that allows data to be written into the destination register.
- `wrdata`: The data to be written into the destination register (e.g., the result from the ALU).

Outputs

- `a`: The data from the first register (e.g., the value stored in `t1`).
- `b`: The data from the second register (for other instruction types).

6.5.4 R-Type Instructions and Second Operand Selection

R-Type instructions, such as `add t0, t1, t2`, require two register operands and involve several components for execution. The instruction specifies two source registers (`t1` and `t2`) and a destination register (`t0`), with the second operand selected from the register file rather than using an immediate value. The multiplexer plays a key role in selecting the correct second operand based on the instruction type.



Register File

Inputs

- `clk`: The clock input that ensures register updates are synchronous with the system clock.
- `aa`: The address of the first register (e.g., `t1`) from which data will be read.
- `ab`: The address of the second register (e.g., `t2`) from which data will be read.
- `aw`: The address of the destination register (e.g., `t0`) where the result will be written.
- `wren`: Write enable signal that allows data to be written into the destination register.
- `wrdata`: Data to be written into the destination register (e.g., the result from the ALU).

Outputs

- `a`: The data from the first register (e.g., the value stored in `t1`).
- `b`: The data from the second register (e.g., the value stored in `t2`).

Multiplexer (sel_b)

Inputs

- `b`: The second operand, which can either be the register value (`t2`) or an immediate value, depending on the instruction type.
- `sel_b`: The select signal from the controller, determining whether the second operand is a register value (`t2`) or an immediate value.

Outputs

- `selected_b`: The selected operand output, which forwards either the register value (`t2`) or the immediate value to the ALU as the second operand.

ALU (Arithmetic Logic Unit)

Inputs

- `a`: First operand input from the register file (e.g., `t1`).
- `b`: Second operand input, selected by the multiplexer, from the register file (e.g., `t2`).
- `op_alu`: Control signal from the controller specifying the operation to perform (e.g., addition for the `add` instruction).

Outputs

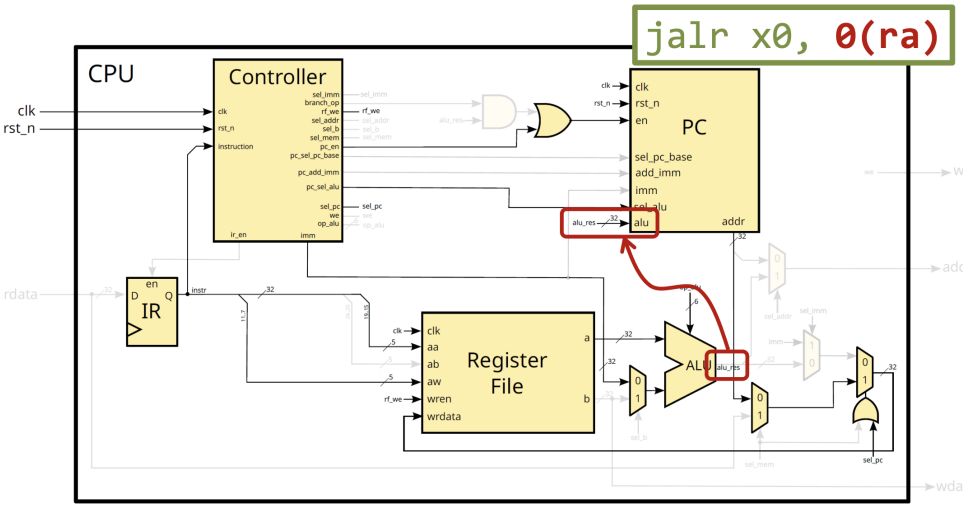
- `alu_out`: The result of the operation performed by the ALU (e.g., the sum of `t1` and `t2`), which is written back into the destination register.

6.5.5 And More, and More...

After these few additions, you basically get the point, we keep adding, block by block, the components we need for the full use of our processor, professor also goes pretty quickly over this (you'll also see the full implementation of this in LAB B.)
The rest of the additions being :

- U-Type Instructions Write an Immediate
- Load and Stores Produce a Memory Address
- Loads Write the Read Data into the RF
- Stores Send an Operand to Memory
- Branches Need to Write an Offset to the PC
- jal Needs to Store PC + 4 in the RF
- Jumps Need to Write an Address to the PC

The processor after all of this looks like this:



6.5.6 Guidelines for Writing Verilog

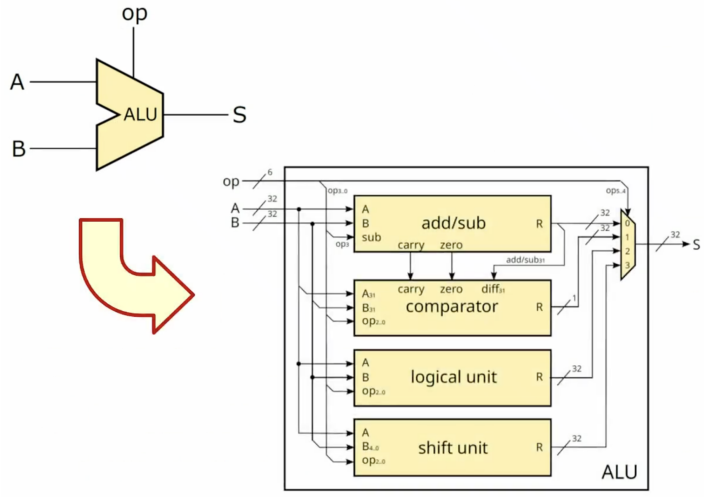
Before beginning to write Verilog code, it is crucial to follow certain guidelines to ensure clarity and correctness in your hardware design. Verilog and VHDL are Hardware Description Languages (HDLs) that require a clear and structured approach.
Anything that's complicated is a Module, anything that is trivial, we need to know if it's sequential or combinational.

- **Clarity and Preparation:**
  - Ensure that you have drawn a diagram, as demonstrated in previous examples.
  - Clearly distinguish between **combinational** and **sequential** blocks in your design.
- **Decomposition of Complex Sequential Blocks:**
  - Break down complex sequential blocks into simpler, well-defined elements. For instance, sequential blocks should primarily consist of simple registers (e.g., Instruction Register - IR).
  - Continue refining your hierarchical diagrams until all sequential blocks become trivial to implement.
- **Adopt a Hierarchical Approach:**
  - Use a hierarchical approach, similar to programming practices, and employ your diagrams to guide the creation of modules, such as the Program Counter (PC).

For example, for our processor, identifying that a register file is sequential while a PC is combinational is crucial before starting to write Verilog.

6.5.7 Detailing Complex Combinational Modules (ALU)

When designing complex combinational modules, it is essential to clearly define and break down each component to ensure accurate and efficient implementation. The following steps outline the process of detailing these modules:



- **ALU (Arithmetic Logic Unit) Overview:**
  - The ALU receives inputs A, B, and an operation code (op), and produces an output S.
  - It contains multiple submodules, such as add/subtract, comparator, logical unit, and shift unit.
- **Add/Subtract Unit:**
  - The add/subtract unit performs addition and subtraction operations based on the control signal *sub*.
  - It includes circuitry to handle carry and zero detection, essential for arithmetic operations.
- **Hierarchical Design:**
  - The ALU is composed hierarchically, where each submodule (e.g., add/subtract, comparator) performs specific functions and connects to the overall ALU structure.
  - Such a design allows for easier debugging, maintenance, and understanding of each module's role within the ALU.

6.5.8 Verilog - Sticking to Basic Paterns

When writing Verilog, it is essential to adhere to basic patterns for describing combinational and sequential logic. This section provides guidelines on structuring Verilog code efficiently.

Combinational Logic

Combinational logic blocks should be described using the `always @(*)` construct. This approach ensures that outputs are updated whenever the inputs change.

```
always @(*) begin
    if (a)
        y = \~b;
    else
        y = b;
end
```

Complex combinational blocks, such as the next state in a finite state machine (FSM), can also be described using this pattern.

Sequential Logic

Sequential logic blocks should be described using the `always @(posedge clk)` construct. This pattern is suitable for describing registers and counters.

```
always @(posedge clk) begin
    if (reset == 1)
        q <= 0;
    else if ((enable1 == 1) && (
        enable2 == 1))
        q <= d;
end
```

Use `posedge clk` to trigger updates on the rising clock edge.

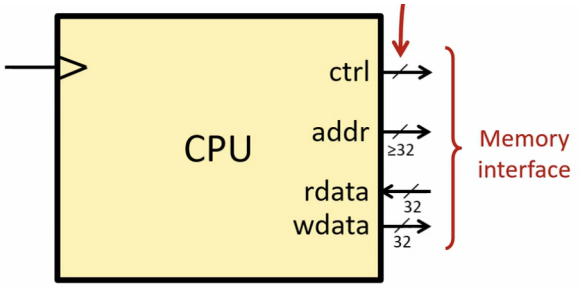
For detailed guidelines, refer to the Verilog guidelines provided in Moodle.

Chapter 7

Part II(b) - Processor, I/Os, and Exceptions W - 4.1 - 4.2

7.1 The CPU

The CPU is a very sequential component responsible for executing instructions in a controlled manner. The CPU interacts with the memory through a defined memory interface, which includes various control signals and data pathways.



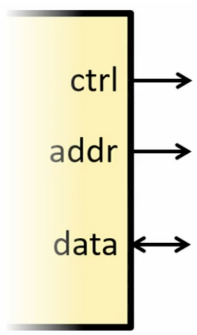
- **Control Signals (ctrl)**: These signals manage the behavior of memory access, indicating whether to read or write data.
- **Address (addr)**: Specifies the memory address where the CPU wants to read or write data. The width of the address bus is typically 32 bits or more.
- **Read Data (rdata)**: A 32-bit pathway through which the CPU receives data from the memory.
- **Write Data (wdata)**: A 32-bit pathway through which the CPU sends data to be stored in memory.

The memory interface is also controlled by two important signals:

- **Circuit Enable (CE)**: Validates the address, indicating that the address provided is active and the operation should proceed.
- **Write Enable (WE)**: Indicates that the current access is a store operation, allowing data to be written into memory.

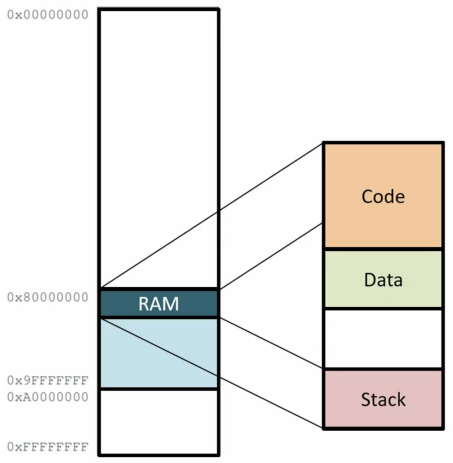
From now on, the clock signal, which drives the sequential behavior, may be omitted for simplicity. This interface design allows for a clear and structured method of communication between the CPU and memory, ensuring reliable execution of instructions and data management.

Some processors, instead of having two seperate buses, have a single data bus, that can be used for both reading and writing. This is known as a `bidirectional data bus`. This means, this kind of system uses a tri-state buffer to control the direction of the data flow.

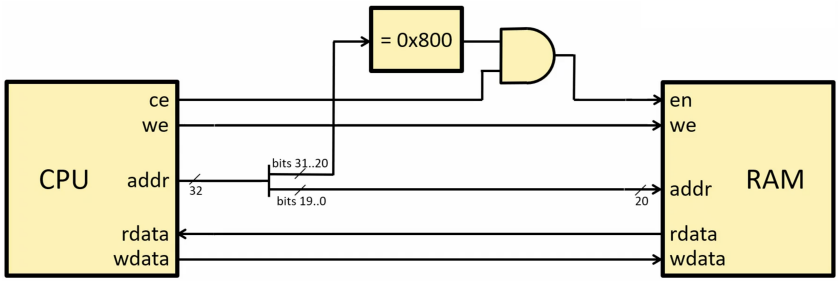


7.2 Physical Memory Map

To connect the CPU to memory, we need to define a `physical memory map`*. As we had 32 bits of address, we can address* 2^{32} *bytes of memory (aprox. 4GB of memory at least in a CPU).*



7.2.1 Connecting CPU and Memory



Address Mapping: The upper segment of the address (bits 31..20) is compared with the hexadecimal value `0x800` to determine whether the target memory location resides within the RAM range. If the comparison (XNOR gate) yields a match, the `en` (enable) signal for the RAM is activated, allowing the subsequent read or write operation.

Control Signals: Two primary control signals are involved in coordinating memory operations:

- `ce` (chip enable): Indicates whether the CPU is ready to perform an operation (read or write) on a specific memory chip.
- `we` (write enable): Indicates whether the operation is a write operation, allowing data to be written into RAM.

The lower segment of the address (bits 19..0) is passed directly to the RAM as the 20-bit address input, specifying the exact memory location within the enabled RAM region.

This mapping allows the CPU to access specific regions of RAM by comparing higher address bits with predefined values and appropriately enabling or disabling memory segments.

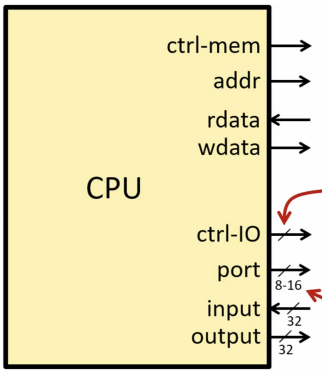
7.3 Input/Output (I/O) Devices

Input/Output (I/O) devices serve as crucial interfaces for various types of communication between a computer system and the external environment. I/O device data rates vary widely based on their type and purpose. Low-bandwidth devices like keyboards and mice handle simple inputs, while modern storage and networking technologies, such as PCIe 4.0 and USB 4.0, achieve much higher data rates for demanding tasks.

Type	**Peripheral**	**Data Rate**
Human Interaction	Keyboard	$\sim$ kbps
Human Interaction	Mouse	$\sim$ kbps
Generic	Serial Port (RS-232)	115.2 kbps (max)
Generic	Parallel Port (LPT)	150 kbps
Generic	USB 4.0	20-40 Gbps
Generic	Bluetooth 5.0	2 Mbps
Generic	PCIe 4.0	16 Gbps per lane
Storage	SATA III (HDD/SSD)	6.0 Gbps
Storage	NVMe (PCIe 4.0)	64 Gbps (4-lane)
Networking	Ethernet (10BASE-T)	10 Mbps
Networking	10 Gigabit Ethernet (10GBASE-T)	10 Gbps
Networking	Wi-Fi 6 (802.11ax)	Up to 9.6 Gbps
Displays	VGA (analog video)	0.6-1.5 Gbps (approx.)
Displays	HDMI 2.1	48 Gbps
Optical Discs	CD-ROM	150 KB/s (1x) - 7.68 MB/s (52x)
Optical Discs	DVD-ROM	1.32 MB/s (1x) - 21.1 MB/s (16x)
Optical Discs	Blu-ray	4.5 MB/s (1x) - 54 MB/s (12x)

7.3.1 Accessing I/Os: Port-Mapped I/O (PMIO)

Port-Mapped I/O (PMIO) is a technique used to create a separate interface for Input/Output (I/O) operations, which is distinct from the memory interface. This method allows the CPU to access peripheral devices using dedicated I/O ports. In PMIO, specific control signals and new instructions are introduced to facilitate I/O operations.
Similar to the Memory Interface.



- **New Interface**: The CPU has a control interface for both memory (`ctrl-mem`) and I/O (`ctrl-IO`), along with an address bus (`addr`), read data bus (`rdata`), and write data bus (`wdata`).
- **Port Numbering and Control Signals**: The I/O ports are addressed separately using a dedicated `port` line (typically 8-16 bits wide), and additional control signals are introduced:
  - `CE` (Circuit Enable): Indicates that a valid port number is provided.
  - `OE` (Output Enable): Indicates that the I/O access is an output operation.
- **Data Buses**: The CPU communicates with I/O devices using dedicated `input` and `output` buses, which may not necessarily be 32 bits wide due to the limited number of peripheral devices.

Accessing I/Os: Memory Mapped I/O(MMIO)

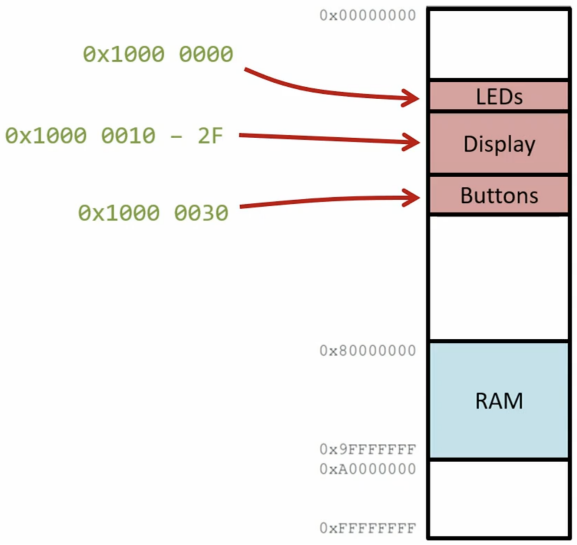
Memory-Mapped I/O is a technique that leverages the same address space for both memory and I/O operations. This allows devices to be accessed using standard memory instructions, eliminating the need for special hardware or dedicated I/O instructions.

Address Space Allocation: In this configuration, specific memory addresses are allocated for different peripherals:

- `0x1000 0000`: Base address for controlling LEDs.
- `0x1000 0010` to `0x1000 002F`: Address range for controlling the display.
- `0x1000 0030`: Base address for reading button states.

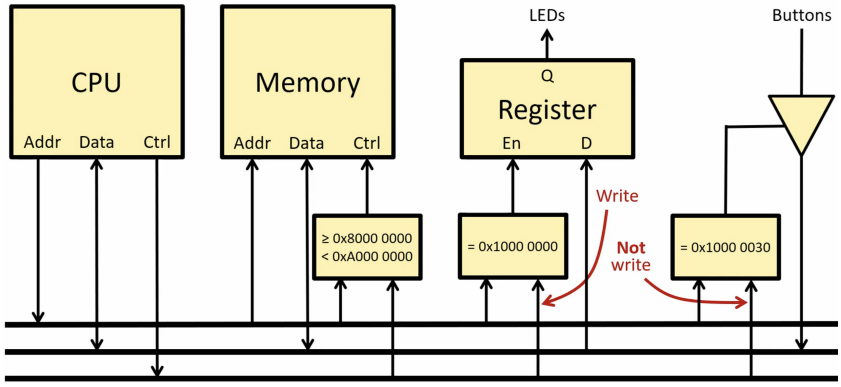
Standard Instructions: Since I/O devices are accessed as part of the memory space, standard load and store instructions can be used to interact with them. For example:

```
# Load upper immediate to set pointer to I/Os
lui   t0, 0x10000
# Write value in t1 to the address of LEDs
sw    t1, 0(t0)
# Read button states into t2
lw    t2, 0x30(t0)
```



7.3.2 Memory Mapped I/O (MMIO)

In computer architecture, Memory-Mapped I/O (MMIO) is a technique used to access input and output devices. Instead of using separate I/O instructions, devices are assigned specific memory addresses. The CPU interacts with these devices by reading or writing data to these memory locations as if they were normal memory addresses.



Components

- **CPU:** Acts as the central processing unit that interacts with memory and registers. It uses address, data, and control buses to communicate.
- **Memory:** Represents the conventional memory address space accessible by the CPU. Specific memory ranges (e.g., between $0x8000\ 0000$ and $0xA000\ 0000$) are reserved for memory-mapped devices.
- **Register:** The CPU communicates with the register through a dedicated memory-mapped address (e.g., $0x1000\ 0000$ for writing operations). This register drives outputs such as LEDs or accepts inputs from buttons.

Operation

The CPU interacts with memory and I/O devices through common buses. Depending on the address, data is either directed to regular memory or to an I/O device register. When the address matches a specific register (e.g., $0x1000\ 0000$ for a write operation), the corresponding action is triggered, such as updating an LED state. In contrast, accessing $0x1000\ 0030$ might perform a read operation, retrieving button states.

7.4 Example - A/D Converter

This example describes an Analog-to-Digital (A/D) Converter or (ADC) and its associated signals. The A/D Converter converts an analog input signal into a digital representation. The conversion process and signal behaviors are described below.

Signals

- `Start (START):` This input signal, when active a new conversion begins.
- `Data Valid (/DV):` This output signal indicates the validity of the data. When active, the output data bits (`D7-D0`) contain the converted digital value and are Valid.
- `Data (D7-D0):` The output data bits representing the last conversion result in digital form.

7.4.1 Bus Interface

The A/D Converter interacts with an 8-bit processor using a simple bus interface. This bus interface allows data exchange and control signals to flow between the A/D Converter and the processor. The following signals are used:

- `Address (A23--A0):` Output - Serves as the address bus to select a specific device or memory location.
- `Data (D7--D0):` Bi-directional - Represents the data bus used for data exchange between the processor and the A/D Converter.
- `Address Strobe (/AS):` Output - Indicates the presence of a **valid** address on the address bus during a memory access cycle.
- `Read/Write (R//W):` Output - Determines the direction of the data flow (read from or write to the A/D Converter).
- `Data Acknowledge (/DTACK):` Input - Signals the completion of a memory access by the A/D Converter when activated, indicating that the data is ready or has been latched.

The bus interface provides a simple mechanism for connecting the A/D Converter to the system bus, allowing the processor to initiate conversions and read the results.

7.4.2 Memory Mapping

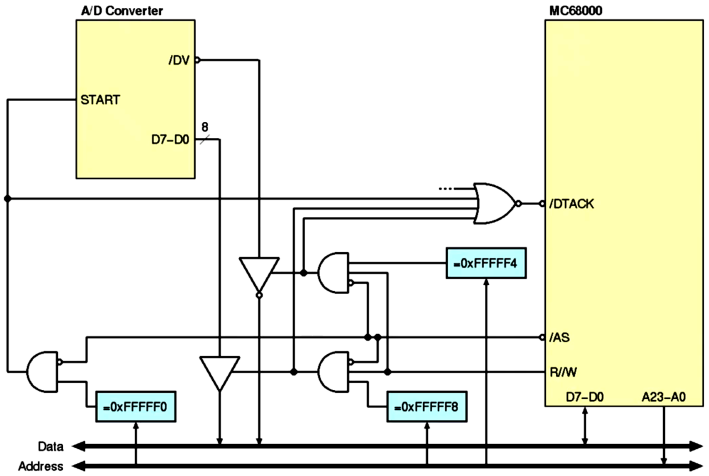
The A/D Converter is connected to the processor using a memory-mapped interface. Specific memory addresses are reserved for starting conversions, reading the data valid signal, and accessing the conversion result. The following address mapping is used:

- `0xFFFFF0:` Any access (read or write) to this address initiates a new conversion by the A/D Converter.
- `0xFFFFF4:` The processor reads the data valid signal from this address. Bit 0 of this location indicates whether the conversion result is ready.
- `0xFFFFF8:` The processor reads the conversion result from this address. The value stored here represents the digital output of the A/D Converter.

This memory-mapped interface simplifies the interaction between the processor and the A/D Converter by using standard read and write instructions to control the conversion process and retrieve the results.

7.4.3 Assembling everything

To get to this point, it is highly advised to first draw a timing diagram of the expected signals, and then start drawing connections, also, be careful, the notation "/AS" for example, means that the signal is active low.

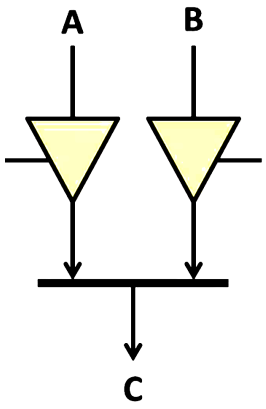


Software Implementation

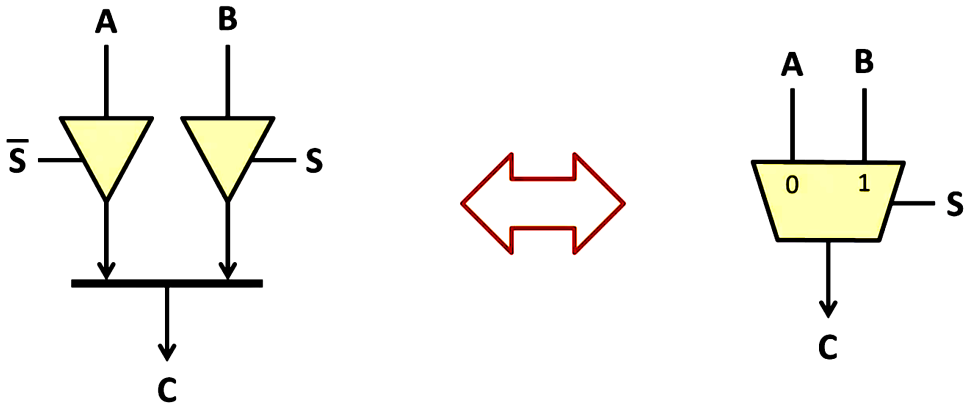
```
.section .data
START_ADDR:       .word 0xFFFFF0      # Address to initiate conversion
DATA_VALID_ADDR: .word 0xFFFFF4      # Address to check if data is valid
RESULT_ADDR:      .word 0xFFFFF8      # Address to read conversion result

.section .text
.globl _start

# Start of the main program
_start:
    # Initiate A/D Conversion
    lui t0, %hi(START_ADDR)        # Load upper 20 bits of START_ADDR
    lw t1, %lo(START_ADDR)(t0)     # Load lower 12 bits into t1
    # Write zero to start the conversion (writing to address 0xFFFFF0)
    sw zero, 0(t1)

wait_for_data:
    # Check if the data is valid
    lui t0, %hi(DATA_VALID_ADDR)  # Load upper 20 bits of DATA_VALID_ADDR
    lw t1, %lo(DATA_VALID_ADDR)(t0) # Load lower 12 bits into t1
    lw t2, 0(t1)                   # Load the data valid status into t2
    andi t2, t2, 0x1               # Mask bit 0 (check if data is ready)
    beq t2, zero, wait_for_data    # If bit 0 is zero, data is not valid, wait

    # Read the conversion result
    lui t0, %hi(RESULT_ADDR)       # Load upper 20 bits of RESULT_ADDR
    lw t1, %lo(RESULT_ADDR)(t0)    # Load lower 12 bits into t1
    lw t3, 0(t1)                   # Read conversion result into t3

    # End of program (infinite loop)
end:
    j end                          # Loop indefinitely
```

7.5 What do these tri-state buffers do?

Tri-state buffers are crucial components used to control data flow on a bus. They can exist in one of three states: high impedance (effectively disconnected), logic high, or logic low.
If not controlled properly, a tri-state buffer can cause bus contention, where multiple devices attempt to drive the bus simultaneously, leading to data corruption or damage.



In essence, a tri-state buffer acts like a decentralized multiplexer, functioning similarly to a multiplexer with a select line to manage data transmission.



7.5.1 A Classic UART

UART (Universal Asynchronous Receiver-Transmitter) is one of the simplest and most common communication peripherals, typically used to connect terminals to embedded devices.
Our UART employs a simple programmed I/O interface, consisting of four key registers:

- **Control register**: Configures the UART. Bit 7 must be set to 1 to enable the UART, while bits 2 to 0 determine the communication speed (e.g., `0b001` for 9600 baud).
- **Status register**: Provides the current status of the UART. Bit 1 indicates if data is available, and bit 0 signals if the UART is ready to transmit data.
- **Data input register**: Holds the received data available to the processor.
- **Data output register**: Contains data placed by the processor for transmission.

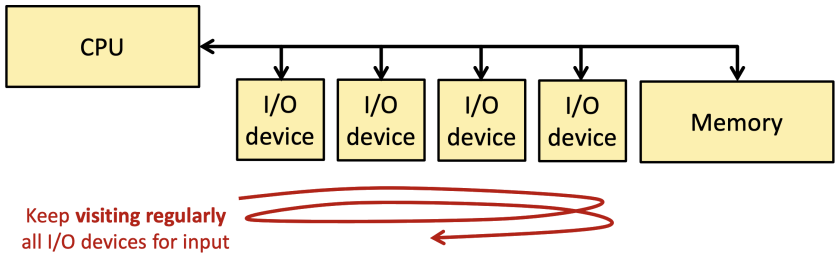
```
# Constants
UART_CTRL_ADDR      = 0x10000000  # UART control register address
UART_ENABLE_BIT     = 0x80        # Enable bit (bit 7)
UART_SPEED_9600     = 0x01        # Speed setting for 9600 baud (4 bits, [3:0])
UART_STATUS_ADDR    = 0x10000004  # UART status register address
TX_READY_BIT        = 0x01        # Transmitter ready bit (bit 0)
UART_DATAIN_ADDR    = 0x10000008  # UART data input (receive) register address
UART_DATAOUT_ADDR   = 0x10000008  # UART data output (send) register address

# Send a string using UART
send_string:
    li t0, UART_CTRL_ADDR         # Load UART control register address into t0
    li t1, UART_STATUS_ADDR       # Load UART status register address into t1
    li t2, UART_DATAOUT_ADDR      # Load UART data output register address into t2
    li t3, UART_ENABLE_BIT        # Load enable bit (0x80) into t3
    li t4, UART_SPEED_9600        # Load speed setting (0x01) into t4
    or t3, t3, t4                 # Combine enable and speed bits into t3
    sw t3, 0(t0)                  # Configure UART by storing combined value into
        control register

next_char:
    lb t5, 0(a0)                  # Load the current byte of the string into t5
    beqz t5, finish               # If the byte is zero (null terminator), jump to
        finish

check_tx_ready:
    lw t6, 0(t1)                  # Load the value of the UART status register into
        t6
    andi t6, t6, TX_READY_BIT     # Check if the TX ready bit is set
    beqz t6, check_tx_ready       # If not ready, loop back to check again

    sw t5, 4(t2)                  # Store the character in UART data register
    addi a0, a0, 1                # Move to the next character in the string
    j next_char                   # Jump to send the next character

finish:
    ret                           # Return from function
```

Chapter 8

Part II(c) - Interrupts W - 5.1 - 5.2

8.1 I/O Polling

I/O Polling is a method used by the CPU to check if any peripheral devices, such as a keyboard or network interface, have data to provide. The CPU continuously monitors each connected I/O device at regular intervals to see if they need attention.



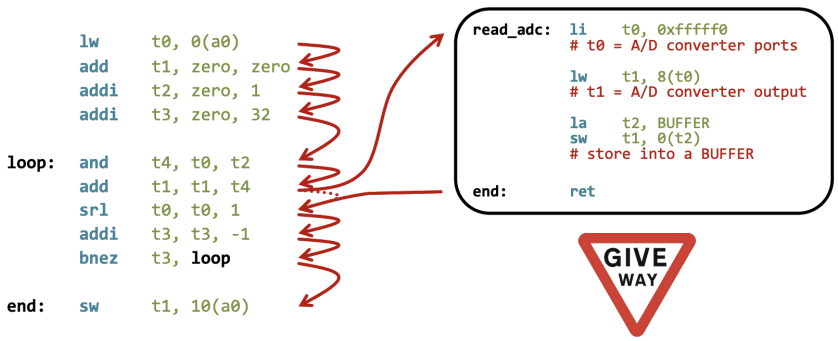
How It Works: The CPU keeps visiting each I/O device in a loop to check for input or status changes. This is known as "polling" the devices.

Drawbacks: This approach can be very resource-intensive. If a device operates at high speed and requires immediate handling, the CPU must check it frequently, which can consume significant processing time.

8.2 I/O Interrupts

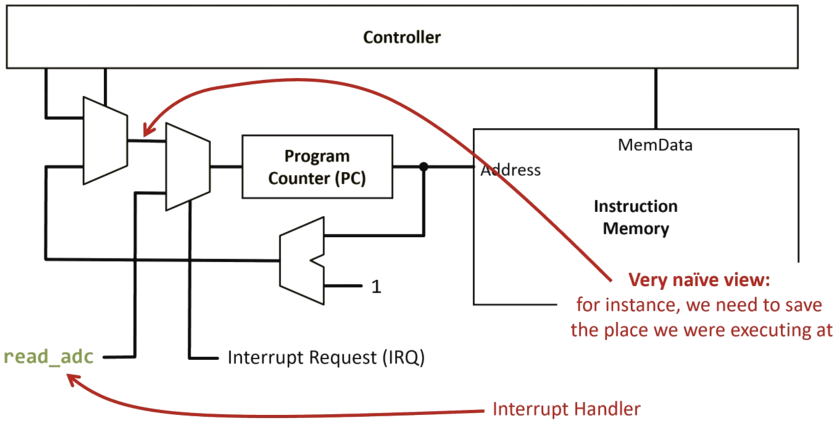
Instead of continuously checking the status of peripherals, it is more efficient to have them request attention when needed. This approach minimizes CPU usage by eliminating the need for constant polling.

- **Polling Method:** The CPU checks the status of a peripheral device by repeatedly executing a loop to monitor the peripheral register. This approach requires continuous CPU attention, which can be inefficient in systems with multiple peripherals.
- **Interrupt Method:** In an interrupt-driven approach, peripherals alert the CPU only when they need attention. The CPU executes an interrupt service routine (ISR) to handle the request. This method allows the CPU to focus on other tasks until interrupted, improving efficiency.



8.2.1 The Basic Concept of I/O Interrupts

I/O interrupts provide a mechanism for a controller to handle external requests efficiently by temporarily diverting program execution.
This is not the actual implementation, but basic concept for you to help you understand what we're aiming for.



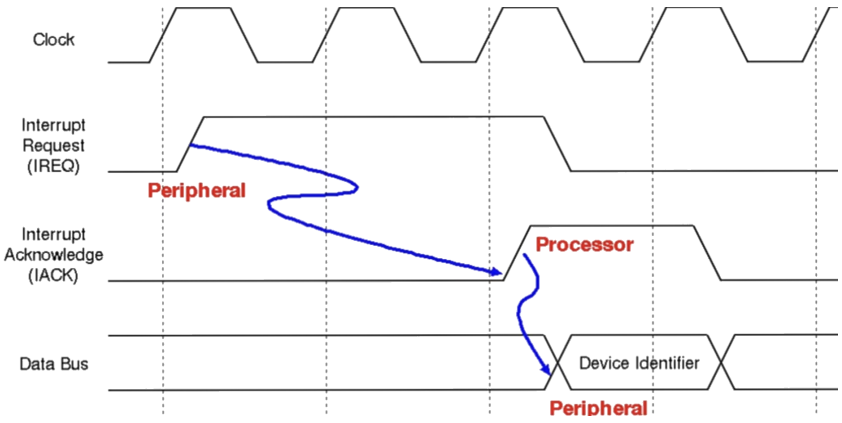
- **Interrupt Request (IRQ):** An interrupt signal is triggered, typically from an I/O device, to request immediate attention from the controller.
- **Program Counter (PC) Preservation:** The current value of the Program Counter (PC), which holds the address of the next instruction, is saved to allow resumption of normal execution after the interrupt is handled.
- **Interrupt Service Routine (ISR):** The controller redirects the PC to the address of the interrupt handler function, denoted here as `read_adc`. This function processes the interrupt by executing specific instructions related to the I/O request.
- **Instruction Memory Access:** The `Instruction Memory` is accessed to fetch instructions at the new PC address, executing the ISR for the interrupt.
- **Resuming Program Execution:** Once the interrupt has been serviced, the controller restores the saved PC value, allowing the program to continue from the point it was interrupted.

Considerations for I/O Interrupt Handling
When managing multiple I/O interrupts, several issues must be addressed:

- **Identifying the Source of the Interrupt:** In systems with multiple peripherals, it is essential to determine which device triggered the interrupt. This can be achieved through:
  - *Polling:* After an interrupt, the software checks each peripheral sequentially.
  - *Identification by the Peripheral:* The I/O peripheral itself sends an identification signal.
- **Handling Different Priorities:** Some interrupts may require immediate attention, while others can be delayed. Assigning priorities ensures critical interrupts are serviced promptly, while less urgent ones may wait.
- **Impact on Current Execution:** The system must decide whether to allow the current instruction(s) to complete before handling the interrupt or to pause immediately. This decision impacts program flow and execution timing.

8.2.2 Interrupt Cycle Description

The interrupt cycle is a sequence where a peripheral device signals an interrupt to the processor, which responds by acknowledging the interrupt and reading the device identifier from the data bus. The following signals are involved in this process:



- **Clock Signal**: The clock signal provides the timing for synchronization between the processor and peripherals.
- **Interrupt Request (IREQ)**: A peripheral asserts this signal to request service from the processor. When IREQ goes high, the processor detects an interrupt request.
- **Interrupt Acknowledge (IACK)**: In response to IREQ, the processor sends an acknowledgment signal (IACK) to the peripheral. This signal indicates that the processor is ready to handle the interrupt.
- **Data Bus**: After the IACK signal is asserted, the peripheral places its device identifier on the data bus, allowing the processor to identify the source of the interrupt.

The interrupt cycle proceeds as follows:

1. The peripheral raises the **IREQ** line to signal the interrupt request.
2. The processor detects the interrupt and, after some clock cycles, responds by asserting the **IACK** line.
3. The peripheral then places its **Device Identifier** on the data bus.
4. The processor reads the device identifier to determine the source of the interrupt and proceeds with the appropriate interrupt service routine.

This cycle ensures that the processor can handle asynchronous requests from peripheral devices in an organized and timely manner.

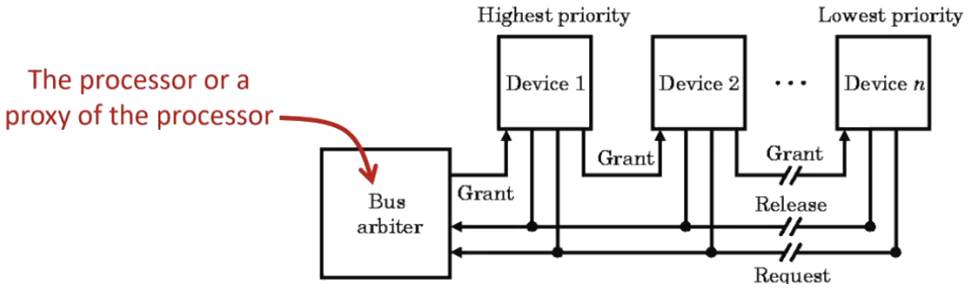
8.2.3 I/O Interrupt Priorities: Daisy Chain Arbitration

Daisy Chain Arbitration is a basic method used to manage I/O interrupt priorities. The process operates as follows:

- **Request Placement:** Any device can initiate a request to access the bus, indicated by signals such as `IREQ` (Interrupt Request).
- **Acknowledgment Line:** An acknowledgment signal, referred to as `IACK` or `Grant`, is sequentially passed from one device to the next.
- **Signal Interception:** The first device that requires access intercepts the acknowledgment signal, preventing it from being passed to devices further down the chain.

This method, while simple and easy to implement, has some limitations:

- **Slow Performance:** Due to the sequential nature of signal passing, response times can be slower as the chain length increases.
- **Fixed Priorities:** Devices closer to the bus arbiter have higher priority by design, leading to a rigid priority structure.



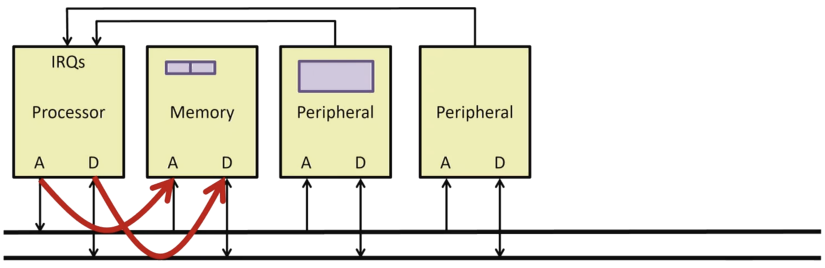
In this setup, the bus arbiter, which acts as the processor or a proxy for the processor, grants access in a priority chain from the highest-priority device to the lowest. This method is suitable for systems where simplicity is valued over flexibility and speed.

8.3 Direct Memory Access (DMA)

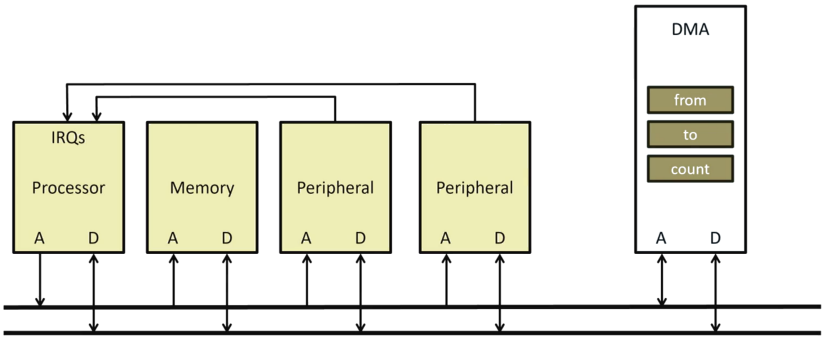
Direct Memory Access (*DMA*) is an efficient mechanism designed to offload the processor from managing repetitive and resource-intensive data transfers. Key considerations include:

- **Interrupts Efficiency:** *Interrupts* save the processor from continuously polling Input/Output (I/O) devices, allowing it to focus on computation.
- **Large Data Transfers:** Despite the use of interrupts, the processor may still spend considerable time transferring large chunks of data to and from high-throughput peripherals (e.g., disks, networks).
- **Solution:** A dedicated peripheral, known as the *DMA Controller*, is introduced. This controller autonomously handles data transfers between memory and peripherals (read/write operations), freeing the processor to focus on more critical tasks.

DMA significantly enhances system performance by reducing processor overhead during data transfer operations.

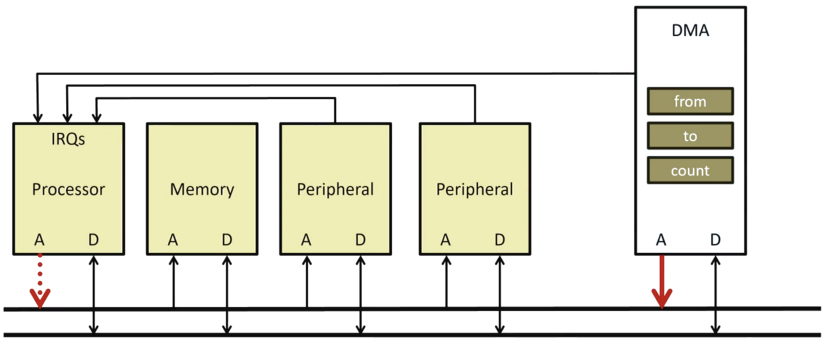


The diagram above illustrates a key inefficiency: *using the processor—a complex and expensive machine—to handle simple data transfer operations.* This inefficiency forms the basis for introducing DMA.



When initiating a data transfer, the **CPU** communicates with the **DMA Controller** to start the operation with a specific peripheral. The **DMA Controller** then handles communication with the **I/O device**, transferring the data to or from memory.

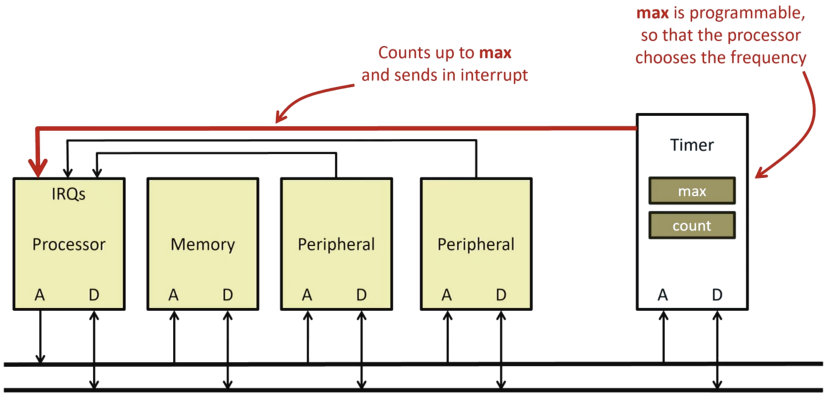
However, this introduces a new challenge. Previously, the *CPU* was the sole **master of the BUS**. With the **DMA Controller** capable of two-way communication with the BUS (on its A input), it also becomes a master of the BUS. This requires a mechanism to manage BUS access between the processor and the DMA Controller.



This is achieved using a **Tri-State Buffer**. However, during the data transfer, the processor is temporarily *disconnected from the BUS*, meaning it cannot track the progress of the transfer. Once the transfer is complete, an **interrupt** is required to notify the processor, ensuring it can resume operations with the updated data.

8.3.1 Timer and Interrupt Mechanism

A **timer** is a critical hardware component used to manage periodic tasks in embedded systems. The timer operates by incrementing a `count` register until it reaches a programmable `max` value, at which point it generates an **interrupt request (IRQ)**. The key features of this mechanism are outlined below:



- **Programmable Frequency:** The `max` value is configurable, allowing the processor to adjust the interrupt frequency based on system requirements.
- **Interrupt Handling:** Upon reaching the `max` value, the timer sends an IRQ signal to the processor. This allows the processor to execute specific tasks at regular intervals.
- **System Integration:** The timer interacts with the processor, memory, and peripherals via the system bus, ensuring synchronized operation.
- **Task Management:** Without a timer, it would be impossible for a processor to manage multiple tasks simultaneously, as there would be no mechanism to divide time between different operations. The timer enables multitasking by providing precise time slicing for task scheduling.

This mechanism is essential for time-sensitive operations such as task scheduling, event triggering, and real-time control in embedded systems, enabling efficient multitasking and coordination between components.

Chapter 9

Part II(d) - Processor, I/Os, and Exceptions W - 5.1

9.1 Exceptions, Interrupts, Faults, Traps, and Checks

Control Flow Under normal circumstances, the *control flow*—the sequence of instructions executed by a program—is fully determined by the programmer. This includes the use of jumps, branches, and procedure calls.

Exceptions Exceptions represent a deviation from the normal control flow. They are triggered by **special conditions** that are not explicitly defined in the program. When an exception occurs, the control flow changes unexpectedly, and the program must respond accordingly.

Exception Handlers To manage exceptions, *exception handlers* are invoked. These are specialized functions designed to take appropriate actions when an exception arises. An example of this is **I/O interrupts**, which signal specific events related to input/output operations.

Naming Conventions The terminology for exceptions and related events varies widely across systems. For clarity, we adopt the following convention based on RISC-V and the COD:

- **Exceptions:** A general term encompassing all control flow deviations.
- **Interrupts:** A specific type of exception generated outside the processor.

Thus far, interrupts are the only form of exception encountered.

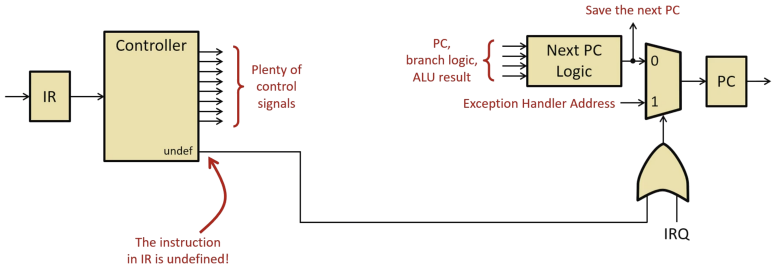
9.1.1 Undefined Instruction

Undefined instructions are instructions that the controller does not recognize, as they do not correspond to any valid operation in the Instruction Register (IR). These scenarios require special handling to ensure system stability and proper exception processing.

- **Detection:** When an undefined instruction is detected in the IR, the controller generates a signal (`undef`) indicating the presence of an invalid operation.
- **Exception Handling:** The Program Counter (PC) is updated to the address of the Exception Handler to manage the undefined instruction. This involves:

  - Saving the current PC for potential recovery.
  - Redirecting the control flow to the exception handler's address using multiplexer logic.

- **Control Logic:** The system leverages the Next PC Logic to determine whether the next instruction comes from the regular PC logic or the exception handler, based on the `undef` signal or an external interrupt (IRQ).



- **Synchronous Nature:** These exceptions occur at a specific point in the program, precisely where the undefined instruction resides. This predictable behavior ensures that if the program is re-executed from the same initial state, the exception will occur at the exact same point, making debugging more straightforward.

- **Immediate Handling:** Serving the exception before executing the next instruction allows advanced features, such as efficient error recovery and the potential to extend system capabilities. This mechanism ensures that undefined

instructions do not disrupt the execution flow and are handled systematically, enabling robust error recovery and system stability.

9.1.2 Optional `fadd.s` Instruction

Suppose we want to include a floating-point addition instruction, denoted as:

```
1 fadd.s rd, rs1, rs2
```

- Some processors might include a specialized ALU to support this instruction, whereas **cheaper processors do not**.

- For processors that lack support for this instruction, its execution would trigger an *undefined instruction exception*, which invokes a handler.

- The handler can **emulate** the behavior of the `fadd.s` instruction, ensuring compatibility across processors.

9.1.3 Outline of an Undefined Instruction Handler

To handle an undefined instruction, such as `fadd.s`, the following steps wouls be executed:

Save all registers on the stack that the handler or its callees might modify.

- Note: Standard calling conventions do not apply.

Retrieve the problematic instruction:

- If the program counter (PC) is saved, load the instruction from the corresponding address.

Decode the instruction in software and identify it as `fadd.s`.

Read the source registers (operands) and either:

- Call a library function, or
- Implement the floating-point addition in software.

Store the result in the destination register.

Update the program counter (PC) to point to the next instruction.

Jump to the updated PC to resume execution.

9.2 Exceptions and Interrupts

Exceptions, interrupts, and related mechanisms handle critical events during execution. Key use cases include:

I/O Requests: Processing data or new inputs.

Timer Interrupts: Handling time-based events.

Undefined Instructions: E.g., unsupported floating-point operations.

Arithmetic Faults: Errors like division by zero.

Memory Violations: Unauthorized access to restricted memory.

Debugging: Breakpoints and execution control.

Hardware Failures: Malfunctions such as power loss.

9.2.1 A Possible Classification of Exceptions

Type	Synchronous?	Coerced?	Resume?
I/O request	Asynchronous	Coerced	Resume
Invoke OS	Synchronous	User requested	Resume
Trace instruction	Synchronous	User requested	Resume
Breakpoint	Synchronous	User requested	Resume
Page fault	Synchronous	Coerced	Resume
Misaligned access	Synchronous	Coerced	Resume
Memory protection violation	Synchronous	Coerced	Terminate
Bus error	Synchronous	Coerced	Terminate
Arithmetic fault	Synchronous	Coerced	Terminate
Undefined instruction	Synchronous	Coerced	Terminate
Hardware malfunction	Asynchronous	Coerced	Terminate
Power failure	Asynchronous	Coerced	Terminate

- **Synchronous?** Indicates whether the exception occurs as a direct result of the execution flow (synchronous) or independently of it (asynchronous).
- **Coerced?** Specifies whether the exception is forced by the system (coerced) or triggered by a user request.
- **Resume?** Denotes whether the system can continue executing after handling the exception (resume) or must terminate.

9.2.2 Watchpoint

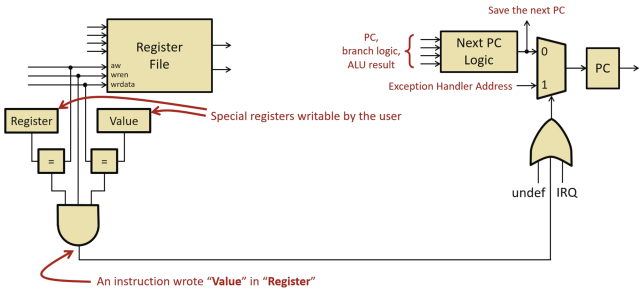
A **watchpoint** is a debugging feature in a processor architecture that monitors specific registers and triggers an exception when predefined conditions are met. This mechanism allows developers to track changes in critical registers and execute exception handlers when necessary.

Key Components

- **Register File:** Contains the set of registers, including special registers that can be configured by the user.
- **Watchpoint Logic:** Compares the monitored *register* and its *value* with user-defined conditions.
- **Next PC Logic:** Determines the next program counter (PC) value based on standard flow or exception handling.

Functionality

- When an instruction writes a specific *value* into the monitored *register*, the watchpoint logic evaluates the predefined conditions.
- If the conditions are met, the system triggers an exception by redirecting the program counter to the *exception handler address*.
- Otherwise, normal program execution continues.



9.2.3 Raising Exceptions

To handle exceptions, the processor must be able to provide which exact exception occurred.

Saving the Address: The address of the current or next instruction must be stored in a dedicated register, typically the *Exception Program Counter (EPC)*. This prevents overwriting the `ra` register, which is used for ordinary returns. A special instruction is required to resume execution using the `EPC`.

Multiple Causes: Since exceptions can arise from various causes, a mechanism is needed to identify the source:

- A single handler address with a *cause register*.
- A vector of handlers, where each handler corresponds to a specific cause.

Nested Interrupts/Exceptions: While handling an exception, another may occur. To manage this:

- Disable exceptions that can be suppressed (e.g., interrupts).
- Avoid unpreventable exceptions (e.g., division by zero or undefined instructions).

9.2.4 Assessing the Position of an Exception

When an exception occurs, it is essential to determine the position in the instruction flow to decide the appropriate handling strategy. Some key considerations are:

- **Asynchronous Exceptions:**
  - Identify the *next instruction* to execute.
- **Synchronous Exceptions:**
  - Determine the *instruction that caused the fault.*
    - Restart from the faulting instruction if correction and retry are needed (e.g., invoking the operating system).
    - Restart from the next instruction if the functionality is implemented differently (e.g., emulating an undefined instruction in software).
- **Solutions:**
  - Reserve a dedicated ordinary register for the *Exception Program Counter (EPC)* (e.g., Nios II).
  - Store the EPC on the stack (e.g., x86).
  - Use special registers dedicated to exception handling with specific instructions to access them (e.g., MIPS, RISC-V).

9.2.5 Assessing the Cause of Exception

To handle exceptions, the processor must pass control to the appropriate handler. The mechanism for determining and invoking the correct handler can be categorized as follows:

Single Handler with Cause Register (e.g., RISC-V, MIPS):

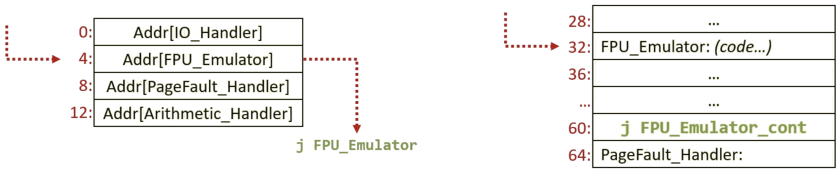
- The processor executes a jump to a fixed address.
- The handler performs the dispatching in software by reading a special *cause register*.

Vector of Handler Addresses (e.g., RISC-V, 68k):

- The processor executes a jump to `mem[Exception Vector Address + (4 × Exception Number)]`.

Vector of Handlers (e.g., PA-RISC 2.0, SPARC):

- The processor directly jumps to `Exception Vector Address + (32 × Exception Number)`.



9.2.6 RISC-V Machine-Mode Exception Handling

Control and Status Registers (CSRs)

Control and Status Registers (CSRs) are essential for handling exceptions and managing processor states. Key registers include:

- **mstatus:** Contains global interrupt enable flags (e.g., `MIE`) and status bits.
- **mie:** Manages individual interrupt enable bits (e.g., `MEIE`, `MTIE`).
- **mip:** Indicates pending interrupts (e.g., `MEIP`, `MTIP`).
- **mtvec:** Stores the base address for exception vectors.
- **mepc:** Holds the Program Counter (PC) value at the time of an exception.
- **mcause:** Stores the cause of the exception, differentiating between interrupts and exception codes.

Instructions for Accessing CSRs

The following instructions facilitate read/write operations on CSRs:

Instruction	Pseudocode	Meaning
`csrrw rd, csr, rs1`	`rd ← csr; csr ← rs1`	Read/Write CSR
`csrrs rd, csr, rs1`	`rd ← csr; csr ← csr | rs1`	Read/Set Bits CSR
`csrrc rd, csr, rs1`	`rd ← csr; csr ← csr & ( rs1)`	Read/Clear Bits CSR
`csrrwi rd, csr, imm`	`rd ← csr; csr ← imm`	Read/Write CSR (Immediate)
`csrrsi rd, csr, imm`	`rd ← csr; csr ← csr | imm`	Read/Set Bits CSR (Immediate)
`csrrci rd, csr, imm`	`rd ← csr; csr ← csr & ( imm)`	Read/Clear Bits CSR (Immediate)

Returning from Exceptions

The `mret` instruction is used to return from machine mode:

- Restores the previous interrupt enable state (`mstatus.MIE ← mstatus.MPIE`).
- Updates the Program Counter (`pc ← mepc`).

9.2.7 RISC-V Interrupt and Exception Codes

The RISC-V architecture defines a set of interrupt and exception codes, categorized based on their source and purpose. These codes are represented in the `mcause` register, with the highest bit (`mcause[31]`) distinguishing between interrupts and exceptions.

Interrupts

Interrupts occur asynchronously and are triggered by external or internal events. Key examples include:

- **Supervisor software interrupt (Code 1):** Triggered by software at the supervisor level.
- **Machine software interrupt (Code 3):** Triggered by software at the machine level.
- **Supervisor timer interrupt (Code 5):** Indicates a timer interrupt at the supervisor level.
- **Machine timer interrupt (Code 7):** Indicates a timer interrupt at the machine level.
- **Supervisor external interrupt (Code 9):** Indicates an external I/O interrupt at the supervisor level.
- **Machine external interrupt (Code 11):** Indicates an external I/O interrupt at the machine level.

Exceptions

Exceptions are synchronous events that occur during instruction execution. Common examples include:

- **Instruction address misaligned (Code 0):** Caused by an attempt to fetch an instruction from a misaligned address.
- **Instruction access fault (Code 1):** Occurs when an instruction fetch violates access permissions.
- **Illegal instruction (Code 2):** Raised when an undefined or restricted instruction is executed.
- **Breakpoint (Code 3):** Triggered by a breakpoint instruction for debugging.
- **Page faults:**
  - **Instruction page fault (Code 12):** Indicates a fault during instruction fetch due to virtual memory issues.
  - **Load page fault (Code 13):** Raised during a load operation when a page-related fault occurs.
  - **Store page fault (Code 15):** Raised during a store operation when a page-related fault occurs.

Understanding and properly handling these interrupts and exceptions is crucial for effective RISC-V programming and system design.

9.2.8 Possible Undefined Instruction Handler

Below is a possible implementation of an undefined instruction handler in RISC-V assembly:

```
handler:  addi  sp, sp, -128          # Save all registers but sp (leaving space in memory so that it is a regular array)
          sw    x0, 0(sp)
          sw    x1, 4(sp)
          sw    x3, 12(sp)
            … etc. …
          sw    x31, 124(sp)

          csrr  a1, mcause            # Read exception cause
          beqz  a1, interrupt         # Branch if not an exception
          andi  a1, a1, 0x3f          # Isolate exception cause
          li    a2, 2                 # a2 = illegal instruction cause
          bne   a1, a2, otherExcp     # Branch if not an illegal instruction exception

          csrr  t0, mepc              # Load the current faulting instruction address
          lw    a0, 0(t0)             # Read the faulty instruction
          jal   decodeInst            # Gets a0 = instruction; returns a0 = 0 if it cannot be emulated or a0 = operation, a1, a2, and a3 = index of rs1, rs2, and r
          beqz  a0, undefinedInst     # Branch if really undefined and not possible to emulate
          mv    s0, a3                # Save index of rd
            … etc. …                  # Read from the stack the content of the original registers pointed by a1 and a2 into a1 and a2 ("read the source registers")
          jal   emulateInst           # Gets a0 = operation, a1, a2 = operands; returns a0 = FP result
            … etc. …                  # Write a0 on the stack into original register pointed by s0 ("write the destination register")

          lw    x1, 4(sp)             # Restore all registers but zero and sp
            … etc. …
          lw    x31, 124(sp)
          addi  sp, sp, 128

          csrr  t0, mepc              # Load the current faulting instruction address
          addi  t0, t0, 4             # Increment to point to the next instruction (4 bytes for RV32/RV64)
          csrw  mepc, t0              # Write the updated value back to me
          mret
```

9.2.9 RISC-V Machine-Mode Interrupt Handling

In RISC-V architecture, machine-mode interrupt handling is managed through three key control and status registers: `mie`, `mip`, and `mstatus`. These registers play distinct roles in enabling, monitoring, and controlling interrupts.

- **`mie` (Machine Interrupt Enable):** This register determines which interrupts the processor can take and which it must ignore. Key bits include:
  - `MEIE`: Enables machine-level external interrupts.
  - `MTIE`: Enables machine-level timer interrupts.
- **`mip` (Machine Interrupt Pending):** This register lists the interrupts that are currently pending. Key bits include:
  - `MEIP`: Indicates a pending machine-level external interrupt.
  - `MTIP`: Indicates a pending machine-level timer interrupt.
- **`mstatus` (Machine Status):** This register contains the global interrupt enable flag and other state information. Important fields include:
  - `MIE`: Globally enables interrupts when set to 1, and disables them when set to 0.
  - `MPIE`: Holds the value of `MIE` prior to a trap.

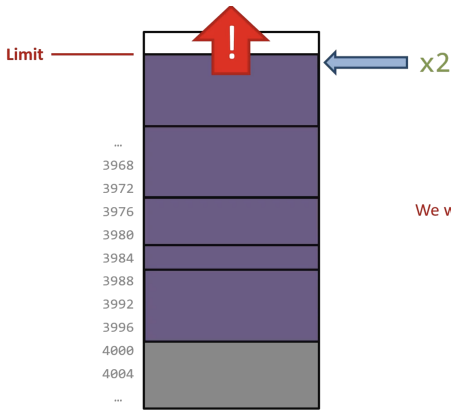
The diagram below illustrates the structure of these registers:

mstatus	*reserved*			MPIE	*reserved*	MIE	*reserved*
mie	*reserved*	MEIE	*reserved*	MTIE	*reserved*		
mip	*reserved*	MEIP	*reserved*	MTIP	*reserved*		

These registers provide the foundation for interrupt handling in machine mode, ensuring efficient and precise interrupt management.

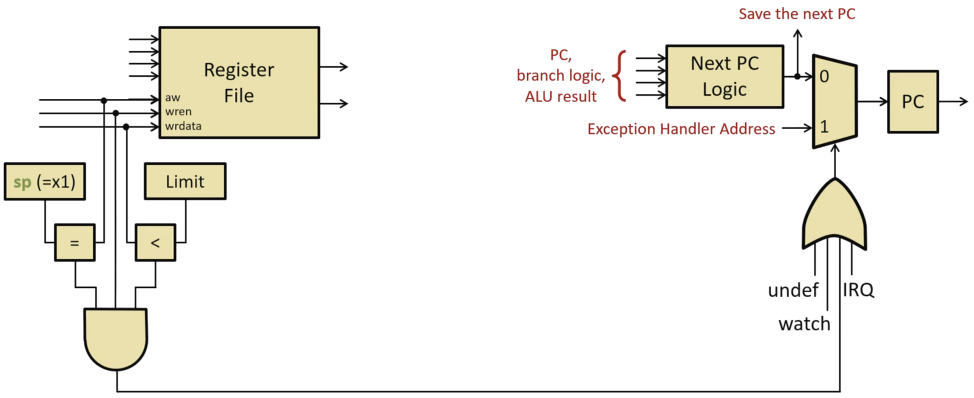
9.3 The Stack Problem

A few weeks ago, we discussed a potential issue with the stack, that was,
"What should we do when the stack hits its limit?"
We might be able to find a solution to this problem now.



9.3.1 Stack-Full Detection ?

To detect when the stack is full, we can use a watchpoint.



9.3.2 Writing Handlers is Very Very Tricky

To write the exception handler for the stack-full detection, we **cannot** use the stack.
Writing interrupt or exception handlers is inherently complex, particularly due to the restriction that the stack cannot be used. Additionally, many registers may be untouchable during execution. This necessitates careful design to handle these constraints.

Challenges

- **Stack usage**: Direct stack usage is prohibited, necessitating alternative storage mechanisms.
- **Register constraints**: In many cases, touching any general-purpose register is disallowed.

Solutions

Various architectures provide mechanisms to address these challenges:

- **Reserved Registers**: As seen in MIPS, specific registers such as `$k0` and `$k1` are reserved for handler use.
- **Shadow Registers**: Early processors (e.g., x86 and earlier) employ shadow registers for temporary data storage.
- **Safe Stack Switching**: Architectures like x86 allow automatic switching to a predefined safe stack.

RISC-V Solution

The RISC-V architecture employs a dedicated Control and Status Register (CSR) called `mscratch` (Machine Scratch) to facilitate temporary data storage during handler execution. The `mscratch` register can:

- Store one word of data for temporary usage.
- Hold a pointer to an empty memory region or a predefined safe stack.

9.3.3 Speaking of the Stack...

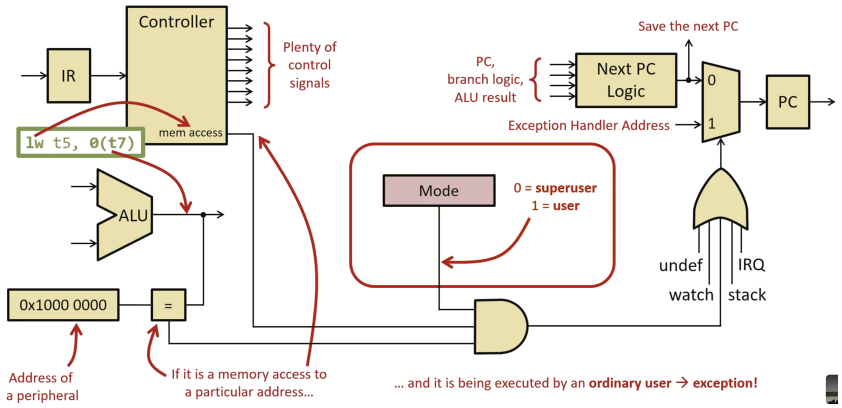
Speaking of the stack, when writing assembly code, we often ask ourselves: is the code on the right side as valid as the code on the left side? The answer is yes, but handling stack overflow, as we were planning to do using a watchpoint, would've made it impossible to write the right version.

```
funct:
    addi sp, sp, -12
    sw   ra, 0(sp)
    sw   s0, 4(sp)
    sw   s1, 8(sp)
    ... etc. ...
    lw   ra, 0(sp)
    lw   s0, 4(sp)
    lw   s1, 8(sp)
    addi sp, sp, 12
    ret
```

```
funct:
    sw   ra, -12(sp)
    sw   s0, -8(sp)
    sw   s1, -4(sp)
    addi sp, sp, -12
    ... etc. ...
    addi sp, sp, 12
    lw   ra, -12(sp)
    lw   s0, -8(sp)
    lw   s1, -4(sp)
    ret
```

9.4 Protection: I/Os Are Not for Everyone

The system enforces restricted access to I/O peripherals to ensure security and proper operation.



- **Instruction Register (IR):** Holds the current instruction, e.g., `lw t5, 0(t7)`, which specifies a memory access operation.
- **Controller:** Generates the necessary control signals to manage instruction execution and memory access.
- **ALU (Arithmetic Logic Unit):** Computes addresses and performs arithmetic or logical operations. In this case, it calculates the effective memory address for the load instruction.
- **Peripheral Address Check:** Compares the computed address against a predefined address (`0x10000000`) to determine if the operation targets a restricted peripheral.
- **Privilege Mode:** Maintains the execution mode:
  - `0`: Superuser mode.
  - `1`: User mode.

  If a user-mode instruction attempts to access a restricted address, an exception is triggered.
- **Exception Handling:** Routes control to an exception handler when a violation occurs. This involves:
  - Saving the next program counter (PC).
  - Redirecting execution to the exception handler address.
- **Interrupt Sources:** Includes various triggers such as undefined instructions (`undef`), IRQ, watchpoints, and stack violations, which can cause exceptions.

This mechanism ensures secure memory access, prevents unauthorized peripheral usage, and enforces privilege separation to maintain system security.

9.4.1 Levels of Privilege: Processor Modes

Modern processors are designed with multiple levels of privilege to ensure proper execution of user programs and operating system tasks. These levels are referred to as **processor modes** and include: - **Distinction Between Processor Modes:**

- **User Mode:** For executing user programs.
- **Kernel/Supervisor/Executive Mode:** For handling operating system tasks and privileged instructions.
- **RISC-V:** Includes up to three modes: Machine, Supervisor, and User.

- **Processor State and Privilege Levels:**

- Some parts of the processor state are *readable by all* privilege levels, but can only be *written to* by the highest privilege levels.
- Examples include:
  - The **current mode register**.
  - Configuration registers (e.g., memory hierarchy configuration).

- Methods for Switching Between Modes:

- Processors provide:
  - A *dedicated instruction* to trigger a software exception.
  - An instruction to reset the mode.
- In RISC-V:
  - `ecall` is used for system calls.
  - `mret/sret` are used to return from exceptions.

9.4.2 Processor Tasks on Exception

When an exception is raised, the processor typically performs a series of tasks. These tasks depend on the processor architecture and the type of exception. The main steps include:

1. **Mask further interrupts:** Prevent additional interrupts to ensure the exception is handled correctly.
2. **Save the Exception Program Counter (EPC):** Store the address of the instruction causing the exception.
3. **Save exception details:** Record the reason or context for the exception.
4. **Modify privilege level:** Switch to a higher privilege level as exception handlers often run in privileged mode.
5. **Free up registers:** Temporarily save or copy certain registers to shadow registers if supported.
6. **Jump to the exception handler:** Transfer control to the handler routine.

After the exception is handled, most or all of these tasks are *implicitly reverted* using special instructions. For example, the `mret` instruction in RISC-V resets the privilege level and re-enables interrupts.

Some tasks, however, must be *explicitly reverted* by the handler. For instance, programmers may need to unmask further interrupts manually as soon as it is safe to do so.

9.4.3 Priorities in Interrupt Handling

Interrupt controllers play a critical role in managing priorities, determining which interrupt is more urgent to serve. While hardware mechanisms primarily affect the order in which Interrupt Requests (IRQs) are presented to the processor, there are scenarios where it is desirable to serve a high-priority interrupt even while handling a lower-priority one.
However, there is a limitation: the processor architecture typically provides only a single `mepc` (Machine Exception Program Counter) and `mcause` (Machine Cause Register) register. Consequently, as soon as the processor accepts an interrupt, it must disable further interrupts to preserve critical state information.

Potential Solutions

To address this limitation, we can implement the following strategies:

- **Saving State:** Critical information about the interrupt (`mepc`, `mcause`, `mstatus`) can be saved on a safe stack. This ensures that Control and Status Registers (CSRs) can be overwritten by subsequent interrupts without losing essential context.
- **Re-enabling Interrupts:** The `mstatus` register can be manually updated to re-enable interrupts without returning from the handler, allowing higher-priority interrupts to preempt the current one.

9.4.4 More challenges in Writing Exception Handlers

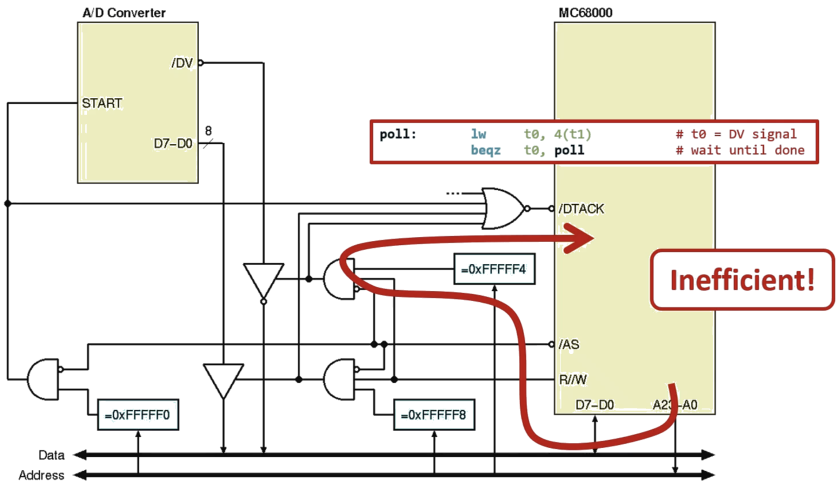
Writing exception handlers is a complex and challenging task due to the following reasons:

- **Stack Constraints:** In some cases, the stack cannot be used, such as when the exception arises from a stack overflow.
- **Non-Interruptibility:** Exception handlers may not be interruptible if they rely on static locations to save data, including registers like `mscratch`, making them non-reentrant.
- **System Limitations:** The system might be unable to tolerate prolonged interruptions, for instance, when I/O buffers risk filling up due to unserved interrupts.

Additionally, buggy **device drivers** from peripheral vendors often run in privileged mode, invoked by the operating system's interrupt handler, and can destabilize the system.

9.5 Example - Back to Our A/D Converter

Let's revisit the example of an A/D converter, which converts analog signals to digital data. This device is connected to the processor via an I/O port, and the processor reads the converted data from the device.



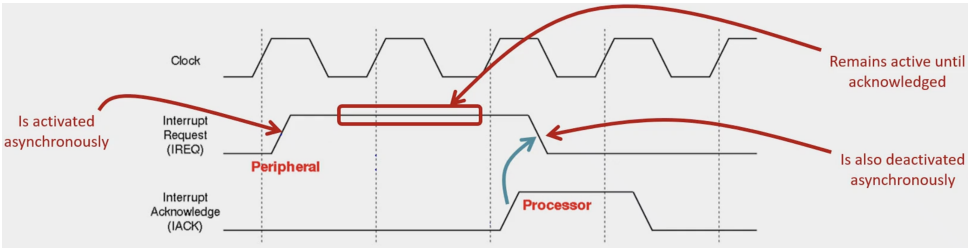
Here though, we used to have an efficient approach to read the data from the A/D converter which was that we used to keep reading the Data Valid signal waiting for it to become active. (highly inneficient)

9.5.1 Simple IREQ and IACK Mechanism

In an 8-bit processor with an internal interrupt controller, various `IREQ/IACK` signal pairs are used for I/O interrupt requests. For the Analog-to-Digital Converter (ADC), the following signals are assigned:

- `IREQ3`: An input signal dedicated to the peripheral to request attention from the processor.
- `IACK3`: An output signal used by the processor to acknowledge the request and signal that it is being served.

The interaction between the peripheral and the processor can be described as follows:



1. The interrupt request (`IREQ`) is activated asynchronously by the peripheral.
2. The request remains active until it is acknowledged by the processor.
3. Once the processor acknowledges the request using `IACK`, the interrupt is served, and `IREQ` is deactivated asynchronously.

This mechanism ensures smooth communication and handling of interrupt-driven tasks between the peripheral and the processor.

9.5.2 A/D Converter - startADC

The `startADC` function initiates the Analog-to-Digital Conversion process by setting the `start` bit in the control register of the A/D converter.

```
1    lui t0, 0xfff
2    addi t0, t0, 0xff0 #t0 = 0xffffff0
3    sw zero, 0(t0)     # start conversion
4    ret
```

9.5.3 A/D Converter - Software:handler

```
handler:
    addi sp, sp, -120          # Save all registers but zero and sp
    sw x1, 0(sp)
    sw x3, 4(sp)
    ...                        # etc.
    sw x31, 116(sp)

    csrr s0, mcause            # Read exception cause
    bgez s0, handleExceptions  # Branch if not an interrupt (MSB = 0,
                               # looks like zero or a positive number...)
    slli s0, s0, 1             # Get rid of the MSB of s0,
    srli s0, s0, 1             # so that what is left is the cause
    li s1, 11                  # s1 = external interrupt cause
    bne s1, s2, handleOtherInts # Branch if not an external interrupt

    jal readADC                # Returns a0 = ADC result
    jal insertIntoBuffer       # Gets a0 = value to add to a circular buffer

restore:
    lw x1, 0(sp)               # Restore all registers but zero and sp
    lw x3, 4(sp)
    ...                        # etc.
    lw x31, 116(sp)
    addi sp, sp, 120

    mret                       # Return from interrupt
```

9.5.4 A/D Converter - insertIntoBuffer

```
.section .data
    .equ bufferSize, 1024              # Define buffer size (must be a power of two)
    .equ bufferBytes, bufferSize * 4   # Compute the total size in bytes for the
        buffer
    bufferPointer:
        .word 0                        # Initialize the pointer to index 0
    buffer:
        .space bufferBytes             # Allocate space for bufferSize * wordSize
            bytes

.section .text
insertIntoBuffer:
    la    t0, bufferPointer            # Load address of bufferPointer into t0
    lw    t1, 0(t0)                    # Load current buffer pointer into t1
    la    t2, buffer                   # Load base address of the buffer into t2
    slli  t3, t1, 2                    # Multiply buffer pointer (t1) by 4 to get byte
         offset
    add   t4, t2, t3                   # Add offset to buffer base address (= next
        word)
    sw    a0, 0(t4)                    # Store a0 into buffer at calculated position
    addi  t1, t1, 1                    # Increment buffer pointer by 1
    li    t5, bufferSize - 1           # Load bufferSize - 1 into t5 (mask for power
        of 2)
    and   t1, t1, t5                   # Apply bitwise AND to wrap around
    sw    t1, 0(t0)                    # Store updated buffer pointer
    ret                                # Return from the function
```

9.5.5 A/D Converter - readADC

```
readADC:
    li t0, 0xfffff0 # t0 = 0xfffff0
    lw a0, 8(t0) # get ADC data output
    ret
```

Chapter 10

Part II(e) - Processor, I/Os, and Exceptions - Example W - 6.1

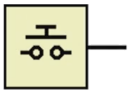
10.1 Part Ia: Connecitng an Input Peripheral

Consider a hypothetical processor with the following buses and control signals:

- **A[31:0]**: Address bus
- **D[31:0]**: Data bus
- **AS** (Address Strobe): Active when a valid address is present on **A[31:0]**.
- **WR** (Write): Active along with **AS** during a write cycle.

The input peripheral consists of 10 buttons numbered from 0 to 9, where:

Each button outputs a logic '1' when pressed and '0' otherwise.

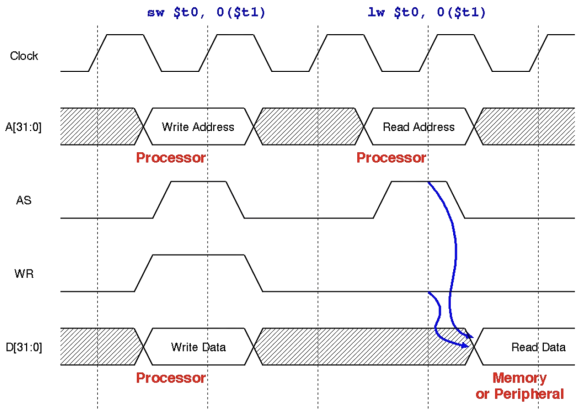


The processor reads the **state of the buttons** from memory location `0xFFFF'FFF0`:

- A value of '0' indicates no button is pressed.
- A value of '1' indicates at least one button is pressed.

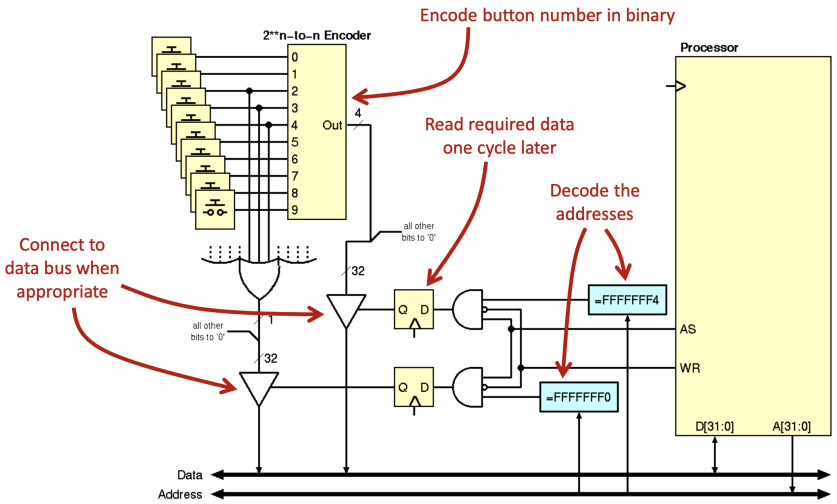
The processor reads the **number of the button pressed** from memory location `0xFFFF'FFF4`.

10.2 Bus Protocol



10.3 Assembling the Circuit

Looking at the timing diagram is really really recommended when assembling a circuit.



The input peripheral circuit connects 10 buttons to the processor, allowing it to detect button presses and identify which button is pressed. Below are the components of the circuit and their purposes:

- **Buttons:** Represent physical inputs numbered 0 to 9. Each button outputs a logic '1' when pressed and '0' otherwise.
- **2^n-to-n Encoder:** Converts the 10 individual button signals into a 4-bit output, representing the button number.
- **Address Decoders:** Determine the memory location being accessed (`0xFFFF'FFF0` or `0xFFFF'FFF4`) based on the address bus (**A[31:0]**).
- **Latches (Q):** Store the state of the buttons and the number of the button pressed, enabling stable data retrieval by the processor.
- **Control Signals (AS, WR):**
  - **AS** (Address Strobe): Ensures the address on **A[31:0]** is valid.
  - **WR** (Write Enable): Activates during write cycles to store data.
- **Data Bus (D[31:0]):** Transfers data between the processor and the peripheral.

10.4 Part 1b: Reading the Input Ports

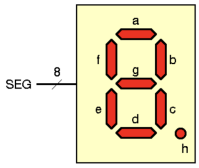
Write a RISC-V program named `buttons` to poll the state of the input buttons. The program must meet the following requirements:

- Every time a button is pressed, the program should call the function `ShowIt`.
- Register `a0` must contain the ASCII code of the character corresponding to the button pressed. For example:
  - Button "0" → ASCII code 48
  - Button "1" → ASCII code 49
  - Button "2" → ASCII code 50
- The function `ShowIt` is provided, and you do not need to implement it.

10.4.1 Software: buttons

```
    li s0, 0xFFFFFFF0
poll:
    lw t0, 0(s0)
    beqz t0, 0(s0)
    lw t0, 4(s0)
    jal showIt
    j poll
```

10.5 Part 2a - Connecting an Output Peripheral



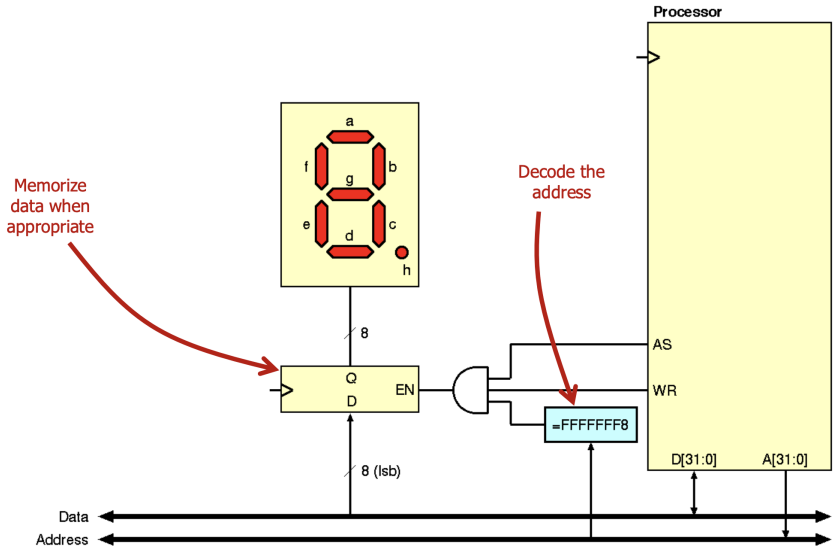
The peripheral receives an 8-bit signal, `SEG`, where each bit corresponds to a segment of the display:

- Bit 0 → Segment `a`
- Bit 1 → Segment `b`
- Bit 2 → Segment `c`, and so on.

A bit value of `1` indicates that the corresponding segment is lit.

The processor writes a digit to the display by performing a write operation to the memory location `0xFFFF'FFF8`.

10.6 Assembling everything



10.7 Part 3a: Use Interrupts

The processor includes three **Interrupt Priority Level** input pins, denoted as `IPL[2:0]`, which allow I/O devices to initiate interrupts. The interrupt handling mechanism functions as follows:

The binary value on the `IPL[2:0]` pins determines the **Priority Level** of the interrupt:

- `0`: No interrupt request.
- `1`: Lowest priority.
- `7`: Highest priority.

An interrupt is **served only if** its priority level exceeds the current level stored in the processor's special status register.

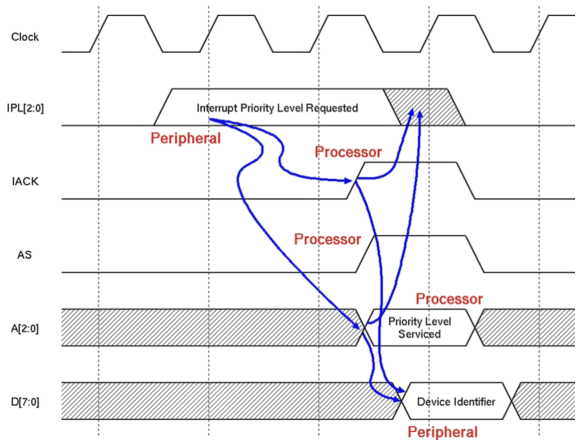
Interrupts at **Priority Level 7** are always served, as they are non-maskable.

Modify the button interface to generate an interrupt with priority level 3 and identifier 0x45 when a button is pressed.

The port at memory location 0xFFFF'FFF0 is no longer used.

10.7.1 Interrupt Acknowledgement Process

The interrupt acknowledgement process involves a coordinated sequence between the processor and peripheral devices. The key steps are outlined as follows:



The peripheral device asserts an **Interrupt Priority Level Request** on the `IPL[2:0]` lines, indicating the interrupt priority level.

The processor recognizes the interrupt and initiates an **Interrupt Acknowledge (IACK)** signal to acknowledge the request.

The processor evaluates the priority level of the interrupt:

- If the priority level matches or exceeds the current threshold, the interrupt is serviced.
- Otherwise, the interrupt is ignored or deferred.

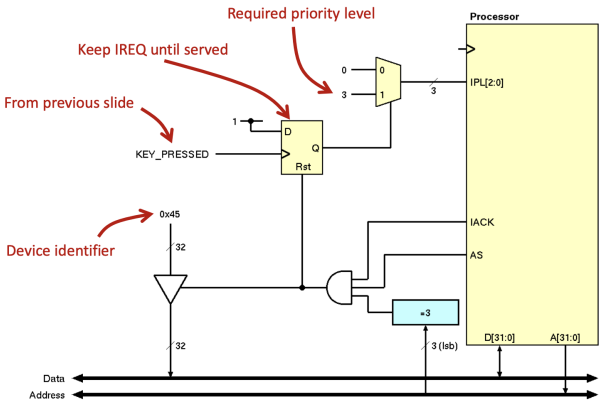
Once the interrupt is acknowledged, the processor asserts the **Address Strobe (AS)** signal to identify the interrupting device.

The peripheral responds by placing the **Device Identifier** on the `D[7:0]` data lines for the processor to process.

10.7.2 Solution

Final circuit solution with interrupt handling mechanism:

- **Retaining the Interrupt Request (IREQ):** The `IREQ` signal is maintained active until the interrupt is served. This ensures that the request is not lost if the processor is handling a lower-priority task at the time of the interrupt.
- **Required Priority Level:** The updated implementation includes a multiplexer that selects the required priority level for the interrupt. This allows the processor to dynamically assess whether the priority of the interrupt request is sufficient to preempt the current task.
- **Device Identifier:** The device identifier (`0x45`) is explicitly encoded and transmitted over the data bus. This provides a clear and efficient way for the processor to identify the source of the interrupt.
- **Integration with the Processor:** The processor interacts with the interrupt handling circuitry through the `IPL[2:0]`, `IACK`, and `AS` signals. These signals ensure that the interrupt servicing process aligns with the required priority levels and device identification.
- **New Comparator Mechanism:** A comparator checks whether the least significant bits (LSBs) of the address match the expected value (`=3`). This additional check further validates the interrupt source and enhances system robustness.



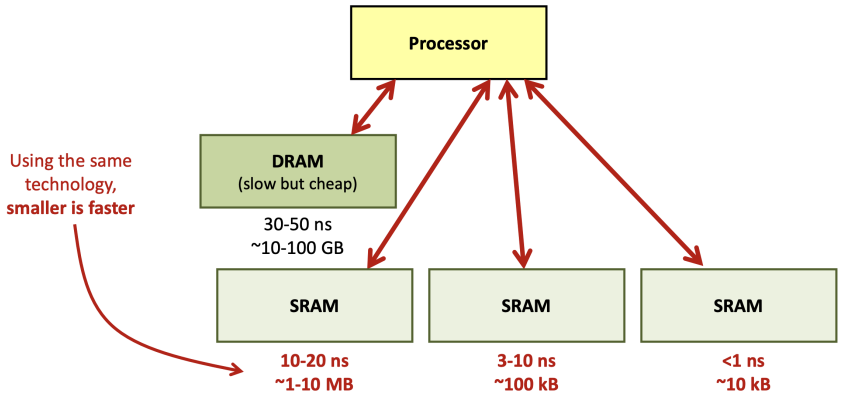
Chapter 11

Part III(a) - Memory Hierarchy - Caches - W.6.2 - 7.1

Memory is a fundamental component of computing systems, as its performance directly impacts the speed at which data can be accessed and processed, influencing overall system efficiency.
The balance we aim to achieve lies in optimizing speed to meet the demands of the CPU while maximizing memory capacity to satisfy our requirements.

11.1 Our Goal : Use Different Memories

Instad of using a single memory, with fixed monolothitic caracteristics, we can use a hierarchy of memories, each with different characteristics.



- **DRAM (Dynamic Random Access Memory):**
  - *Characteristics:* DRAM is slow but cost-effective.
  - *Access Time:* Typically between 30 and 50 nanoseconds.
  - *Capacity:* Provides a large storage size, ranging from 10 GB to 100 GB.
- **SRAM (Static Random Access Memory):**
  - *Characteristics:* SRAM is fast but expensive.
  - *Access Time:* Less than 1 nanosecond.
  - *Capacity:* Offers smaller storage sizes, around 10 KB.

Objective: The goal is to leverage the advantages of both DRAM and SRAM to achieve an efficient memory system, combining the speed of SRAM with the cost-effectiveness and capacity of DRAM.

Can we get the best of both worlds?

11.1.1 What Memory ot Use?

Efficient memory usage is crucial for ensuring high performance in iterative computations, as seen in the following example:

```
1 i = 0;
2 sum = 0;
3 while (i < 1024) {
4     sum = sum + a[i];
5     i = i + 1;
6 }
```

- **Instruction locality:** Instructions corresponding to lines 3-5 in the loop are read repeatedly. These should reside in fast memory (e.g., caches) to minimize latency.
- **Variable access:** Frequently accessed variables like `i` and `sum` should be stored in fast memory, such as registers or cache, to reduce access time.
- **Prefetching:** To improve performance, future instructions and array elements (e.g., `a[i+1]`, `a[i+2]`, etc.) can be loaded into memory in advance using techniques like hardware or software prefetching.

11.1.2 Spatial and Temporal Locality

Two important criteria for deciding on data placement in memory are:

Temporal Locality This refers to data that has been **used recently** and thus has a high likelihood of being reused. Examples include:

- **Code:** loops, functions, etc.
- **Data:** local variables and data structures.

Spatial Locality This refers to data located **near other data currently in use**, which is likely to be accessed soon. Examples include:

- **Code:** sequentially read instructions.
- **Data:** arrays and other contiguous structures.

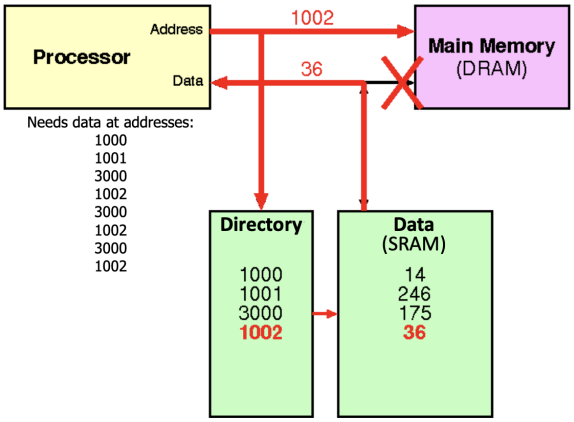
11.1.3 Placement Policy Design

Our placement policy must satisfy two essential requirements:

- **Invisible to the Programmer:**
  - While it is possible to analyze data structures and program semantics to detect heavily used variables or arrays for placement decisions, this approach is only suitable in specific contexts (e.g., embedded systems).
  - The goal is to alleviate the burden on programmers by introducing hardware mechanisms to manage placement transparently.
- **Extremely Simple and Fast:**
  - Placement decisions, when delegated to hardware, must be simple to ensure efficiency.
  - The primary objective is to facilitate access to fast memory within nanoseconds or less, leaving little room for complex decision-making.

11.2 Cache: The Idea

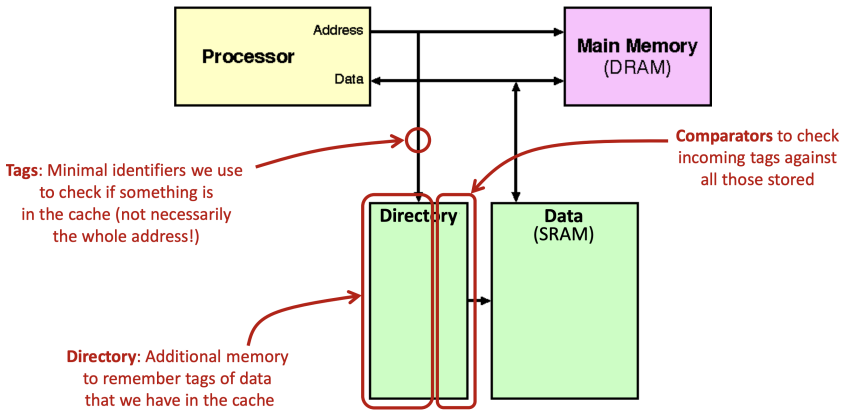
Caching is a mechanism used to improve the performance of a computer system by storing frequently accessed data in a smaller, faster memory known as the cache.



- **Processor Request:** The processor requires data located at specific memory addresses, such as `1000`, `1001`, `3000`, and `1002`.
- **Cache Directory and Data:** The cache maintains a directory that maps memory addresses to their corresponding data stored in the cache. For example, the data for address `1002` is located in the cache with a value of `36`.
- **Cache Hit:** If the requested address exists in the cache directory (e.g., `1002`), the processor retrieves the data directly from the cache. This is referred to as a *cache hit*.
- **Cache Miss:** If the requested address is not found in the cache, the data is fetched from the main memory (DRAM), stored in the cache, and then delivered to the processor.
- **Performance Benefit:** By prioritizing access to the cache (SRAM), which is faster than main memory (DRAM), the system reduces latency and improves overall performance.

11.2.1 Cache Memory: Directory and Tags

Cache memory is not just about speed but also about efficient management of data. Two critical components of a cache system are the **Directory** and **Tags**, which play a vital role in ensuring data consistency and fast access.



- **Tags:** Minimal identifiers used to determine if a specific data block is in the cache. These identifiers are typically smaller than the full memory address, optimizing comparison speed and storage requirements.
- **Directory:** A dedicated memory structure that stores the tags of data currently present in the cache. This allows for efficient lookups and ensures the correct data is retrieved.
- **Comparators:** Hardware elements that compare incoming tags against those stored in the directory. They enable quick validation of cache hits or misses.

11.2.2 Cache Hits and Misses

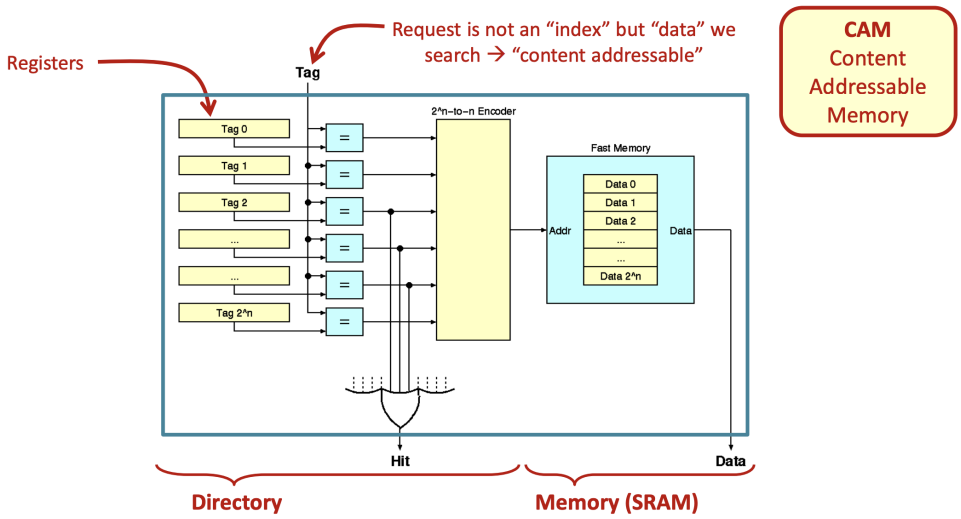
A **cache** is a form of storage that *automatically* leverages the **locality of accesses** to improve performance. The concept has been widely adopted beyond processors, and examples include:

- Web browsers caching frequently accessed data.
- Network routers caching routing information.
- DNS servers caching frequent domain names.
- Databases caching queries.

When the required data is found in the cache, it is called a **Hit**. Conversely, if the data is not found, it is termed a **Miss**.
The **Hit Rate** (or **Miss Rate**) is defined as the ratio of hits (or misses) to the total number of accesses.

11.2.3 Fully-Associative Cache

A fully-associative cache is a type of cache memory that allows any block of main memory to be stored in any cache line.



Key Components

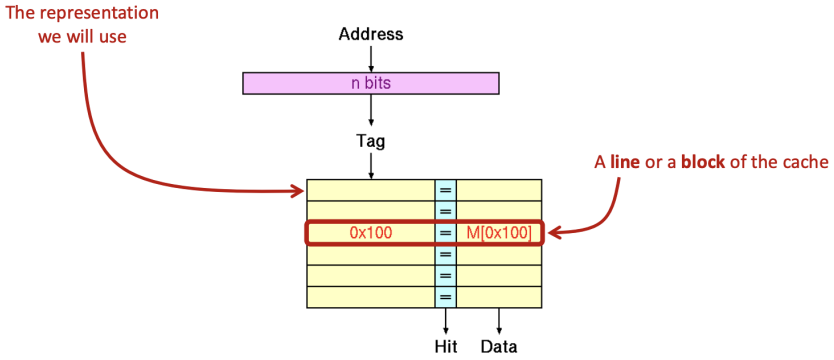
1. **Directory (Tag Array):** Each entry in the tag array (referred to as "Tags") stores metadata about the cached blocks. Tags uniquely identify the memory block stored in each cache line.
2. **Content Addressable Memory (CAM):** To find a specific block in the cache, the requested address is compared simultaneously with all stored tags. This parallel comparison is achieved using CAM, which enables *content-based addressing*.
3. **Comparison Logic:** The CAM outputs a signal indicating whether the requested tag matches any of the stored tags. This determines a *hit* or *miss*.
4. **Fast Memory (SRAM):** This stores the actual data blocks associated with each tag. When a hit occurs, the data corresponding to the matched tag is retrieved from this memory.
5. **Encoder:** If a match (hit) is found, the encoder selects the appropriate line from the data memory to access the requested data.

How It Works

1. A memory access request is initiated with an address containing the desired data's *tag*.
2. The tag is compared in parallel against all tags stored in the directory using CAM.
3. If a match is found, a *hit* is signaled, and the corresponding data is retrieved from the fast memory using the encoder. If no match is found, a *miss* occurs, and the block is fetched from main memory.
4. The requested data is returned to the processor.

11.2.4 Fully-Associative Cache

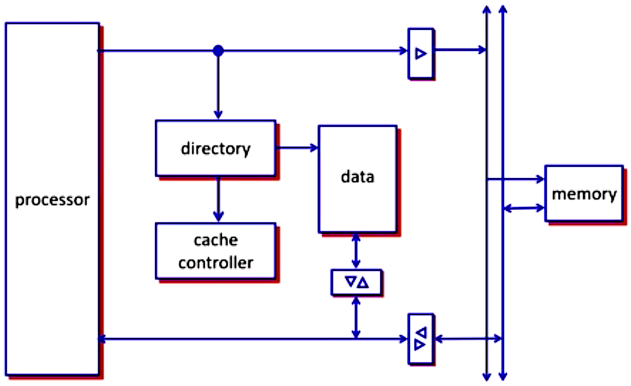
In a *fully-associative cache*, any block of memory can be stored in any cache line. This flexibility removes the need for a specific mapping between memory blocks and cache lines, allowing for maximum utilization of cache space.



- **Address Representation:** The memory address is represented using n bits. Each address can be divided into a *tag* and an optional *block offset* (if the cache stores data in blocks).
- **Cache Lines:** Each line (or block) of the cache contains:
  1. The *tag* to identify the memory block stored in that line.
  2. The actual *data* fetched from memory.

11.3 Cache and Cache Controller

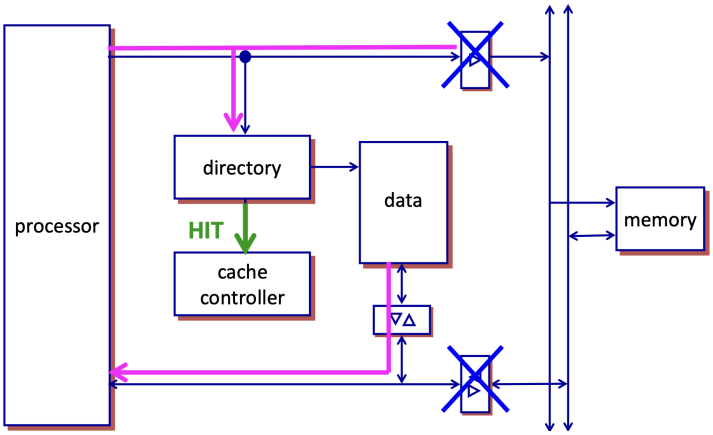
The cache and cache controller system serves as an intermediary between the processor and the main memory to enhance the overall performance of memory access. Below is an explanation of each component in the system:



- **Processor:** The central processing unit (CPU) that executes instructions and requests data from the memory hierarchy.
- **Directory:** A component that tracks the mapping of cached data to memory addresses. It determines whether a requested data item is present in the cache (cache hit) or not (cache miss).
- **Cache Controller:** This module manages the operation of the cache. It controls data flow between the directory, cache memory (data), and main memory. It ensures coherency and consistency of data when multiple memory requests occur.
- **Data:** The actual memory space within the cache that stores copies of frequently accessed data from main memory.
- **Main Memory:** The primary storage location that holds all data and instructions required by the processor. It communicates with the cache when the required data is not found in the cache (cache miss).
- **Bidirectional Arrows:** Indicate the flow of data between components. Data can flow from the processor to the cache, from the cache to the main memory, and vice versa.
- **Control Signals:** Represented as small triangular symbols, these signals are responsible for controlling the flow of data and ensuring synchronization between components.

11.3.1 Cache Hit

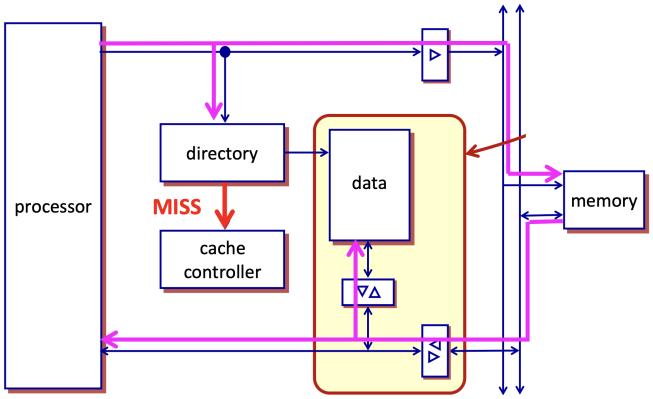
A *cache hit* occurs when the processor requests data that is present in the cache. This scenario significantly improves performance by reducing the latency associated with memory access.



1. The processor sends a request for a specific data item.
2. The *directory* checks whether the requested data is available in the cache.
3. If a match is found, a *hit* is registered, and the *cache controller* retrieves the data from the cache.
4. The requested data is then sent directly back to the processor without accessing the main memory, as indicated by the absence of memory interaction in this scenario.

11.3.2 Cache Miss

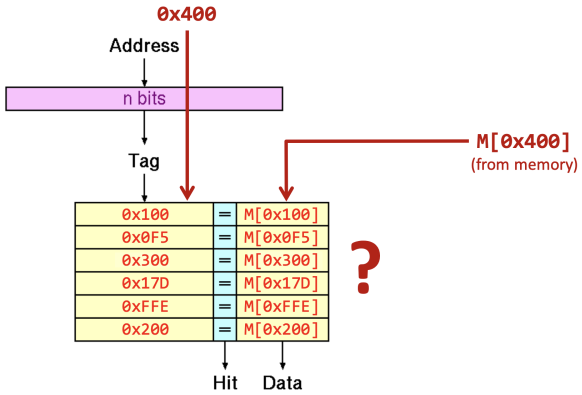
A cache miss occurs when the requested data is not present in the cache, requiring retrieval from the main memory.



1. **Processor Request:** The processor issues a request for data. The request is forwarded to the cache directory.
2. **Directory Check:** The directory checks whether the requested data is present in the cache. If the data is not found, a *cache miss* is detected, and the directory informs the cache controller.
3. **Cache Controller:** Upon detecting a cache miss, the cache controller initiates a fetch operation from the main memory. It ensures that the requested data is retrieved and forwarded to the processor.
4. **Memory Access:** The cache controller sends a request to the main memory for the missing data. The memory responds by transferring the data back to the cache.
5. **Cache Update:** The retrieved data is stored in the cache for future use. The directory is updated to reflect the new data location in the cache.
6. **Processor Response:** Once the data is available in the cache, it is sent to the processor to fulfill the original request.

11.4 What if the Cache is Full?

When the cache is full, adding new data requires the eviction of existing data to make space. This process is governed by a **cache replacement policy**.



11.4.1 Eviction Policies

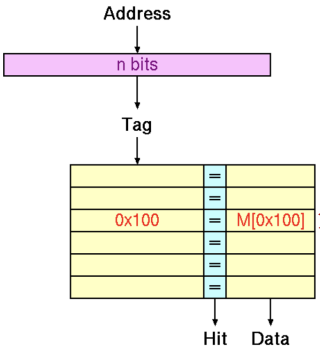
Various eviction policies determine which data to remove when the cache reaches its capacity limit. Common policies include:

- **Least Recently Used (LRU):**
  - Replaces the data that has been unused for the longest time.
- **First-In-First-Out (FIFO):**
  - Evicts the oldest data in the cache, based on the order of entry.
- **Random Replacement:**
  - Selects a cache line at random for eviction, regardless of usage or age.
- **Approximate Schemes:**
  - Use heuristics or approximations to determine eviction, balancing simplicity and performance.

Choosing the appropriate eviction policy depends on the application's requirements, balancing factors like speed, complexity, and data access patterns.

11.4.2 Only Exploiting Temporal Locality

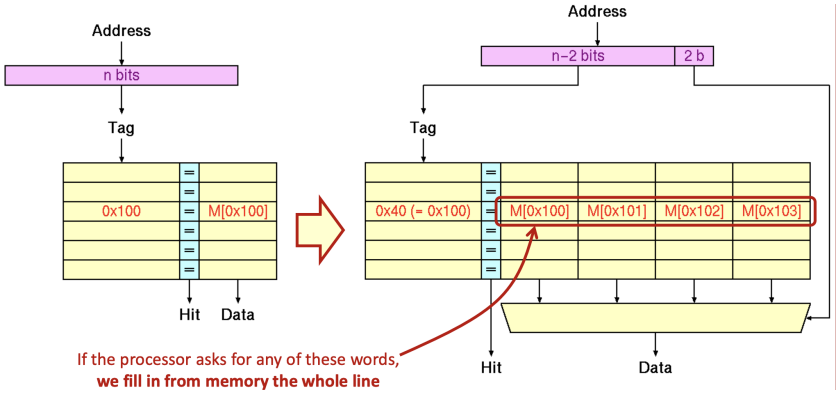
Temporal locality refers to the principle that data recently accessed is likely to be accessed again in the near future. In this approach:



- **Exclusive Data Fetching:** Only the specific data requested by the processor is fetched from the main memory into the cache, with the assumption that it will be reused soon.
- **Limitation:** This strategy does not account for *spatial locality*, which predicts that data stored near the requested data is also likely to be accessed. For example:
  - If the processor requests data at address `M[0x100]`, only this specific data is loaded into the cache.
  - If the processor later requires data at `M[0x101]` (a neighboring address), it results in a cache miss since this data is not preloaded.
- **Implication:** By focusing solely on temporal locality, the system may fail to optimize performance for workloads that benefit from spatial locality.

11.4.3 Exploiting Spatial Locality

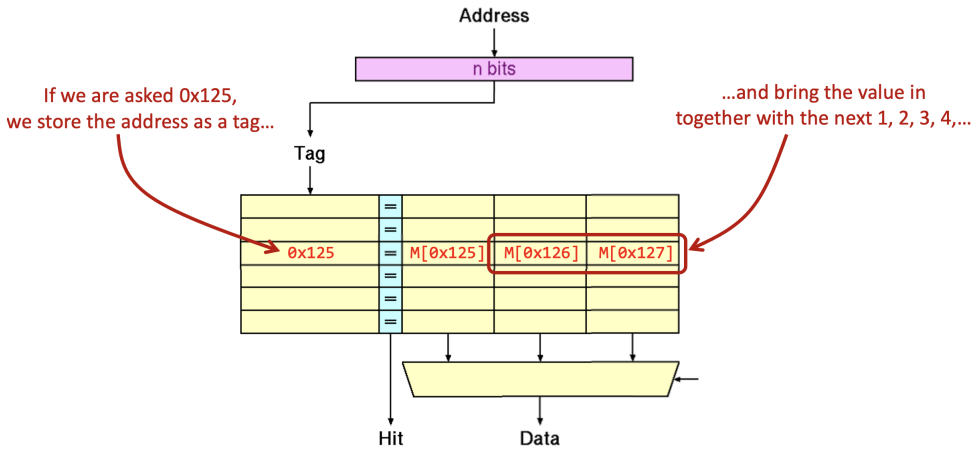
Spatial locality refers to the tendency of a program to access data locations that are close to one another within a short period. To optimize memory performance, the concept of spatial locality involves fetching more data than just the requested word. This process is outlined below:



- **Fetching Multiple Words:** When the processor requests data at a specific address (e.g., `M[0x100]`), the cache not only loads the requested word but also its neighboring words (e.g., `M[0x101]`, `M[0x102]`, and `M[0x103]`).
- **Cache Line:** The cache is structured in terms of lines, where each line contains multiple contiguous memory words. When a memory block is fetched, the entire line is populated in the cache.
- **Advantages:**
  - Reduces the number of cache misses for sequential or nearby data accesses.
  - Improves overall performance for workloads with high spatial locality.
- **Example Scenario:** In the diagram:
  - A request for `M[0x100]` results in the cache fetching an entire block containing `M[0x100]`, `M[0x101]`, `M[0x102]`, and `M[0x103]`.
  - If the processor subsequently requests `M[0x101]`, it is already available in the cache, avoiding a cache miss.

11.4.4 Why Not This ?

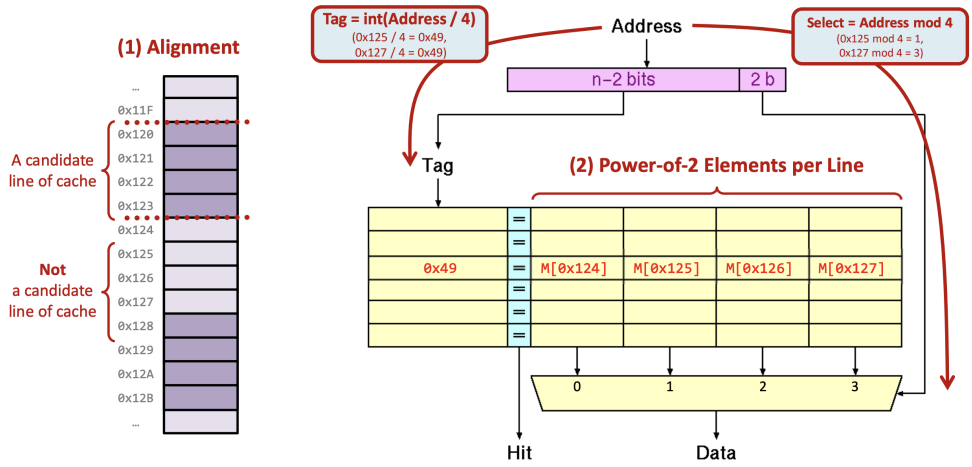
In a cache system, proper address matching and block selection are critical for accurate data retrieval. This subsection explores a specific scenario where matching the tag alone is insufficient and additional logic is required.



- **Address and Tag Matching:**
  - The memory address is divided into a *tag* and an *offset*.
  - The tag helps identify if the requested address corresponds to a block in the cache.
- **Problem:**
  - If a requested address (e.g., `0x127`) is part of a cached block identified by the tag `0x125`, the tag comparison alone cannot directly confirm the presence of the requested word.
  - The system must verify whether the requested address falls within the valid range of the block: $\texttt{Tag} \leq \texttt{Address} < \texttt{Tag + Block Size}$.

11.4.5 Solution

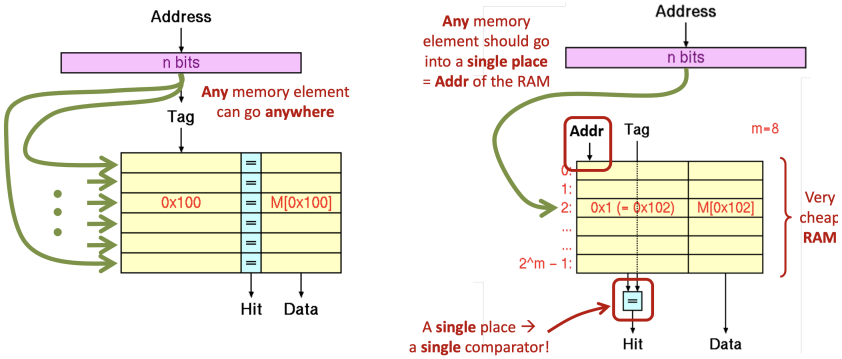
Efficient cache access relies on proper alignment and the use of power-of-2 elements per cache line. -*Personal Remark: The choice of a power of 2 is based on an absolute masterpiece that can be very useful,* $A\%2^n = A\&(2^n - 1)$



- **Alignment of Cache Lines:**
  - Memory addresses are divided into fixed-sized cache lines, with alignment based on the size of the line.
  - Only aligned addresses can serve as valid starting points for cache lines. For example:
    - Address `0x124` is a valid starting point.
    - Address `0x125` is not aligned and, therefore, cannot initiate a cache line.
- **Tag Calculation:**
  - The tag is computed as `Tag = int(Address / Line Size)`.
  - For example:
    - `Tag(0x125) = int(0x125 / 4) = 0x49`.
    - `Tag(0x127) = int(0x127 / 4) = 0x49`.
- **Power-of-2 Elements per Line:**
  - Each cache line contains a power-of-2 number of elements (e.g., 4 elements per line).
  - The specific word within the cache line is selected using the offset:
    - `Select = Address mod Line Size`.
    - Example: `0x125 mod 4 = 1` selects the second word.

11.4.6 Simplifying Cache Design

Simplifying the cache design can reduce hardware complexity and improve performance.



Current Approach:

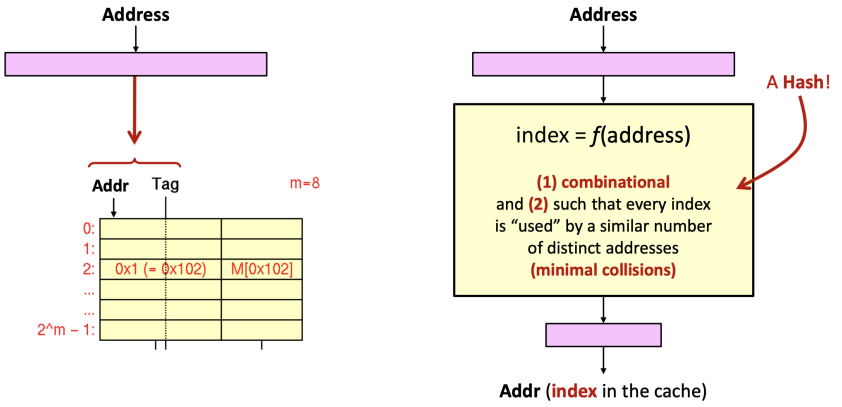
- Memory elements can map to any cache line, requiring multiple comparators to check the tags across all lines.
- This increases hardware cost and complexity.

Simplified Approach:

- Each memory element is mapped to a single, fixed cache location determined by its address.
- The cache uses the lower bits of the address to index into a specific line (`Addr` field).
- The `Tag` field contains the higher bits of the address to verify if the correct block is cached.

11.5 Generating `Addr` and `Tag`

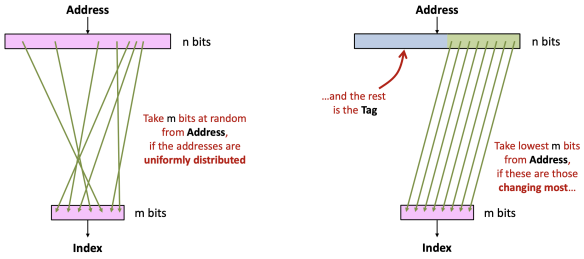
The process of generating the `Addr` and `Tag` from a memory address ensures efficient mapping and lookup in the cache. The procedure involves splitting the address into two components:



- **`Addr` (Index in Cache):**
  - The `Addr` is derived using a function $f(\texttt{Address})$, typically a hash function.
  - Properties of $f(\texttt{Address})$:
    - It is combinational (fast to compute).
    - It distributes memory addresses uniformly across cache indices to minimize collisions.
  - Example: For `0x102`, the computed `Addr` is the index in the cache where the block will reside.
- **`Tag` (Identification of the Block):**
  - The `Tag` represents the higher-order bits of the memory address, used to verify if the desired data resides in the indexed cache line.
  - For example, the `Tag` for `0x102` is `0x1`.
- **Hashing for `Addr` Generation:**
  - A well-designed hash function ensures:
    - Minimal collisions: Every cache index is mapped to by a similar number of distinct addresses.
    - Efficient access: Avoids excessive contention for cache lines.

11.5.1 The Simplest Hash Function

The simplest hashing schemes involve deriving an *Index* from an *Address* of n bits by selecting m bits. Two common approaches for selecting these m bits are as follows:

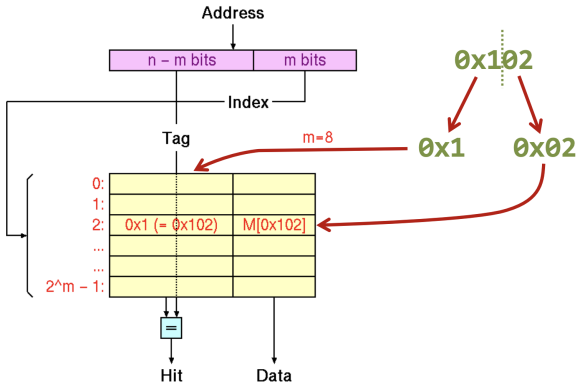


- **Random Selection:** Choose m bits at random from the n-bit *Address*, provided the addresses are uniformly distributed. This random selection ensures an even distribution of indices.
- **Lowest Bits Selection:** Select the m least significant bits (LSBs) from the *Address*. This method is particularly effective when the lower bits of the *Address* are those that change the most frequently.

Once the m bits are extracted to form the *Index*, the remaining $(n - m)$ bits constitute the *Tag*. The *Tag* can be used for further identification or verification processes.

Direct-Mapped Cache

A *Direct-Mapped Cache* is a simplified and cost-effective approach to caching that uses the *Address* to directly index into a specific cache line. This scheme divides the *Address* into two parts:



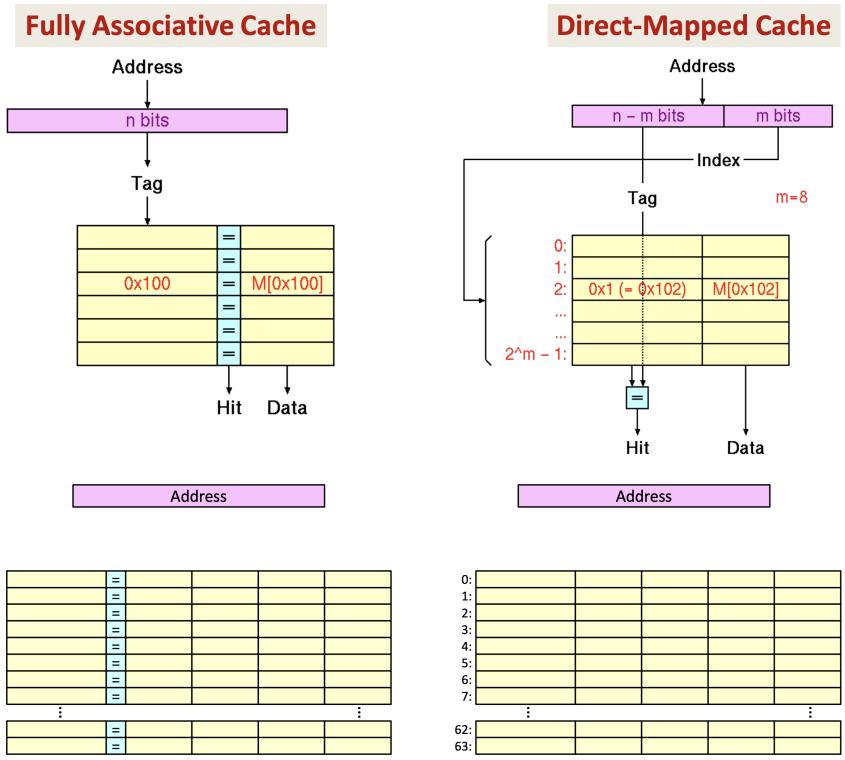
- **Index (m bits):** Specifies the cache line where the data is stored.
- **Tag ($n - m$ bits):** Identifies whether the cache line corresponds to the requested memory address.

The process of accessing the cache is as follows:

1. The *Index* is derived from the m least significant bits of the *Address*.
2. The corresponding cache line is checked, and the *Tag* is compared to validate the match.
3. If the *Tag* matches, it is a *cache hit*, and the requested data is fetched.
4. Otherwise, it is a *cache miss*, and the data must be retrieved from the main memory.

11.6 Which One is the Best Cache ?

Objective: Analyze the hit rate of two types of caches: *fully associative* and *direct-mapped*, given the following scenario.



Cache Details:

- Both caches have 64 lines with four words per line, resulting in a total of 256 words per cache.
- Addresses accessed sequentially: `0x100`, `0x101`, `0x200`, `0x102`, `0x300`, `0x103`, `0x201`, `0x102`, `0x301`, `0x103`, ...

Cache Architectures:

1. **Fully Associative Cache:** Any block can be stored in any cache line. Blocks are located using a tag comparison.
2. **Direct-Mapped Cache:** Each memory block maps to exactly one cache line, determined by the lower m bits of the address (index), where $m = 8$.

Analysis:

- **Fully Associative Cache:** As the cache has no fixed mapping of addresses to lines, it can utilize its capacity efficiently. Cache replacement policies (e.g., LRU) determine which block to evict upon a miss.
- **Direct-Mapped Cache:** Due to fixed mappings, cache conflicts occur when multiple addresses map to the same index. This can result in more frequent misses, even if the cache is not fully utilized.

ima **Question:** What is the **hit rate** of each cache under the given access sequence?

	0x100	**0x101**	**0x200**	**0x102**	**0x300**	**0x103**	**0x201**	**0x102**	**0x301**	**0x103**
Fully Ass.	M	H	M	H	M	H	H	H	H	H
Direct Mapp.	M	H	M	M	M	M	M	M	M	M

Calculation of the First Three Memory Accesses:

1. **Access to `0x100`:**
   - **Fully Associative Cache:** Since the cache is initially empty, this is a **miss**. The block containing `0x100` is loaded into the cache.
   - **Direct-Mapped Cache:** The cache line index is calculated using the lower 8 bits of the address (after accounting for block offset). Assuming a block size of 4 words (i.e., 2 bits for block offset), the index is:

$$\text{Index} = \left(\frac{\texttt{0x100}}{4}\right) \mod 64 = 64 \mod 64 = 0$$

   This is a **miss**, and the block is loaded into cache line 0.

2. **Access to `0x101`:**
   - **Fully Associative Cache:** The block containing `0x100` also contains `0x101` (since they are in the same block). This is a **hit**.
   - **Direct-Mapped Cache:** `0x101` maps to the same cache line as `0x100`. It's within the same block already loaded, resulting in a **hit**.
3. **Access to `0x200`:**
   - **Fully Associative Cache:** This address is not in the cache, leading to a **miss**. The block containing `0x200` is loaded into the cache.
   - **Direct-Mapped Cache:** The index is calculated as:

$$\text{Index} = \left(\frac{\texttt{0x200}}{4}\right) \mod 64 = 128 \mod 64 = 0$$

     `0x200` maps to cache line 0, the same as `0x100`. This causes a **conflict miss** as it replaces the block containing `0x100` and `0x101`.

Why the Direct-Mapped Cache Misses After `0x101`:
In a direct-mapped cache, each block of memory maps to a single cache line based on its index. Addresses `0x100`, `0x200`, `0x300`, etc., share the same index due to their address patterns:

- `0x100` index: $(256/4) \mod 64 = 64 \mod 64 = 0$
- `0x200` index: $(512/4) \mod 64 = 128 \mod 64 = 0$
- `0x300` index: $(768/4) \mod 64 = 192 \mod 64 = 0$

This means they all map to cache line 0. After `0x101`, every new access (`0x200`, `0x300`, etc.) conflicts with previous ones, causing the cache line to be continually overwritten. As a result, subsequent accesses that might benefit from cache hits suffer misses instead.
In contrast, the fully associative cache doesn't have fixed mappings. It can store blocks anywhere, avoiding these conflicts and maintaining higher hit rates after the initial misses.

11.7 Associativity

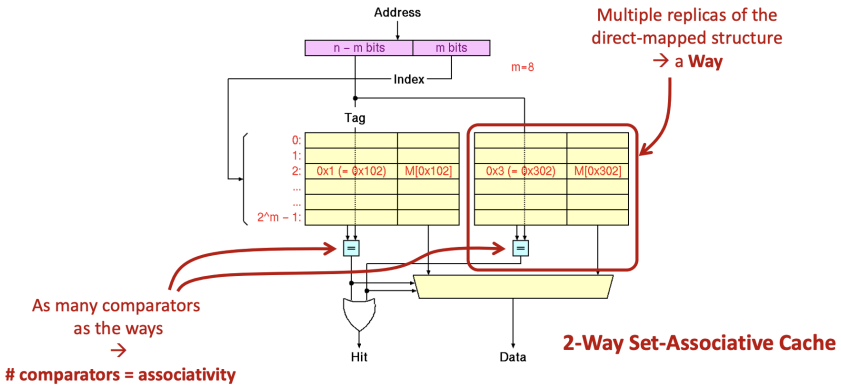
The associativity of a cache defines the number of possible locations in the cache where a single piece of data can be stored. It significantly influences the probability of **aliasing** (multiple memory addressing maps to the same cache data, causing potential coherency issues or conflicts) in cache memory. For now we've seen :

- **Fully Associative:** Each word can be stored in any line of the cache. This implies that the associativity equals the total number of lines in the cache.
- **Direct Mapped:** Each word is mapped to exactly one line in the cache. Thus, the associativity in this case is 1.

but maybe there is a better one...

11.7.1 Set-Associative Cache

A **set-associative cache** combines features of both direct-mapped and fully associative caches. It divides the cache into multiple sets, where each set contains a fixed number of blocks known as *ways*. A typical configuration is a *k-way set-associative cache*, where each set can hold up to k blocks.



Address Breakdown

The memory address is divided into three fields:

- **Tag:** Identifies whether a particular block is in the cache.
- **Index:** Specifies the set to which the block belongs.
- **Offset:** Specifies the location within a block.

Structure and Operation

Each set operates as a miniature associative cache, allowing multiple blocks to reside in the same set. The operation involves:

- **Comparison:** A comparator is used for each way to check if the tag of a block in the cache matches the tag from the memory address.
- **Hit Detection:** If any comparator signals a match, a *cache hit* occurs, and the corresponding data is retrieved.
- **Miss Handling:** If no match is found, a *cache miss* occurs, and the block is fetched from the next level of memory and stored in the appropriate set.

2-Way Set-Associative Cache

A common configuration is the **2-way set-associative cache**, where each set contains two blocks. This design reduces conflicts compared to direct-mapped caches while maintaining lower complexity than fully associative caches.
Key Features:

- Multiple replicas of the direct-mapped structure form the *ways*.
- The number of comparators equals the number of ways (#Comparators = Associativity).
- Flexible block placement within a set reduces conflict misses.

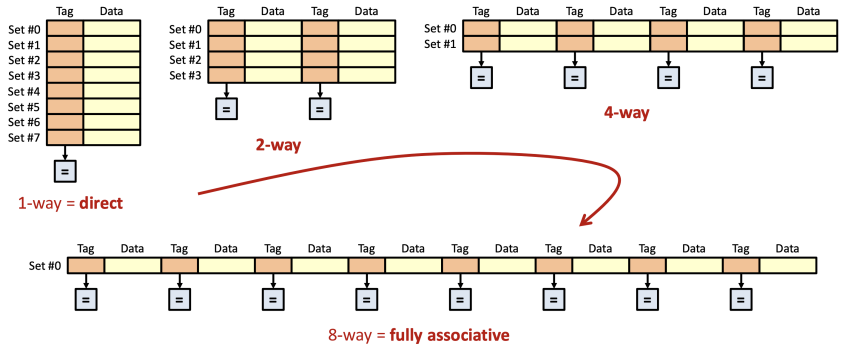
Example: Index Taken in One Way and Not in Another

In a *2-way set-associative cache*, the following scenario can occur:

- **First Way:** The index points to a block, but the tag stored in the block does not match the tag from the memory address. In this case, the block in the first way is ignored.
- **Second Way:** The index points to an empty block (i.e., the block is not taken). Since there is no valid data in this block, no tag comparison is performed.

When neither way results in a tag match or valid data, a *cache miss* occurs. The requested block is then fetched from the next memory level and stored in the appropriate set. Typically, a *replacement policy* (e.g., Least Recently Used, LRU) determines which block in the set will be evicted if the set is full.

11.7.2 A Continuum of Possibilities

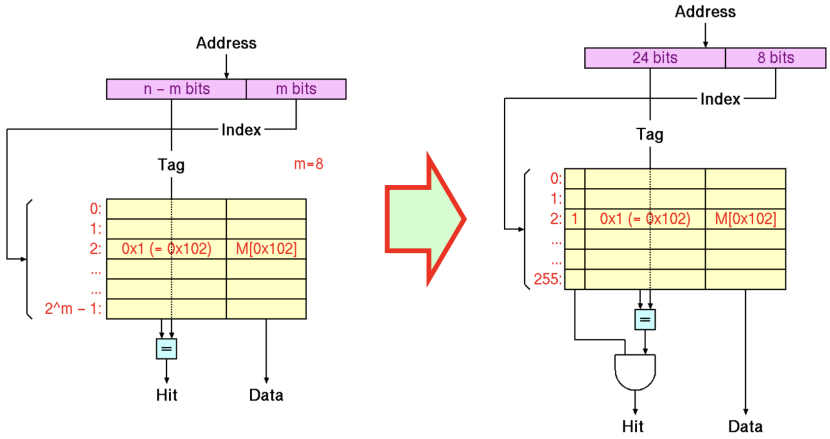


- **Direct-Mapped Cache (1-way):** Each memory block is mapped to exactly one cache line. This simple design ensures fast access but suffers from higher conflict misses.
- **Set-Associative Cache (N-way):** Memory blocks are mapped to a set of cache lines. Each set can hold N blocks, reducing conflict misses while introducing moderate hardware complexity. Common examples include:
  - *2-way set associative:* Each set contains 2 cache lines.
  - *4-way set associative:* Each set contains 4 cache lines.
- **Fully Associative Cache (8-way):** Any memory block can be placed in any cache line. This maximizes flexibility and minimizes conflict misses but requires costly associative hardware for lookup.

The choice of associativity directly impacts the cache's performance and design trade-offs. A fully associative cache offers the lowest miss rate at the expense of higher power and latency, while a direct-mapped cache provides faster access but is prone to frequent conflicts.

11.7.3 Cache Validity

Initial State of Cache: When a cache is initialized, its content is considered garbage. To ensure proper operation, each cache line includes a *Valid Bit*. This special bit indicates whether the data in a specific cache line is meaningful or not.



Valid Bit Mechanism:

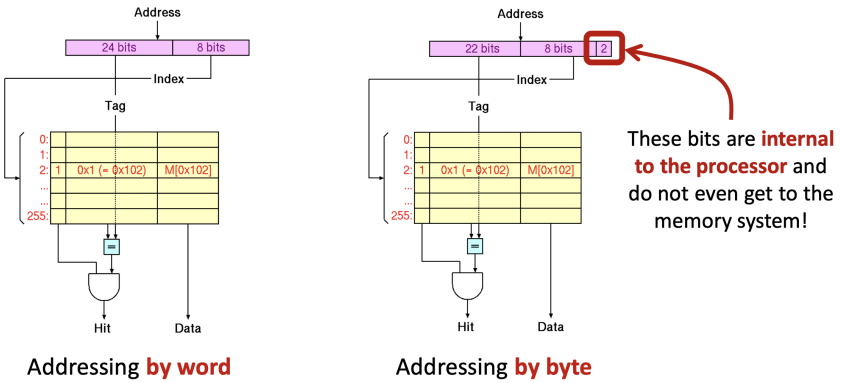
- The *Valid Bit* is set to `0` at reset, indicating that the cache line is invalid.
- When valid data is written into the cache, the *Valid Bit* is updated to `1`.

Cache Hit Check: To determine if a memory address is present in the cache:

1. Compare the *Tag* from the memory address with the *Tag* stored in the indexed cache line.
2. Ensure the *Valid Bit* of the cache line is set to `1`.
3. If both conditions are met, a *Cache Hit* occurs, and the corresponding data is retrieved.

11.7.4 Addressing by Byte vs Addressing by Word

When addressing is **by byte** and the word size is 2^n bytes, the n least-significant bits of the address represent the byte offset. These bits are considered **irrelevant** for accessing the memory system as they are internal to the processor.



- In **byte addressing**, the full address includes these n bits, but they do not participate in determining the memory location at the cache level. For example, in the diagram, the 8 least-significant bits are used to compute the offset within a word.
- In **word addressing**, the memory system processes only the address bits beyond the n least-significant bits. The internal processor uses the irrelevant bits for internal data handling but they are not sent to the memory system.

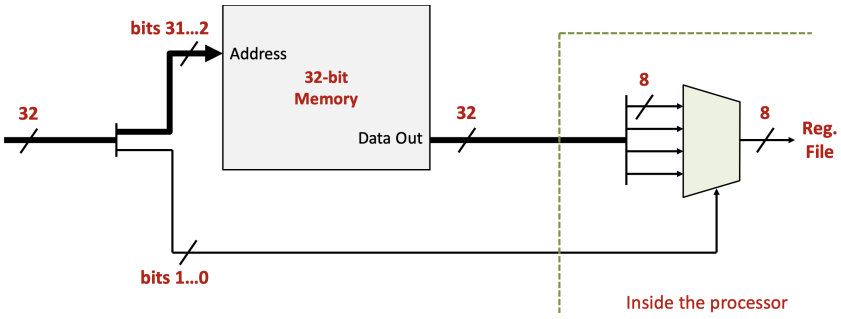
Key Differences:

1. *Byte Addressing:* All address bits are used to identify individual bytes, ensuring fine-grained access.
2. *Word Addressing:* The address bits are divided, with the least-significant bits used only internally, simplifying memory system access.

This distinction between addressing schemes impacts cache design and performance, particularly in systems with varying word sizes.

11.8 Loading Bytes(lb)

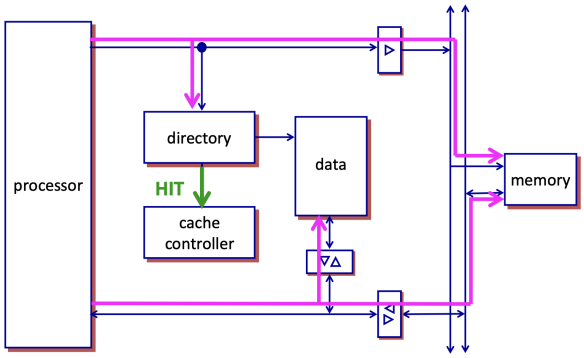
To load a specific byte from memory into a register, the architecture employs a combination of memory addressing and multiplexing. The process involves the following components:



- **Address Calculation:** The 32-bit memory address is split into two parts:
  - `bits 31...2` specify the base address of the 32-bit memory word.
  - `bits 1...0` identify the specific byte within the word.
- **Memory Access:** The calculated address accesses a 32-bit word from memory. The word is then output as `Data Out`, a 32-bit value.
- **Byte Selection:** A multiplexer extracts the required 8-bit byte from the 32-bit word based on the value of `bits 1...0`. The multiplexer outputs the selected byte to the processor's register file.
- **Register File Update:** The selected byte is written to the appropriate register within the processor.

11.8.1 Write Hit

A **write hit** occurs when the data to be written is found in the cache. The process involves the following key components and considerations:



- **Processor Request:** The processor initiates a write operation to a specific address.
- **Cache Lookup:** The *directory* checks if the requested address is present in the cache. Upon finding a match, a **hit** signal is sent to the *cache controller*.
- **Data Update:** The cache controller updates the data in the cache. The operation ensures that the cache remains consistent with the intended value.

An important decision at this stage is whether the data should also be written to the *main memory*. This introduces two potential strategies:

1. **Write-Through:** The updated data is immediately written to both the cache and the main memory, ensuring consistency.
2. **Write-Back:** The data is written only to the cache, with changes propagated to the main memory later when the cache block is evicted.

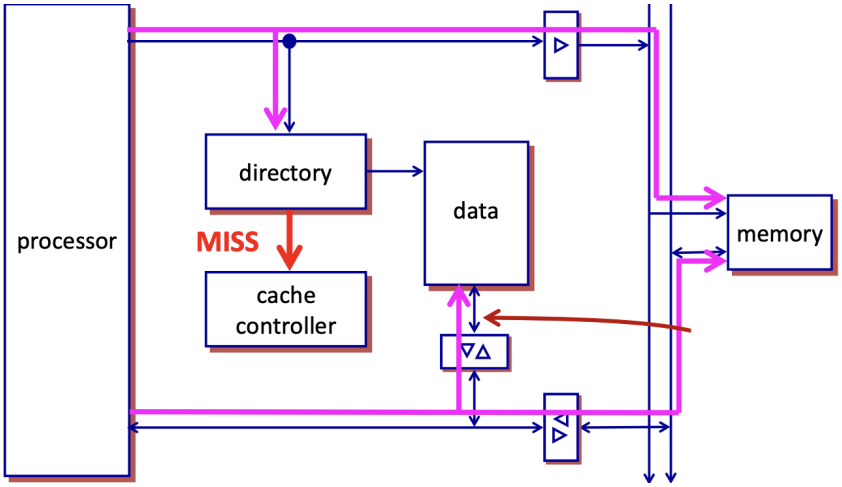
This decision significantly impacts performance and consistency, balancing speed with data reliability. The choice between **write-through** and **write-back** is typically determined by the system's design requirements.

Write Policies in Cache Memory

When a write operation is performed, the system must decide how to handle the update in the cache and main memory. Two primary write policies are used:

- **Write-Through:**
  - Data is immediately written to both the cache and the main memory.
  - **Advantages:** Simplifies data consistency between cache and memory.
  - **Disadvantages:** Can lead to increased memory traffic, keeping the memory and buses busy unnecessarily.
- **Write-Back (or Copy-Back):**
  - Data is only updated in the cache. The main memory remains unchanged until the cache block is evicted.
  - Requires a **Dirty Bit** to indicate whether the cached data has been modified.
  - When a dirty cache line is evicted, it must first be written back to the main memory.
  - **Advantages:** Reduces memory write operations, improving performance.
  - **Disadvantages:** The main memory may become temporarily inconsistent with the cache.

11.8.2 Write Miss in Cache Memory



A **write miss** occurs when the data to be written is not found in the cache. The system must determine how to handle this scenario. The process includes the following steps:

- **Cache Lookup:** The *directory* is checked for the requested address. If the address is not present, a **miss** signal is sent to the *cache controller*.
- **Memory Update:** The data is written directly to the main memory. This operation ensures that the memory reflects the most recent changes.
- **Cache Allocation:** A decision is made whether to allocate space in the cache for the new data. Two strategies can be employed:
  1. **Write-Allocate:** The data is loaded into the cache after being written to memory, ensuring faster future access.
  2. **No-Write-Allocate:** The data is written only to the memory, and the cache remains unchanged. This approach reduces cache pollution.

The choice between **write-allocate** and **no-write-allocate** is influenced by workload characteristics and system design, balancing cache utilization and memory performance.

Allocation Policies

Allocation policies define how data is managed in the cache during a write miss. The two primary policies are:

Write-Allocate:

- On a write miss, the data is also placed in the cache.
- This approach is simple and straightforward.
- Requires fetching the block of data from memory before writing.
- May lead to unnecessary cache pollution if the processor writes data that it will never read back.

Write-Around (or Write-No-Allocate):

- On a write miss, the data is written directly to memory, bypassing the cache.
- If the processor later loads from the same address, it results in a read miss.

11.9 Summary

11.9.1 The "3 Cs" of Caches

Cache misses are categorized into three types, commonly referred to as the "3 Cs":

Compulsory Misses:

- These misses occur during the first reference to a block, even in an infinitely large fully-associative cache.
- Also known as *cold-start misses* or *first-reference misses*.

Capacity Misses:

- Occur when the cache is unable to hold all the required blocks due to its limited capacity.
- These misses happen even in a fully-associative cache.

Conflict Misses:

- These occur when multiple blocks compete for the same cache set due to limited associativity.
- Arise in set-associative or direct-mapped caches.

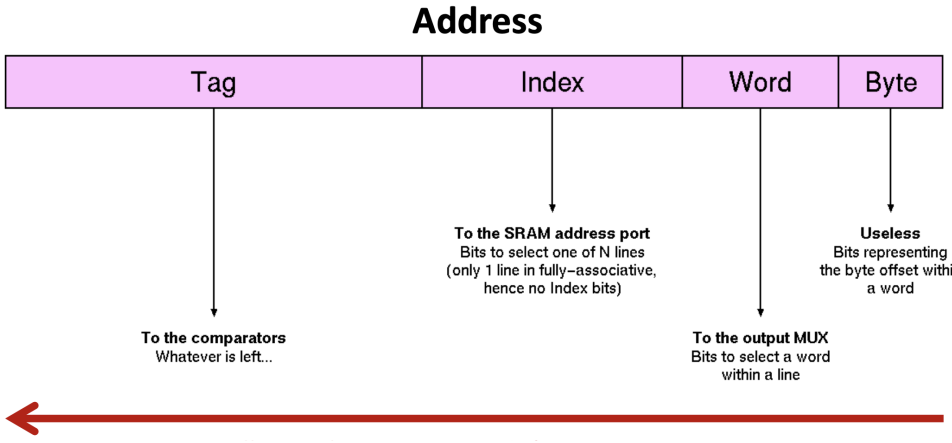
Understanding these types of misses is critical for identifying the source of limited cache performance.

11.9.2 Summary of Cache Features

This section provides an overview of the key features and attributes of caches:

- **Cache Size:** Total data storage capacity (usually excludes tags, valid bits, dirty bits, etc.).
- **Addressing:** Data can be addressed by byte or word.
- **Line or Block Size:** Specifies the size of cache lines or blocks in bytes or words.
- **Associativity:** Can be fully-associative, k-way set-associative, or direct-mapped.
- **Replacement Policy:** Governs how cache blocks are replaced (e.g., LRU, FIFO, random). This is not applicable for direct-mapped caches.
- **Write Policy:** Determines how writes are handled (e.g., write-through or write-back).
- **Allocation Policy:** Defines cache behavior on a write miss (e.g., write-allocate or write-around).

Cache Addressing

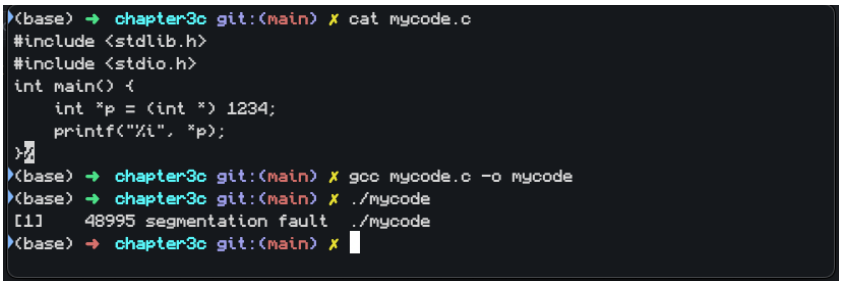


Chapter 12

Part III(a) - Memory Hierarchy - Virtual Memory - W.7.2

12.1 Segmentation Fault: Understanding the Cause

Segmentation faults occur when a program attempts to access a memory location that is either invalid or restricted. Consider the following C code snippet:



In this example:

- The pointer `p` is assigned the value `1234`, which is an arbitrary and invalid memory address.
- When the program attempts to dereference `p` using `*p` to access the value at memory address `1234`, it triggers a segmentation fault because the program does not have permission to access this memory.

Why This Happens:

- Modern operating systems enforce memory protection, disallowing access to memory that the program does not explicitly allocate or own.
- Hardcoding arbitrary addresses, like `1234`, is unsafe and violates these protections.

Assembly Analysis: The following assembly instructions illustrate how the invalid memory access occurs:

- `li t0, 1234` – Load the immediate value `1234` into register `t0`.
- `lw a1, 0(t0)` – Attempt to load a word from address `1234`.
- This results in a segmentation fault because address `1234` is not valid or accessible.

We need to be able to protect memory and prevent such invalid accesses. This is where memory protection mechanisms come into play.

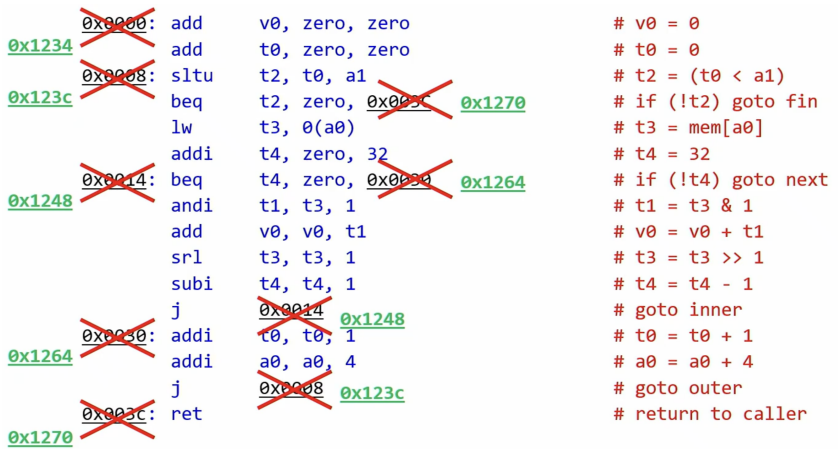
12.1.1 Overview - Problems to Solve

Three main problems need to be addressed:

1. **Memory Protection:** How can we protect memory so that each program (or process) running simultaneously in the system can only access its own data? How can processes be isolated from each other?
2. **Insufficient Main Memory:** What happens if the main memory (DRAM) is not sufficient for the execution of a program? Can we utilize the disk to address this limitation? If so, how?
3. **Running Multiple Programs:** How can we run several programs (processes) simultaneously? How can multiple programs be loaded into memory efficiently, and where should they be stored?

12.2 Relocation at Load Time

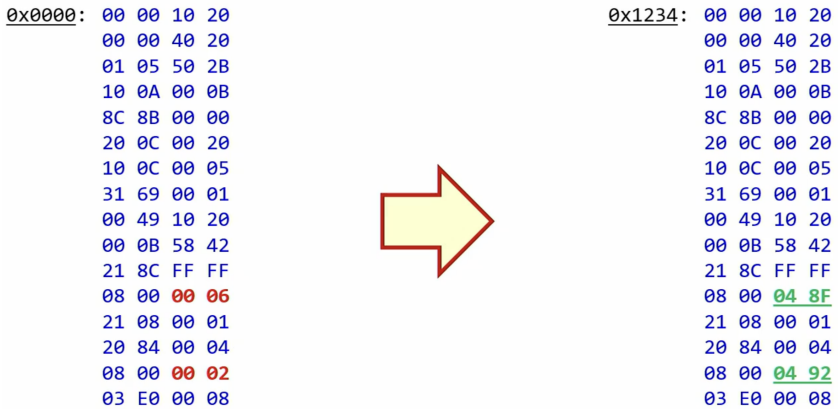
Relocation at load time is a fundamental memory management process that adjusts memory addresses to align a program's instructions with the actual memory layout during execution. This technique ensures that all address references within the program are accurate, allowing it to run seamlessly in its allocated memory space.
Address Adjustments: During the relocation process, address references within instructions are updated from placeholder values (e.g., `0x0000`) to their correct memory addresses (e.g., `0x1270` or `0x1248`). This adjustment applies to all memory references, including branches and jumps such as `beq` or `j`, which depend on accurate target addresses to function correctly.



By performing these updates, the program adapts to the memory layout without requiring additional runtime computations. This process, though effective, operates primarily at the binary level rather than at the assembly code level.

Binary-Level Adjustments

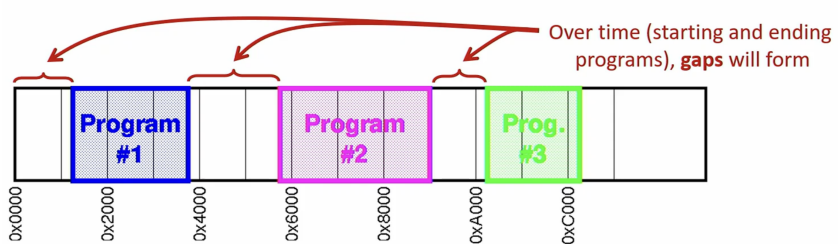
Relocation at load time involves modifying the binary code directly. Specific fields within the machine instructions are updated based on relocation tables, which specify the exact memory addresses to be adjusted. These tables play a crucial role in streamlining the relocation process and ensuring correctness.



For instance, consider a binary program initially loaded at a base address of `0x0000`. During relocation, placeholders in the binary instructions are replaced with actual memory addresses derived from the relocation table, as illustrated in the figure. This guarantees that memory references resolve correctly during execution.

Memory Utilization and Limitations

While relocation at load time simplifies memory address management, it is not without its drawbacks. As programs are loaded and terminated, gaps in memory may form, leading to inefficient utilization. This fragmentation becomes particularly problematic in systems with limited memory or dynamic memory allocation needs.

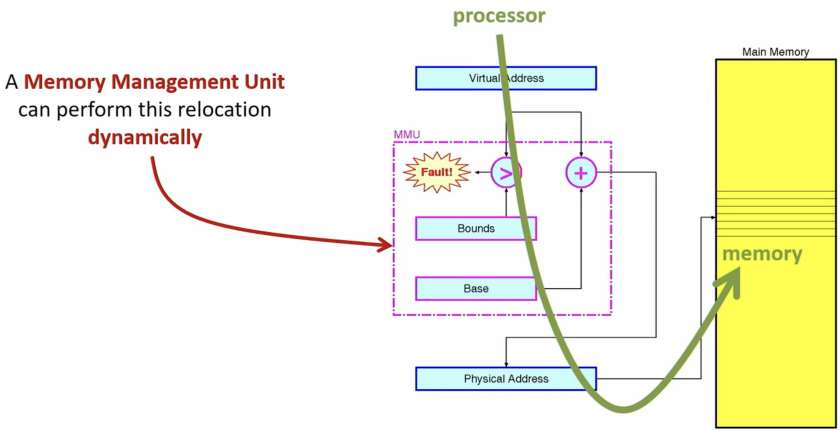


Limitations:

- *High Overhead:* Relocation requires considerable computational effort at load time to allocate and adjust memory segments accurately.
- *Inflexibility:* Once memory is allocated, it cannot be dynamically reconfigured, making it challenging to adapt to changing program requirements.
- *Fragmentation Constraints:* Memory fragmentation caused by terminated programs can prevent loading a new program if its size exceeds the largest available gap, even when total free memory is sufficient (garbage collector...).

12.2.1 Relocation in Hardware: Base and Bounds MMU

Memory relocation is an essential process in modern computer systems to map virtual addresses to physical addresses. The **Base and Bounds Memory Management Unit (MMU)** facilitates this process dynamically by ensuring secure and efficient address translation.

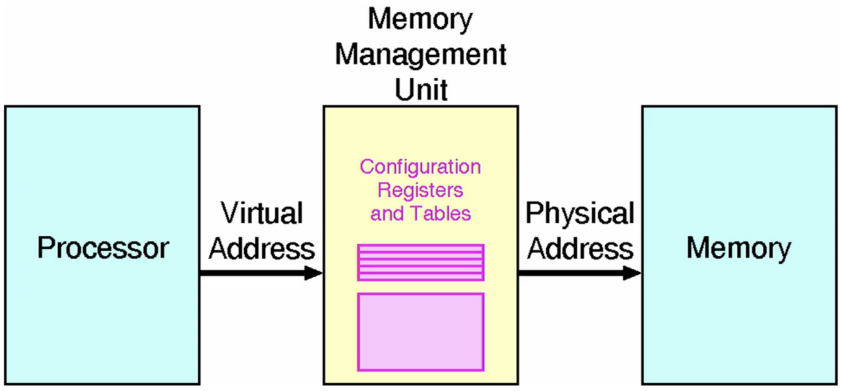


- **Base Register:** Holds the starting physical address of the process's memory.
- **Bounds Register:** Defines the limit or size of the memory allocated to the process.
- **Translation Process:**
  1. The processor generates a *virtual address*.
  2. The MMU checks if the virtual address exceeds the value in the *Bounds Register*.
     - If it exceeds, a **fault** is raised, preventing illegal memory access.
     - Otherwise, the MMU adds the *Base Register* value to the virtual address, producing the *physical address*.
  3. The *physical address* is used to access the main memory.

This mechanism ensures process isolation and protects the system against unauthorized memory access, as only addresses within the defined bounds can be accessed. The dynamic nature of this relocation is key to supporting multitasking and efficient memory utilization.

12.2.2 Memory Management Unit (MMU)

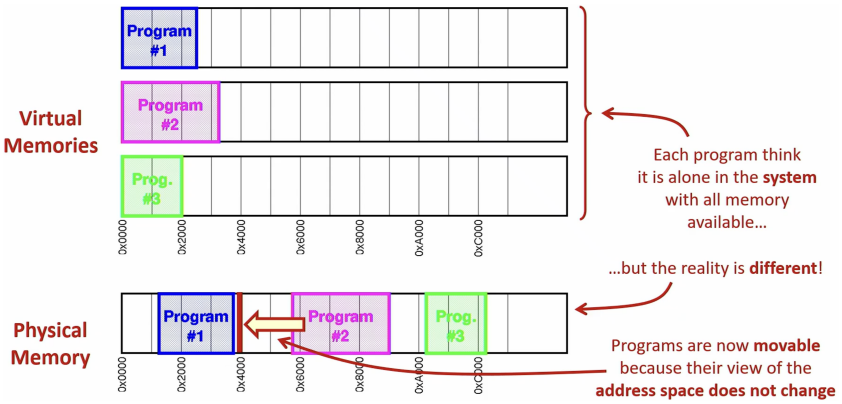
The **Memory Management Unit (MMU)** is a critical hardware component that facilitates the translation of *virtual addresses* generated by the processor into *physical addresses* used by the memory. This process is essential for efficient memory management in modern computer systems.



- **Virtual Address:** Generated by the processor, representing a logical view of memory.
- **Physical Address:** The actual address in the main memory where data is stored.
- **Key Components:**
  1. *Configuration Registers:* Store information such as base addresses, bounds, and page tables.
  2. *Translation Tables:* Facilitate the mapping of virtual to physical addresses.
- **Process Overview:**
  1. The processor generates a *virtual address*.
  2. The MMU uses its configuration registers and translation tables to map the virtual address to a physical address.
  3. The mapped physical address is used to access data in the main memory.

12.2.3 Program Relocation with Virtual Memory

Virtual memory provides a powerful abstraction that decouples a program's logical memory view from the physical memory of the system. This flexibility is achieved by isolating programs from the underlying physical memory addresses, allowing dynamic relocation of programs without affecting their execution.

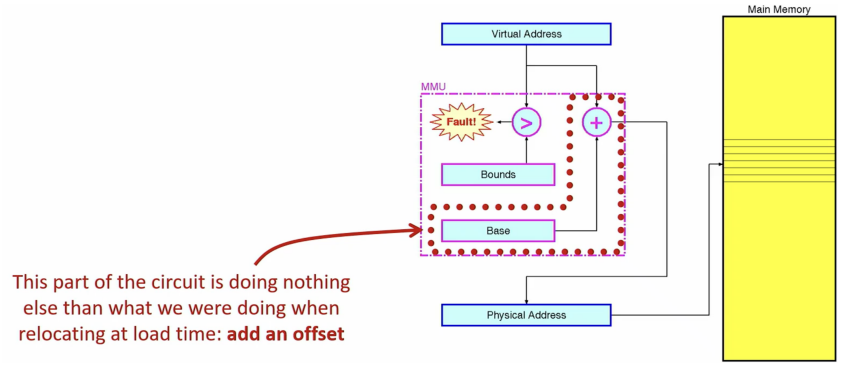


- **Independent Address Spaces:** Each program is assigned its own virtual address space, creating the illusion that it has exclusive access to the entire memory. The program is unaware of its actual location in physical memory.
- **Program Relocation:** Since a program only interacts with its virtual address space, the operating system can freely move its physical location in memory as needed. This movement, known as *relocation*, can occur during execution or at load time.
- **Benefits of Relocation:**

– *Efficient Memory Use:* Programs can be compacted to free up contiguous physical memory for other processes.
– *Load Balancing:* Active programs can be repositioned to optimize memory access speed or reduce fragmentation.
– *Seamless Execution:* Since the memory translation is handled by the hardware (e.g., the Memory Management Unit, MMU), the program remains unaware of any changes in its physical location.

12.3 Relocation in Hardware: Base and Bounds MMU

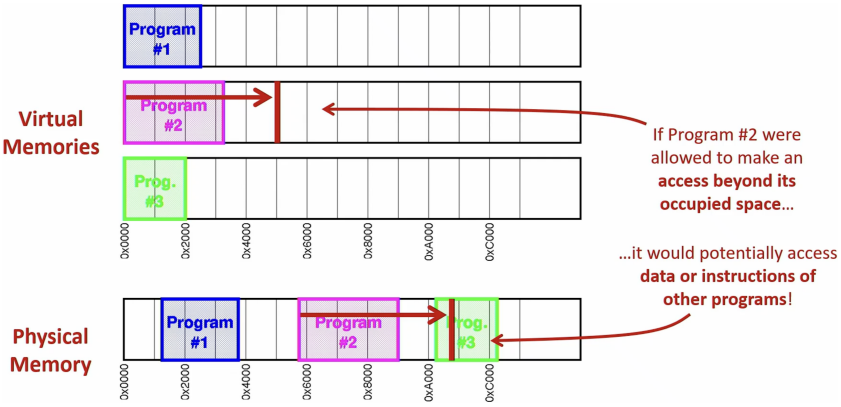
The Base and Bounds MMU (Memory Management Unit) is a hardware mechanism designed to facilitate memory relocation. It operates by performing two key actions on a virtual address provided by a process:



- **Bounds Check:** The virtual address is compared with the `Bounds` register to ensure it is within the allowable range. If the virtual address exceeds the bounds, a fault is triggered, preventing unauthorized access.
- **Offset Addition:** If the virtual address is valid, it is added to the value in the `Base` register. This offset addition translates the virtual address into a physical address, which is then used to access the main memory.

12.3.1 Preventing Overreach in Virtual and Physical Memory

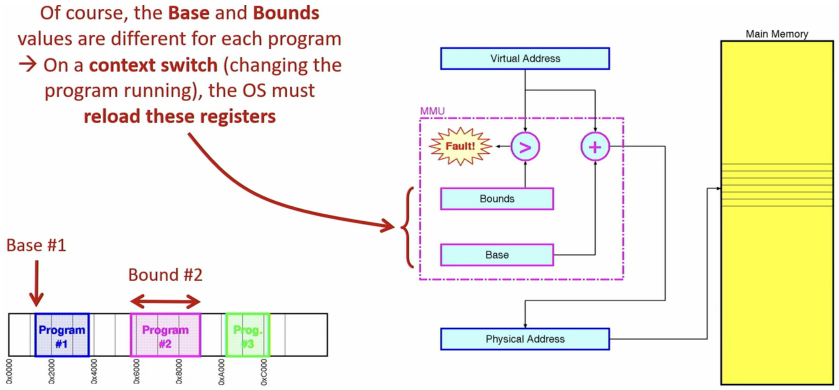
In systems using virtual memory, it is critical to ensure that programs remain within their allocated address spaces. If a program accesses memory beyond its assigned virtual boundaries, it risks overlapping with another program's memory. This can lead to severe security and stability issues.



- **Virtual Memory Overreach:** Each program is given its own virtual address space. However, if a program attempts to access an address outside its bounds, it could inadvertently access data or instructions from another program. This can compromise the integrity of both programs.
- **MMU Checks:** To prevent such overreach, the Memory Management Unit (MMU) enforces strict bounds checking. It ensures that all memory accesses fall within the allowed range defined by the program's base and bounds registers. If a memory access is outside these bounds, the MMU generates a fault, preventing the access.
- **Physical Memory Isolation:** Even though programs use virtual addresses, these are translated to physical addresses by the MMU. Proper isolation ensures that physical memory regions assigned to different programs do not overlap, maintaining system stability.

This mechanism of bounds checking and fault generation safeguards against unintended interactions between programs and ensures that no program can overwrite or access another program's data.

12.3.2 Base and Bounds MMU



The Base and Bounds mechanism in the MMU defines the memory allocation for a process as follows:

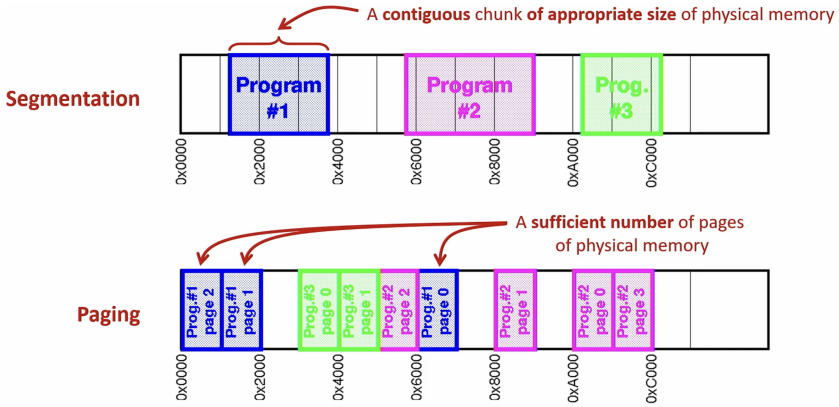
- Base: The starting address of the process's allocated memory.
- Bounds: The upper limit or size of the memory allocated to the process.

12.4 Needs of a Multiprogrammed System

Multiprogrammed systems require efficient handling of memory and process management to ensure reliability and performance. The key requirements are as follows:

- **Relocation:** Programs must be written without prior knowledge of their location in memory. This ensures flexibility when allocating memory during execution.
- **Protection:** Programs are restricted to access only their own data, safeguarding against interference from other programs. However, this protection mechanism can sometimes be crude, as it often limits each program to a single chunk of memory.
- **Space Management:** When several programs run simultaneously, memory shortages may arise. Effective space allocation strategies are needed, which may involve techniques such as garbage collection and moving programs or data to optimize memory usage.

12.5 Segmentation and Paging



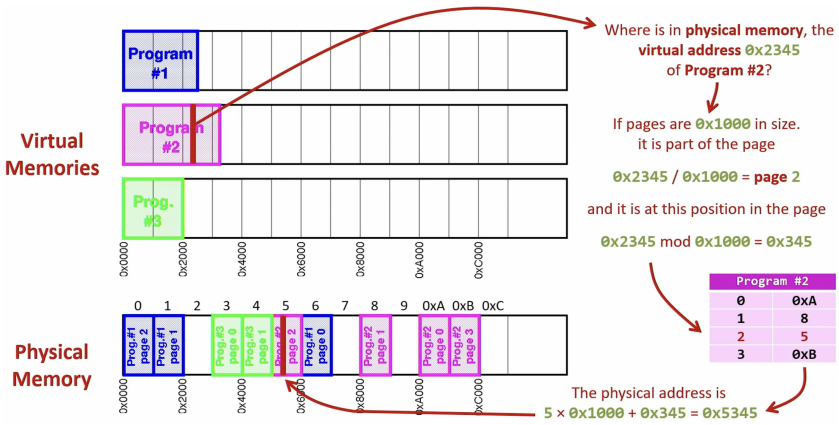
Segmentation: Segmentation, an extension of the Base and Bounds technique, allows memory to be split exactly as needed by each program. Key characteristics include:

- Arbitrary starting point of a memory block.
- Arbitrary length of memory blocks.
- Multiple blocks can be allocated per application.

Paging: Paging divides memory into equal-sized small blocks (e.g., 4–64 KiB) and assigns as many blocks as required to each program. This ensures uniformity in memory allocation.

12.5.1 How do we Translate Now?

Now we need to translate virtual addresses to physical addresses (with our new paging constraints). This process involves the following steps:



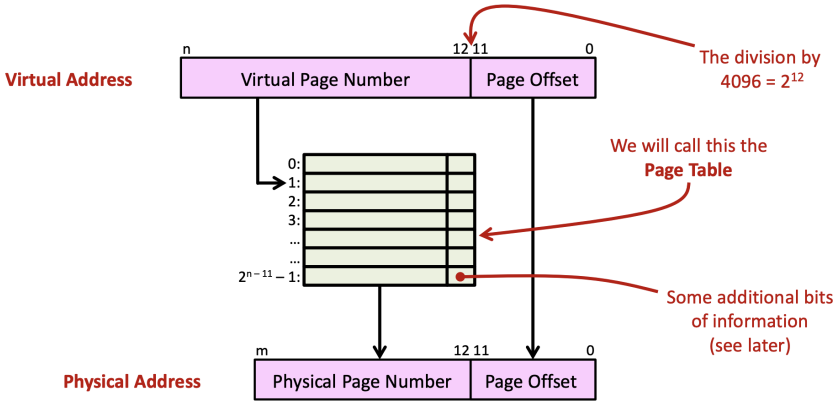
For example,if the system needs to determine the physical memory location corresponding to the virtual address `0x2345` of **Program #2**. The translation process follows these steps:

1. Identify the **page number** and **offset** within the virtual address. Assuming page size is known, `0x2345` translates to a specific page and offset.
2. Consult the **page table** for **Program #2**, which maps virtual pages to physical pages. For instance:

   `Program #2 Page 0` → `Physical Page 8`, `Page 1` → `Physical Page 9`

3. Use the mapping to locate the physical address. In this example, the virtual address resides in `Page 0`, which maps to `Physical Page 8`. Combining the physical page base address with the offset yields the final physical memory address.

Sumarized, to find the physical address, we need to extract the virtual page number and the page offset from the virtual address. The virtual page number is used to look up the corresponding physical frame number in the page table. Finally, the physical address is computed using the formula:

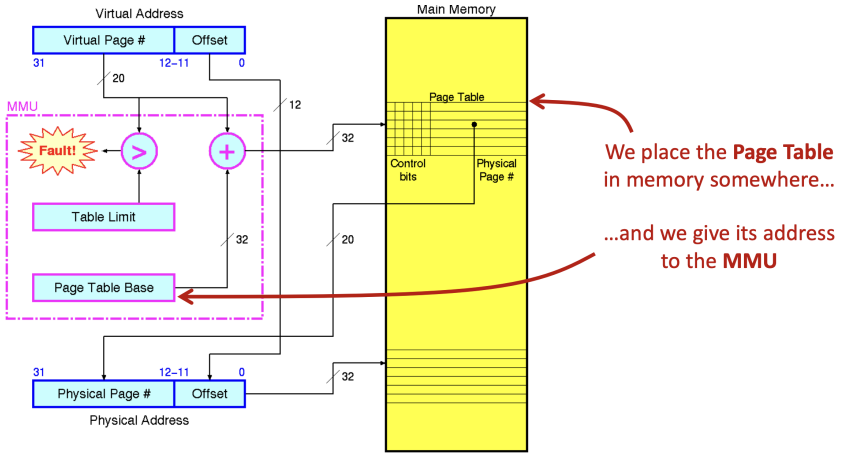


$$\text{Physical Address} = (\text{Physical Frame Number} \times \text{Page Size}) + \text{Page Offset}$$

where the page offset is directly derived from the virtual address, and the physical frame number is obtained from the page table. The page size is often a power of 2 (for example, $4\,\text{KB} = 2^{12}$), which makes extracting the page offset straightforward as it corresponds to the lower-order bits of the virtual address.

12.5.2 Virtual Adress Translation in a Paged MMU

In a paged MMU, the virtual address generated by the processor is translated into a physical address using the page table stored in memory.

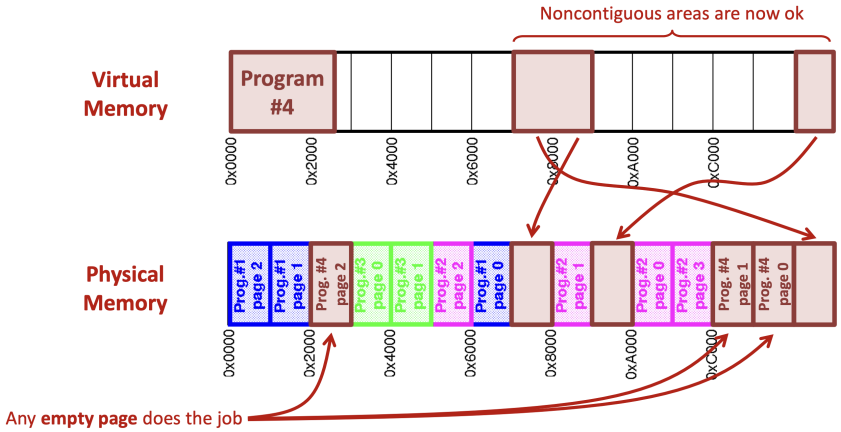


Page Table: The page table, residing in main memory, contains:

- *Control Bits:* Indicate the validity of a page and access permissions.
- *Physical Page Numbers:* Map virtual pages to physical pages.

12.5.3 Memory Allocation is Easy Now

Virtual memory systems simplify memory allocation by allowing noncontiguous physical memory to be mapped to contiguous virtual memory addresses. This enables efficient utilization of physical memory without requiring large, contiguous blocks.



Virtual Memory: Each program operates in its own virtual address space, making it unaware of the physical memory layout. Virtual addresses are mapped to physical addresses using a page table.

Physical Memory: Physical memory is divided into fixed-size blocks called *pages*. Any empty page in physical memory can be allocated to a program's virtual page.

Advantages:

- Programs can use noncontiguous memory regions without manual intervention.
- Memory fragmentation is minimized since any available physical page can be used.
- Programs are isolated from one another, enhancing security and stability.

In the diagram above, Program #4's virtual memory is mapped to noncontiguous pages in physical memory (e.g., 0x2000, 0x8000, and 0xC000). This flexibility ensures efficient allocation.

The use of virtual memory significantly enhances system performance and simplifies memory management by abstracting physical memory complexities.

12.5.4 Page Tables and Their Size

Page tables in virtual memory systems can grow significantly in size, especially in cases with large memory spaces. For instance, a memory size of 64 GiB with 4 KiB pages requires 2^{24} entries, amounting to approximately 64 MiB of space.

Challenges

For programs that utilize only a few megabytes, the majority of these entries remain empty, leading to inefficient memory usage.

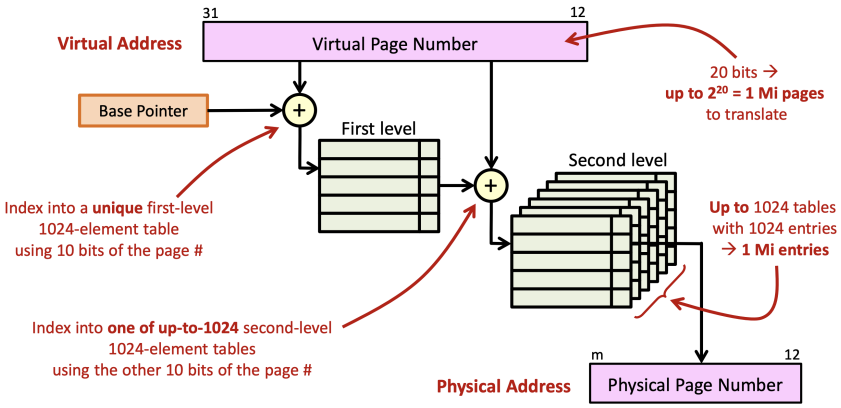
Solutions

Several approaches exist to mitigate this inefficiency, including:

- **Hashed Tables:** An alternative structure to reduce unused entries.
- **Paged Segmentation:** A hybrid approach to manage memory.
- **Multilevel Page Tables:** A hierarchical design to handle sparse page tables efficiently.

12.5.5 Multilevel Page Tables

Multilevel page tables are a hierarchical solution to reduce memory overhead caused by storing a single large page table. This structure is particularly useful in virtual memory systems with large address spaces.



A virtual address is divided into three main parts:

- **Virtual Page Number:** Determines the index within the page tables.
- **Page Offset:** Specifies the exact byte within the page.

The hierarchical organization involves:

1. A **first-level page table**, indexed by the higher-order bits of the virtual page number. This table points to second-level page tables.
2. **Second-level page tables**, indexed by the remaining bits of the virtual page number. These map to the physical page number.

Advantages:

- **Reduced memory usage:** Only necessary parts of the page tables are stored in memory.
- **Scalability:** Easily adapts to varying address space sizes.

Key Steps in Address Translation:

1. Use the first-level index to locate the appropriate second-level page table.
2. Use the second-level index to identify the physical page number.
3. Combine the physical page number with the page offset to obtain the physical address.

Multilevel page tables strike a balance between efficiency and memory overhead, making them a practical choice in modern operating systems.

Chapter 13

Comparch II - Part IV(a) - Instruction Level Parallelism Performance

So far, we've only been building our processor, now it's about performance...

13.1 What is Performance ?

Now what do we mean by performance ?, we need a metric to measure performance.

Does processory frequency matter ? *Is it better an Intel Core i7-7700K at 4.2 GHz or an AMD Ryzen 5 5600X at 3.7 GHz?*

Memory Speed ? Cache efficiency ?

Is it better to have 8 MiB of 4-way set-associative cache or 16 MiB of direct mapped cache?

Is it better to have three levels of overall smaller caches or two levels of overall bigger caches?

13.1.1 Elapsed Time, CPU Time, ...

In reality, none of this matters in it self. What matters is the time it takes to perform a job a user needs.

```
[110]icvm0100> time latex mypaper >& /dev/null
0.79u 0.17s 0:01.20 80.0%
[111]icvm0100>
```

- **Elapsed Time:** The total time taken for the job to complete, measured from start to finish. (e.g., 1.20 seconds)
- **System CPU Time:** The CPU time used by the operating system to execute instructions on behalf of the program. (e.g., 0.17 seconds)
- **User CPU Time:** The CPU time used to execute instructions for the program itself. (e.g., 0.79 seconds)
- 80.0% of the Elapsed Time ($0.96\,s/1.20\,s$) was spent on the job (the rest might be spent on system I/O, other jobs, and other users.)
- **Note:** *User CPU Time + System CPU Time* $\neq$ *Elapsed Time*: The processor spent 0.96 seconds executing for the program, but the overall job took 1.20 seconds to complete.

13.1.2 Relative Performance

Speedup
The speedup metric quantifies how much faster system X is compared to system Y. It is defined as:

$$\text{Speedup} = \frac{\text{Performance}_X}{\text{Performance}_Y} = \frac{\text{Execution Time}_Y}{\text{Execution Time}_X}$$

Common Performance Indices Common benchmarks used to measure the speedups of systems relative to a standard system include:
SPEC CPU (a classic CPU performance benchmark), **Geekbench** (a comprehensive cross-platform benchmark), **Cinebench** (a benchmark focusing on rendering performance), **LinPack HPL** (a high-performance computing benchmark), and **EEMBC ("Embassy") CoreMark** (dedicated to benchmarking embedded processors).

13.1.3 Relating Performance to Hardware Implementation

In hardware design, time is measured by the *clock period* or *cycle*.

Cycles per Instruction (CPI) and Instructions per Cycle (IPC)

- CPI: Average cycles needed per instruction

$$\text{CPI} = \frac{\text{Total Cycles}}{\text{Total Instructions}} = \frac{\text{Execution Time/Clock Period}}{\text{Total Instructions}}$$

- IPC: Average instructions executed per cycle

$$\text{IPC} = \frac{1}{\text{CPI}}$$

- Note: IPC ≤ 1 unless the processor can execute multiple instructions in parallel

13.1.4 Improving Performance

Performance is defined as the reciprocal of execution time:

$$\text{Performance} = \frac{1}{\text{Execution Time}}$$

By breaking down execution time, performance can be expressed as:

$$\text{Performance} = \frac{f_{\text{clock}}}{\text{Instruction Count} \cdot \text{CPI}} = \frac{f_{\text{clock}} \cdot IPC}{\text{Instruction Count}}$$

Where:

f_{clock} is the clock frequency.

CPI is the cycles per instruction.

Instruction Count is the total number of instructions executed.

To improve performance, several strategies can be employed:

1. **Increase Clock Frequency (f_{clock}):** Implement the processor using faster technology to achieve higher clock rates.
2. **Reduce CPI:** Simplify instructions (RISC architecture) to lower the cycles per instruction. However, this may require more instructions to perform the same task.
3. **Decrease Instruction Count:** Use fewer, more complex instructions (CISC architecture). This could increase the number of cycles per instruction or reduce the clock frequency.
4. **Execute Instructions in Parallel:** Increase instructions per cycle (IPC) through techniques like pipelining or parallel execution.

These trade-offs highlight the balance required when optimizing computer architecture for performance.

13.1.5 Factors Influencing Performance

Several factors can significantly impact system performance. Here are a few examples:

- **Instruction Count and the Compiler:**
  - The instruction count depends heavily on the compiler.
  - A well-designed instruction set that the compiler can use effectively (i.e., best instructions for the job) is more important than having a highly reduced or overly complex instruction set. Otherwise, the instruction count may become unnecessarily large.
- **Cycles Per Instruction (CPI) and Cache Performance:**
  - CPI is influenced by the efficiency of the cache.
  - As the overall code size increases, cache performance may degrade due to a higher number of cache misses.
- **Clock Cycle Speed and Memory Access:**
  - Faster clock cycles increase the demand for fetching instructions from memory.
  - As a result, the performance of the cache becomes more critical in maintaining overall efficiency.

13.1.6 What to Improve to Increase Performance

Amdahl's Law (Law of Diminishing Returns): The performance enhancement possible with a given improvement is limited by the amount the improved feature is used.
Typical Software Situation:
If a program spends 20% of its time in subroutine X, the maximum reduction in execution time achievable by optimizing X is 20%. This corresponds to a speedup of:

$$\text{Speedup} = \frac{1}{1 - 0.2} = 1.25$$

In a Processor:
If an instruction Y is used only 0.1% of the time, is it worth optimizing it? It is more practical to focus on optimizing instructions or operations used more frequently, such as those taking 20% of the time.
What often happens is that we start optimizing the most used instructions, but then we forget that once optimized, the less used instructions become the bottleneck. So, we need to start looking at the less used instructions and optimize them, and so on.

Look for where most of the time goes!

13.1.7 Benchmarks

Performance Indices:
Benchmarks such as SPEC CPU, Geekbench, Cinebench, LinPack HPL, and EEMBC ("Embassy") CoreMark require a **precise definition of the user job(s)** to be executed.
Benchmark Suites:
Serious **benchmark suites** consist of collections of large and representative user programs, spanning various areas of typical use. These suites are often agreed upon by manufacturers to ensure standardization.
Key Features:
Benchmark suites do not only define the programs (e.g., written in C, C++, FORTRAN, or Java), but also specify:

- How the programs should be compiled.
- What data should be used during execution.
- The conditions under which the programs should be run.

SPEC CPU2006 Integer Benchmarks

The SPEC CPU2006 benchmark suite evaluates the performance of computer processors by running a set of standardized workloads. Below is a comparison of integer benchmarks between SPEC CPU2000 and SPEC CPU2006. The benchmarks are categorized by their description, language, and reference time (RT).

	CPU2000			CPU2006		
Benchmark Description	**Integer**	**Lng**	**RT**	**Integer**	**Lng**	**RT**
GNU C compiler	176.gcc	C	1,100	403.gcc	C	8,050
Manipulates strings & prime numbers in Perl language	253.perlbmk	C	1,800	400.perlbench	C	9,766
Minimum cost network flow solver (combinatorial optimization)	181.mcf	C	1,800	429.mcf	C	9,120
Data compression utility	256.bzip2	C	1,500	401.bzip2	C	9,644
Data compression utility	164.gzip	C	1,400			
Video compression & decompression				464.h264ref	C	22,235
Artificial intelligence, plays game of Chess	186.crafty	C	1,000	458.sjeng	C	12,141
Artificial intelligence, plays game of Go				445.gobmk	C	10,489
Artificial intelligence used in games for finding 2D paths across terrains				473.astar	C++	7,017
Natural language processing	197.parser	C	1,800			
XML processing				483.xalancbmk	C++	6,869
FPGA circuit placement and routing	175.vpr	C	1,400			
EDA place and route simulator	300.twolf	C	3,000			
Search gene sequence				456.hmmer	C	9,333
Ray tracing	252.eon	C++	1,300			
Computational group theory	254.gap	C	1,100			
Database program	255.vortex	C	1,900			
Library for simulating a quantum computer				462.libquantum	C	20,704
Discrete event simulation				471.omnetpp	C++	6,270
	hours	**5.3**	**19,100**	**hours**	**36.6**	**131,638**

Key Notes:

- **Reference Time (RT):** Measured on a Sun Ultra 5 with a 300MHz UltraSPARC III and 256KB L2 cache, corresponding to 100 SPEC2000.
- **Benchmark Complexity:** The runtime for integer benchmarks was 36.6 hours on a relatively old machine.
- SPEC CPU2006 introduced more complex workloads and higher reference times, demonstrating a significant evolution in benchmarking standards.

Chapter 14

Part Part IV(b) - Instruction Level Parallelism - Basic Pipelining

14.1 Circuit Timing and Performance

Most of the time, we have discussed circuits at a higher level of timing abstraction, focusing on what happens during each cycle:

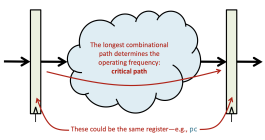
- **Finite State Machines:** `state` ← `next_state`
- **Functional Units and Memory Elements:** Perform one operation over a small number of cycles, e.g., a combinational ALU performs an addition per cycle.

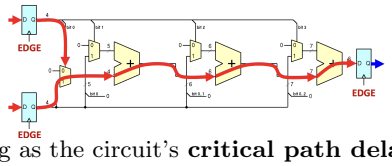
To design faster circuits, it is essential to delve deeper into the concepts of **signal propagation** and timing limitations.

14.1.1 Signal Propagation

In sequential circuits, the edges of the `clock` signal are pivotal for proper operation. They govern:

1. **Data Capture**: Determining when **new data** is latched into the combinational logic.
2. **Data Stability**: Ensuring that **processed data** (i.e., the previous input) has fully propagated through the combinational logic and is ready to be stored at the output.





To guarantee reliable operation, the `clock` period must be at least as long as the circuit's **critical path delay**—the longest delay through the combinational logic. This ensures that all signal transitions complete before the next clock edge arrives.

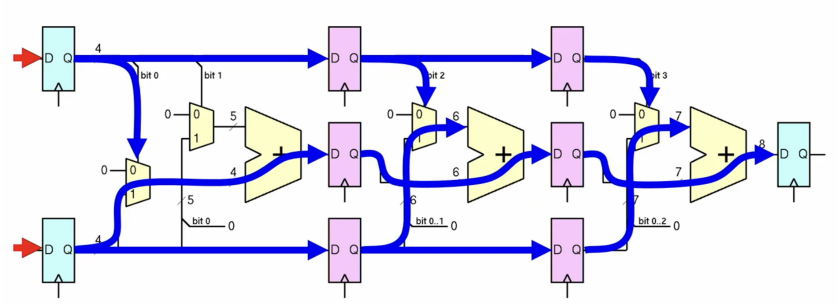
$$\texttt{Clock Period} \geq \texttt{Critical Path Delay}$$

$$T_{\texttt{clock}} \geq T_{\texttt{critical\_path}}$$

Example: In the circuit shown, the critical path is highlighted, indicating the longest combinational delay that dictates the minimum `clock` period.

Adding Intermediate Registers

Intermediate Registers can be added to break up the critical path into smaller segments, reducing the overall delay. This technique is known as **pipelining**.



Here for example, we've divided our overall critical path into three smaller segments such that, on the first clock edge, the first segment is processed, and on the second clock edge, the second segment is processed, and so on. Now, this new circuit has a shorter critical path, allowing for a faster clock period.

$$T_{\texttt{clock, pipe}} \geq T_{\texttt{new\_critical\_path}} \approx \frac{T_{\texttt{critical\_path}}}{3}$$

While this makes clock periods shorter, it also increases the number of clock cycles required to complete the operation.
Conclusion
The system's functionality remains unchanged, but the clock can run N times faster due to reduced critical path length from intermediate registers, at the cost of requiring N cycles to compute results. This allows for a finer control over the system.

14.1.2 Pipelining: Enhancing System Throughput

Pipelining is a technique widely used in computer architecture to improve the throughput of a system by overlapping the execution of multiple operations. It achieves this by dividing a task into smaller stages, where each stage performs a portion of the overall operation. These stages are connected in a pipeline structure, allowing multiple operations to be processed simultaneously.

How Pipelining Works

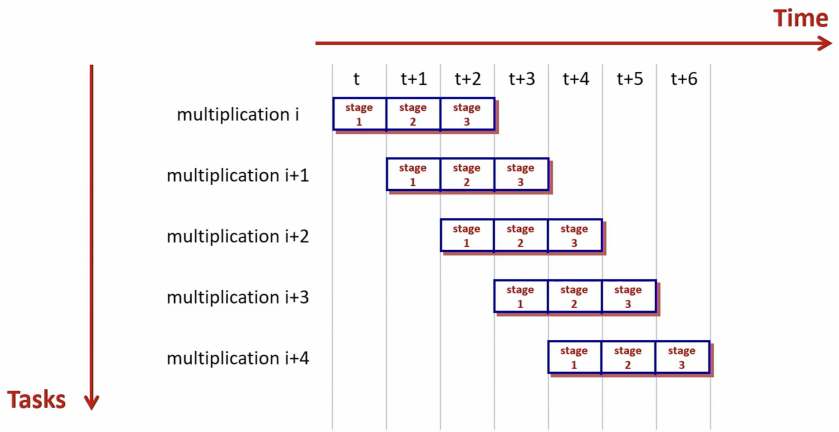
A pipeline is divided into distinct stages, each designed to execute a specific part of the operation. For example, in an arithmetic operation, the stages might include fetching data, decoding instructions, performing calculations, and writing results. Each stage operates independently and processes data sequentially.
To understand this, consider a factory analogy where a product goes through three steps:

Step 1: Assembly

Step 2: Painting

Step 3: Packaging

In a **non-pipelined factory**, one worker completes all three steps for one product before starting the next. If each step takes 1 minute, three products would require $3 \times 3 = 9$ minutes.



In a **pipelined factory**, the work is divided among three workers:

At minute 1, Worker A starts assembling the first product.

At minute 2, Worker A starts assembling the second product, while Worker B paints the first.

At minute 3, Worker A starts assembling the third product, Worker B paints the second, and Worker C packages the first.

By minute 5, all three products are completed, and the pipeline produces one product per minute after it is full. This overlapping of tasks ensures that all workers are continuously busy, reducing the overall time required to produce multiple products.

Advantages of Pipelining

The key benefits of pipelining include:

- **Improved Throughput:** By overlapping tasks, the system produces results at a faster rate. For instance, once the pipeline is full, one result can be produced per cycle.
- **Efficient Resource Utilization:** Each stage works concurrently on different parts of separate operations, preventing idle resources.
- **Scalability:** Pipelining can accommodate larger workloads by increasing the number of stages, enabling more operations to be processed simultaneously.

14.1.3 Latency and Throughput

Latency
Latency refers to the time between the start of a computation and when the result becomes available. It is given by:

- **Original Circuit:** T
- **Pipelined Circuit:** $\frac{T}{N} \times N = T$

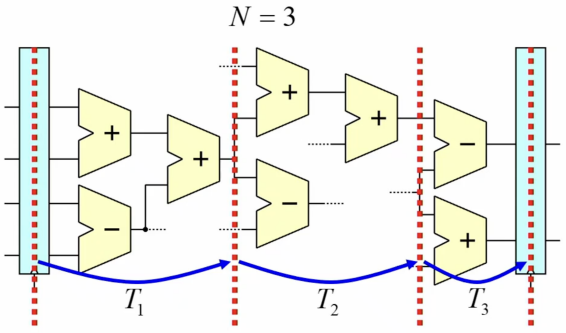
Throughput
Throughput represents the number of results produced per unit time. It is defined as:

- **Original Circuit:** $\frac{1}{T} = f$
- **Pipelined Circuit:** $\frac{1}{T/N} = \frac{N}{T} = N \times f$

14.1.4 Practical Pipelining: Latency and Throughput

Stages and Timing in Pipelining

Consider a pipeline with N stages, where each stage i takes a time T_i to complete. The pipeline is divided by registers (denoted by red dashed lines), which ensure data is synchronized between stages.



The overall operation of the pipeline is governed by:

- **Clock Period ($T_{\text{CLK,pipe}}$):** This is determined by the slowest stage, $T_{\text{CLK,pipe}} = \max(T_i + T_{\text{FF}})$, where T_{FF} accounts for flip-flop delays.
- **Stage Timing:** Ideally, $T_i \approx T_{\text{CLK,comb}}/N$, where $T_{\text{CLK,comb}}$ is the clock period of the original non-pipelined design.

Latency and Throughput of a Pipeline

- **Latency (λ_{pipe}):** The latency is the total time required for a single input to propagate through all N stages of the pipeline. It is given by:
$$\lambda_{\text{pipe}} = N \cdot \max(T_i + T_{\text{FF}}) = N \cdot T_{\text{CLK,pipe}}$$
While pipelining increases latency compared to a non-pipelined system, the trade-off is improved throughput.
- **Throughput (ϕ_{pipe}):** Throughput measures how many operations the pipeline can complete in a given time. Once the pipeline is filled, results are produced every clock cycle. It is calculated as:
$$\phi_{\text{pipe}} = \frac{1}{\max(T_i + T_{\text{FF}})} = f_{\text{pipe}}$$
where f_{pipe} is the pipeline operating frequency.

Pipelining is a practical approach to achieving high-speed operation in digital systems, particularly in processors and signal processing applications. By carefully designing stage timing and managing trade-offs, pipelining can achieve an optimal balance between latency and throughput.

Chapter 15

Part Part IV(c) - Instruction Level Parallelism

In the last chapter, we've seen how pipelining can make it easier to parallelize indepent operations making it the overall process faster.

15.0.1 Pipelining the Processor

Pipelining in processors is a technique that splits the execution of an instruction into multiple stages, each handled in parallel by separate hardware units. By doing so, multiple instructions can be processed simultaneously, thereby increasing the overall throughput of the processor without increasing the clock frequency.

- **Fetch (F)**: Retrieve the instruction from memory (often from the instruction cache).
- **Decode (D)**: Interpret the fetched instruction, identify operands, and configure the control signals for execution.
- **Execute (E)**: Perform the required operations (e.g., arithmetic, logic, load, store).

In a basic pipeline with three stages (F, D, E), each stage takes one clock cycle. While one instruction is being executed, a second instruction can be decoded, and a third can be fetched at the same time. This overlapping of tasks leads to a substantial improvement in instruction throughput.

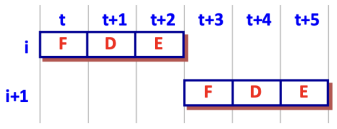
Example Pipeline Schedule

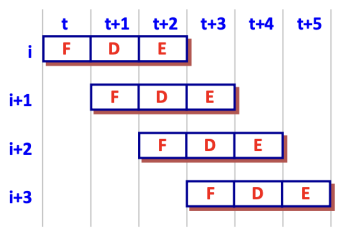
Consider a schedule where three instructions (i, $i+1$, and $i+2$) enter the pipeline. Each instruction occupies a unique pipeline stage in any given clock cycle. Figure **??** illustrates how each instruction advances one stage every cycle:

Time	t	$t+1$	$t+2$	$t+3$	$t+4$	$t+5$
i	F	D	E	–	–	–
$i+1$	–	F	D	E	–	–
$i+2$	–	–	F	D	E	–

Multi-Cycle Processor vs. Pipelined Processor

A *multi-cycle* processor might use multiple cycles to execute every instruction (e.g., separate cycles for Fetch, Decode, ALU, Memory Access, and Write Back), but only one instruction flows through the processor at a time. In contrast, a *pipelined* processor allows the next instruction to begin its Fetch stage in parallel with the Decode stage of the previous instruction, greatly improving throughput.





Key Observations for Pipelining

1. **Repetitive Activity**: Pipelining is effective only when the processor has a large number of instructions to execute.
2. **Subactivities**: Each major task (Fetch, Decode, Execute, etc.) should be clearly separable into sub-stages to allow parallel operation.
3. **Throughput Gain**: Once the pipeline is full, an instruction completes at the end of every cycle (in the ideal case), increasing throughput.

Properly designing pipeline stages and handling hazards (such as data, control, and structural hazards) ensures that the pipeline delivers high performance without correctness issues.

15.1 Hardware Reuse Across Processor Stages

In processor design, the approach to hardware reuse varies significantly between multicycle and pipelined architectures. Understanding these differences is crucial for optimizing performance and resource utilization.

15.1.1 Multicycle Processor Architecture

A multicycle processor divides instruction execution into distinct **states**, allowing certain hardware components to be shared across these states. This sharing is feasible because the components are not required simultaneously, enabling efficient resource utilization.

- **FETCH** State: Typically involves an *adder* to increment the program counter (PC).
- **EXECUTE** State: Requires an *Arithmetic Logic Unit* (ALU) to perform operations.

Since the *adder* and the *ALU* are not active concurrently, the ALU can be repurposed to increment the PC during the FETCH state. This reuse reduces the overall hardware complexity and cost.

15.1.2 Pipelined Processor Architecture

In contrast, a pipelined processor operates with multiple **stages** that are active simultaneously. Each stage performs a different part of the instruction execution process, necessitating dedicated hardware for each stage to avoid conflicts and ensure seamless parallelism.

- All pipeline stages are *active concurrently*, handling different instructions in each stage.
- Hardware components cannot be shared across stages since multiple instructions require access to the same resources simultaneously.
- Consequently, hardware must be *replicated* where necessary to maintain pipeline efficiency and prevent bottlenecks.

The inability to share hardware across pipeline stages often leads to increased hardware requirements compared to multicycle processors. However, this replication is essential for achieving high instruction throughput and maximizing pipeline performance.

15.2 Two Main Challenges in Processor Design

Designing efficient processors involves addressing several challenges. Two prominent issues are the **CISC vs. RISC** debate and **instruction independence**.

15.2.1 CISC vs. RISC

1. **Pipeline Efficiency in CISC vs. RISC**

   Question: Can we construct a pipeline for a Complex Instruction Set Computer (CISC) that matches the efficiency of a pipeline designed for a Reduced Instruction Set Computer (RISC)?

   Implications:

   - RISC architectures typically use simpler, fixed-length instructions, which are easier to pipeline efficiently.
   - CISC architectures have more complex, variable-length instructions, potentially complicating pipeline design and reducing efficiency.
   - The distinction influences processor complexity, performance, and power consumption.

2. **Ensuring Correct Execution with Dependent Instructions**

   - **Issue**: Instructions are often *dependent* on the results of preceding instructions, violating the assumption of **instruction independence**.
   - **Challenge**: Executing code correctly in the presence of such dependencies requires soxphisticated mechanisms to handle hazards, such as data forwarding or pipeline stalls.

Addressing instruction dependencies is critical for maintaining the integrity of program execution while striving for optimal pipeline performance. Techniques such as out-of-order execution and speculative execution are often employed to mitigate the impact of these dependencies.

15.3 Multi-Cycle Execution Using an FSM

In a multi-cycle processor design, each instruction's execution is broken down into multiple steps (states), and the processor transitions through these steps via an FSM.

15.3.1 FSM vs. Pipeline

While a pipeline has a fixed sequence of stages (fetch, decode, execute, memory, writeback) for any instruction, an FSM-based multi-cycle design can assign different numbers of steps to each instruction. The FSM transitions vary based on the instruction being executed.

15.3.2 Adding Instructions in a Multi-Cycle Design

When introducing new instructions (e.g., `lw` or `add`), the FSM must be extended to accommodate additional states. For example, `lw` requires computing the address and accessing memory, whereas `add` mainly requires using the ALU to perform arithmetic.

Adding `add` instruction
We can support the `add` operation without changing drastically the design, we just need to add it to the ALU.

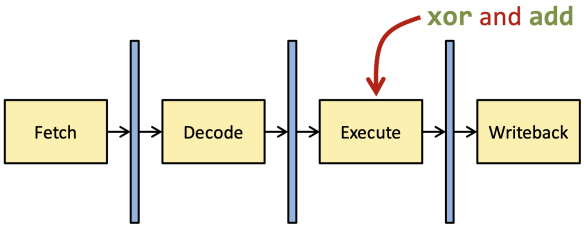
Fetch → Decode → Execute → Writeback.

Adding `lw` instruction
However, for supporting `lw` instruction, we need to introduce **memory** to our design, so we add new steps.

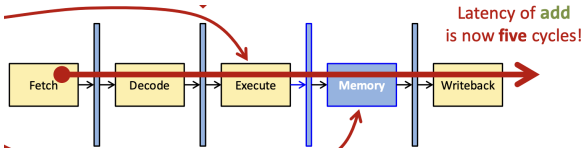
Fetch → Decode → Execute → Memory → Writeback.

15.3.3 Adding Instructions to a Pipelined Processor

Let's look at how this looks like in a pipelined processor.
In this example, we suppose xor and add instructions are both well supported.



Now, to support `lw` instruction, we need to add a memory steps, the problem here is that, in a pipelined processor, changes affect **all instructions**, meaning that now, an `add` instruction will also take 5 cycles to complete. Thus,



15.4 The Importance of the ISA (CISC vs. RISC)

The Instruction Set Architecture (ISA) heavily influences how instructions map onto hardware. A single complex CISC instruction might perform multiple memory accesses and arithmetic operations. In contrast, a RISC instruction set typically emphasizes simplicity: each instruction performs a smaller, more uniform set of operations.

15.4.1 A CISC Example

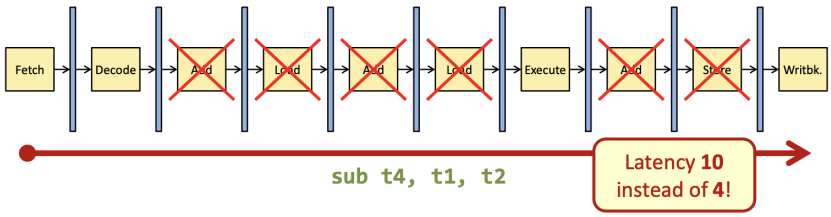
Consider a hypothetical CISC instruction:

```
sub 8(t4), 0(t1), 0(t2)
```

This single instruction might:

1. Read the value in memory at address `t2 + 0`.
2. Read another value in memory at address `t1 + 0`.
3. Subtract these two values.
4. Finally, store the result in memory at address `t4 + 8`.

Such complexity can inflate pipeline latency for *all* instructions if the pipeline must accommodate these multi-step operations within a single instruction.



15.4.2 The RISC Alternative

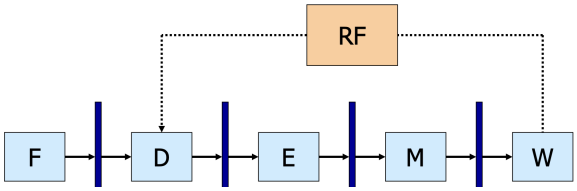
Instead of imposing a **huge penalty** to every simple instruction by making complex instructions possible, the RISC approach advocates for **only using similarly simple instructions** and building programs with these.

```
sub 8(t4), 0(t1), 0(t2)   ⟶   lw t3, 0(t1)
                              lw t5, 0(t2)
                              sub t3, t3, t5
                              sw t3, 8(t4)
```

It turns out that while this is not the only approach, **it is a good one**, and we will follow it in this course.
In practice, modern CPUs blend design philosophies, using pipelining and other advanced techniques while balancing the complexities of their ISAs. A clear understanding of these concepts—from how an FSM handles instruction steps to how a pipeline benefits from simpler instructions—is crucial to mastering processor design.

15.4.3 MIPS Pipelining Example

The MIPS architecture uses a 5-stage pipeline to execute instructions. These stages are: Fetch (F), Decode (D), Execute (E), Memory (M), and Writeback (W). Each instruction moves through these stages, enabling the overlapping of instruction execution, which improves performance by allowing multiple instructions to be processed simultaneously.

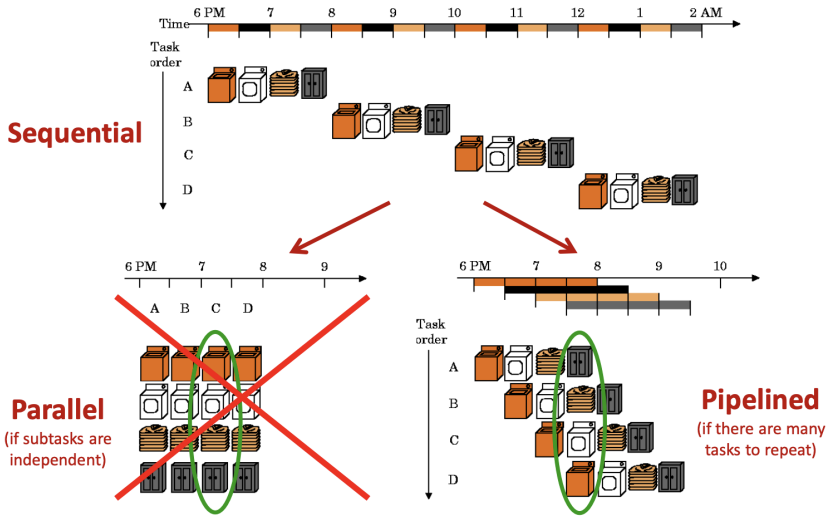


1. **Fetch (F):** The instruction is fetched from the instruction memory.
2. **Decode (D):** The instruction is decoded, and the required arguments are obtained from the register file.
3. **Execute (E):** The required operation is performed in the Arithmetic Logic Unit (ALU), including address calculations for loads and stores.
4. **Memory (M):** Access to data memory is performed if needed, particularly for load and store operations.
5. **Writeback (W):** The result of the operation, whether from the ALU or memory, is written back to the register file.

This pipelining model allows instructions to be executed in parallel, thus improving the throughput of the processor and allowing more efficient use of system resources.

15.4.4 The Laundry Metaphor for Pipelining

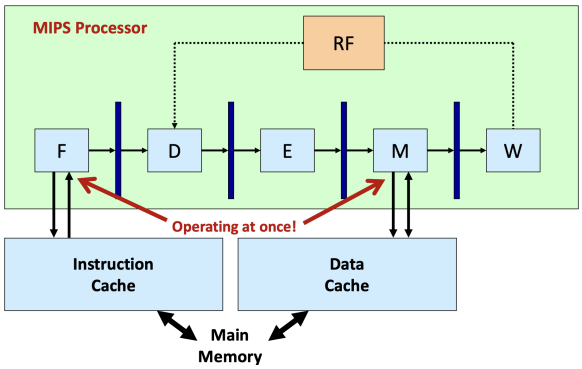
Pipelining in computer architecture can be explained using the laundry metaphor. Consider the tasks involved in doing laundry: washing, drying, folding, and putting away. Each of these steps represents a stage in the pipeline.



- **Sequential Execution:** In a sequential process, one load of laundry is completed through all stages before starting the next. This approach takes a long time because each load must wait for the previous one to finish.
- **Parallel Execution:** If the subtasks are completely independent, multiple washing machines, dryers, and folders could be used simultaneously. However, this is often impractical due to resource limitations.
- **Pipelined Execution:** In pipelining, multiple loads of laundry are processed simultaneously, with each load at a different stage. For example, while one load is being washed, another is dried, and a third is folded. This overlaps the tasks, significantly reducing total time.

15.4.5 Two Distinct Memory Interfaces in MIPS

In a MIPS processor, two distinct memory interfaces are utilized to enhance performance by allowing concurrent operations: the **Instruction Cache** and the **Data Cache**. These interfaces are depicted in the following architecture:



- The **Instruction Cache** is accessed during the **Fetch** (F) stage, retrieving instructions for execution.
- The **Data Cache** is utilized during the **Memory** (M) stage, providing data required for processing.

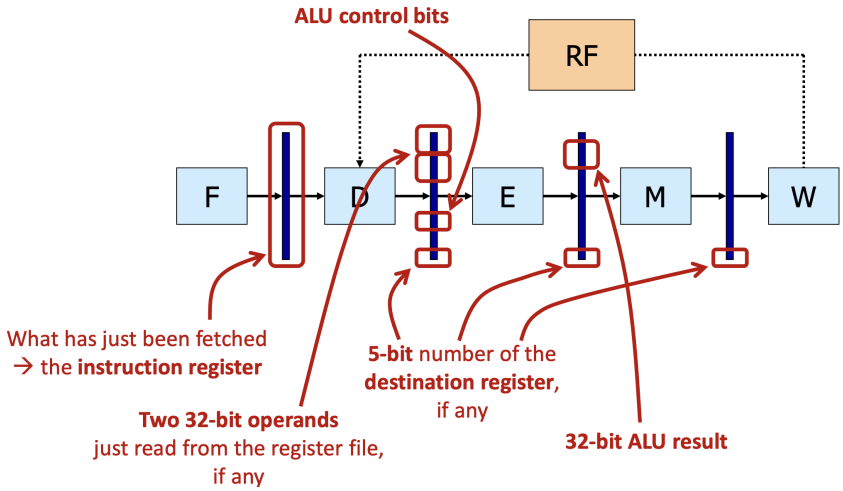
The pipeline stages of the MIPS processor are as follows:

1. **F (Fetch)**: Instructions are fetched from the Instruction Cache.
2. **D (Decode)**: Instructions are decoded, and operands are prepared using the Register File (RF).
3. **E (Execute)**: The instruction is executed.
4. **M (Memory)**: Data is accessed from or written to the Data Cache.
5. **W (Write-back)**: Results are written back to the register file.

The separation of memory interfaces allows the Instruction and Data caches to operate simultaneously, enabling improved throughput and reduced bottlenecks in the pipeline. Additionally, the Register File (RF) facilitates data flow between the stages.
This dual-interface approach is critical for high-performance pipelined architectures, allowing overlapping of instruction fetch and memory access operations.

15.4.6 Pipeline Registers and Their Contents

In a pipelined processor, each stage contains a pipeline register that holds specific data required for the correct execution of instructions. The key contents of these registers are described as follows:



- **Instruction Register (F stage):** Contains the instruction that has just been fetched from memory.
- **Operand Registers (D stage):** Stores two 32-bit operands read from the register file (if applicable).
- **ALU Control Bits (D to E stages):** Transferred to control the arithmetic logic unit (ALU) operations.
- **Destination Register Identifier (E and M stages):** Holds the 5-bit number of the destination register, specifying where the result should be written, if required.

- **ALU Result (M stage):** Stores the 32-bit result computed by the ALU for use in subsequent stages.

The flow of data across these registers ensures efficient execution of instructions while maintaining data dependencies and avoiding hazards. The proper design of pipeline registers is critical to the performance of a pipelined processor.

15.4.7 Pipeline Initialization and Execution

The Animation below illustrates the state of a pipelined processor during its execution.Initially, all pipeline stages contain `nop` (no operation) instructions, ensuring the pipeline is empty and ready to fetch the first instruction from memory.

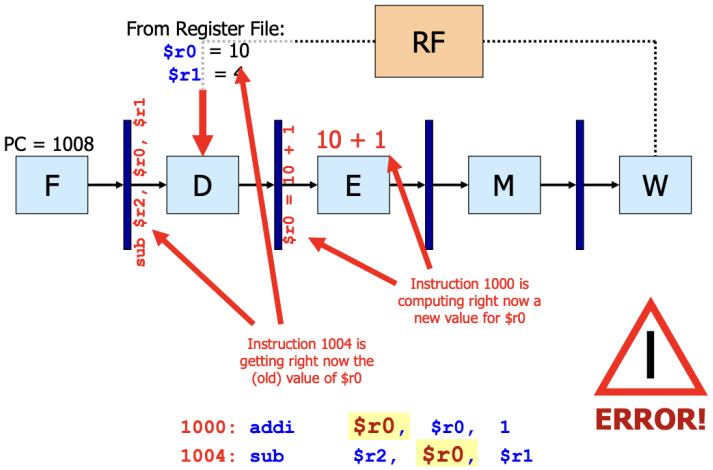
You can view an animation of the execution here.

- **Pipeline Stages:**
  - `F (Fetch):` Fetches the instruction from memory using the program counter (PC).
  - `D (Decode):` Decodes the instruction and retrieves the required register values from the register file (RF).
  - `E (Execute):` Executes arithmetic or logical operations as specified by the instruction.
  - `M (Memory):` Accesses memory for load or store operations.
  - `W (Write-back):` Writes the computed results back to the register file.
- **Initial Program Counter:** Set to 1000, fetching the first instruction (`add $r2, $r0, $r1`).

```
1000: add $r2, $r0, $r1
1004: sub $r5, $r3, $r4
1008: sw  $r6, 50($r7)
1012: lw  $r9, 20($r8)
1016: mul $r12, $r10, $r11
```

Pipeline Hazard: Data Dependency Error

Pipeline hazards can disrupt the smooth execution of instructions in a pipelined processor. One common type of hazard is the **data dependency hazard**, which occurs when an instruction depends on the result of a previous instruction that has not yet completed its execution.



Consider the following sequence of instructions executed in the previous example:

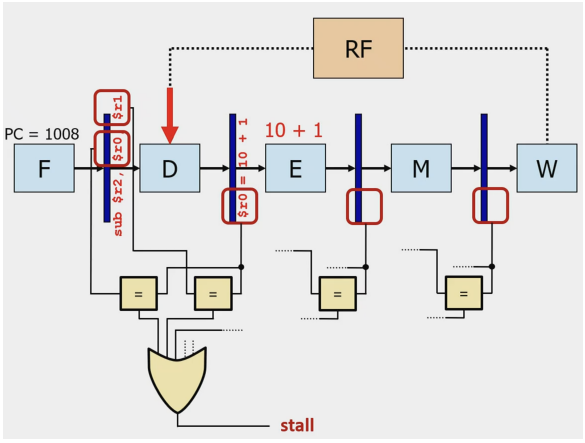
```
1000: addi $r0, $r0, 1
1004: sub $r2, $r0, $r1
```

- At program counter (PC) 1000, the `addi` instruction modifies the value of register `$r0` by incrementing it by 1.
- At PC 1004, the `sub` instruction attempts to compute `$r2` using the value of `$r0`. However, it fetches the old value of `$r0` from the register file because the result of the `addi` instruction is not yet available.

In this scenario, the `addi` instruction at PC 1000 is still in the `E` stage when the `sub` instruction at PC 1004 enters the `D` stage. Consequently, the `sub` instruction reads an outdated value of `$r0`, leading to an incorrect result.

15.5 Data Hazard Detection in Pipelined Processors

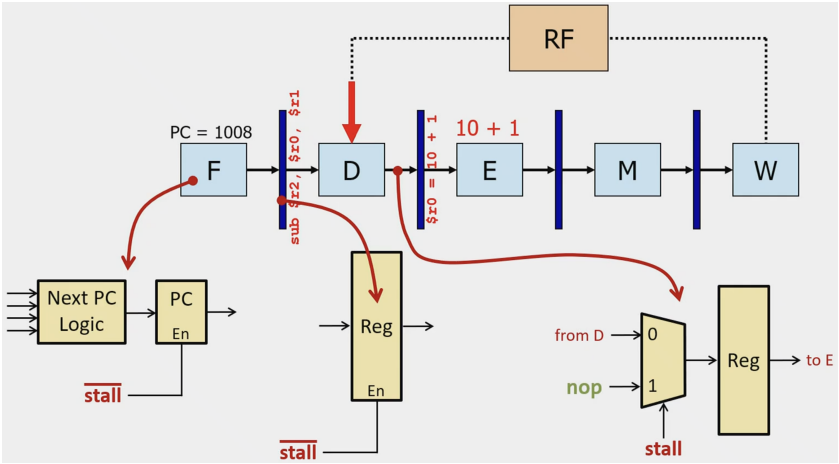
Data hazards occur when instructions in a pipelined processor depend on the results of previous instructions that have not yet completed execution. To address this, the pipeline employs a hazard detection unit. The figure below illustrates the hazard detection mechanism for a sequence of instructions in a pipelined processor.



In this example, a dependency exists between two instructions: the `sub` instruction in the Decode (D) stage requires the result of the `addi` instruction, which is still being processed in the Execute (E) stage. The hazard detection unit identifies this dependency and intervenes to maintain correct operation.

15.5.1 Stalling in Instruction Execution

Once a data hazard is detected, the pipeline resolves it using a stalling mechanism. This ensures that dependent instructions do not proceed until the required data becomes available. For the example discussed above, the stalling mechanism operates as follows:



Stall Mechanism

When the hazard detection unit identifies a data dependency, it triggers a stall in the pipeline to delay the dependent instruction until the required data becomes available. This is achieved as follows:

1. The dependent instruction (e.g., `sub`) is held in the *Decode (D)* stage until the instruction producing the required data (e.g., `addi`) computes its result in the *Execute (E)* stage.
2. A `nop` (no-operation) instruction is inserted into the pipeline to introduce a delay, allowing the producing instruction to complete its operation.
3. The control logic halts the Program Counter (PC) and relevant pipeline registers temporarily to synchronize the execution flow.

The flow of execution proceeds as follows:

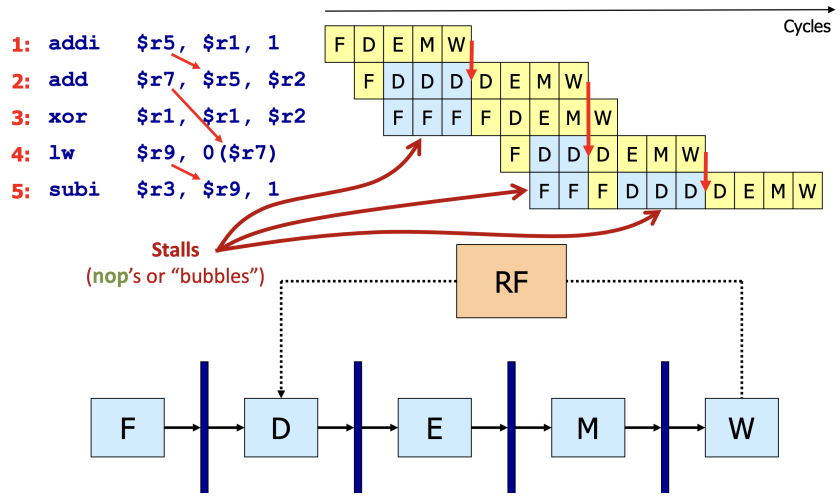
- The producing instruction progresses through the pipeline, computing its result in the E stage.

- Simultaneously, the pipeline stalls the dependent instruction, holding it in the *D* stage and inserting a `nop` in the *Execute (E)* stage.
- Once the producing instruction writes its result to the register file, the dependent instruction resumes execution using the updated value.

While stalling reduces the overall throughput of the pipeline by introducing delays, it is essential for ensuring the correctness of computations. The hazard detection and stalling mechanism together provide a robust solution for managing data dependencies in pipelined processors.

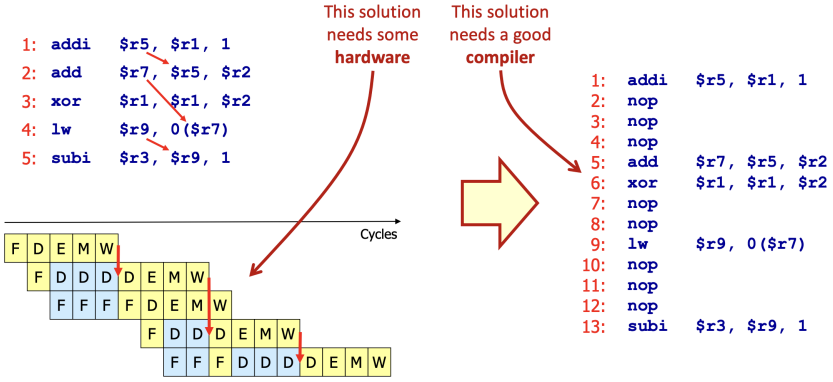
Conclusion

By introducing stalling, the updated execution diagram is as follows:



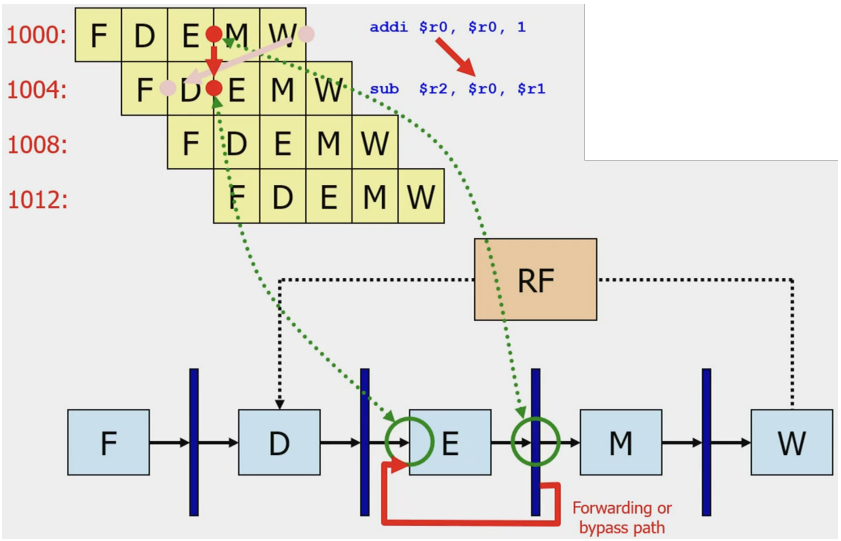
15.5.2 Alternative Solution

An alternative approach to solving this problem could involve managing stalling in software. For instance, the compiler could statically insert the appropriate number of `nop` instructions before runtime to avoid any data hazards.



15.5.3 Data Hazards Resolved by Forwarding

In pipelined processors, data hazards arise when an instruction depends on the result of a previous instruction that has not yet been written to the register file. Forwarding, also known as bypassing, is a hardware technique used to address this issue effectively.



- **Timing vs. Location:** The required data becomes available at the correct time but is not in the necessary location for the dependent instruction.
- **Direct Data Paths:** Forwarding paths are established to transfer data directly from one stage of the pipeline to another, eliminating unnecessary delays.
- **Control Logic:** Additional circuitry, such as multiplexers, is employed to select the appropriate data input during the execute stage.

Analogy:
Imagine an assembly line in a factory where one worker produces a component needed by the next worker. Instead of waiting for the component to be placed in storage and then retrieved, the first worker directly hands it to the second worker. Similarly, forwarding allows data to be passed directly between pipeline stages without waiting for it to be written back to the register file.
Example
Consider the following two instructions:

```
addi $r0, $r0, 1
sub  $r2, $r0, $r1
```

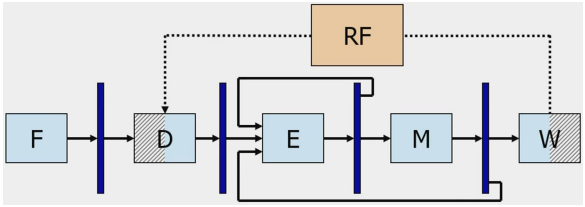
The result of the `addi` instruction must be used by the `sub` instruction. Without forwarding, the pipeline would have to wait until the `addi` result is written back to the register file before the `sub` instruction can proceed, causing a stall. With forwarding, the result from the `addi` instruction is directly sent to the execute stage of the `sub` instruction, allowing it to proceed without waiting.
Implementation:
The forwarding mechanism introduces hardware paths from:

- The **Execute/Memory** pipeline register back to the Execute stage.
- The **Memory/Write-Back** pipeline register back to the Execute stage.

These paths are controlled to ensure that the correct data is provided to the execute stage when needed, bypassing the usual wait for the write-back phase and thereby maintaining the smooth flow of instructions through the pipeline.

15.5.4 Classic MIPS Pipeline with Forwarding

Please take some time to understand this part, The classic MIPS pipeline consists of five stages: *Instruction Fetch* (**F**), *Instruction Decode / Register Read* (**D**), *Execute* (**E**), *Memory Access* (**M**), and *Write Back* (**W**).
Forwarding is a mechanism to resolve data hazards by enabling certain results to be directly passed between pipeline stages, bypassing the need for intermediate storage. When fully implemented, forwarding enables the following data paths:



- **E → E:** The output from the Execute stage (**E**) of one instruction can be forwarded directly to the Execute stage of the next instruction, avoiding dependency-related stalls.
- **M → E:** The output from the Memory stage (**M**) can be forwarded to the Execute stage of the next instruction if required, bypassing the need to wait for the value to reach the Write Back stage.
- **W → D:** The output from the Write Back stage (**W**) can be supplied directly to the Decode stage (**D**) of a subsequent instruction during the same clock cycle.

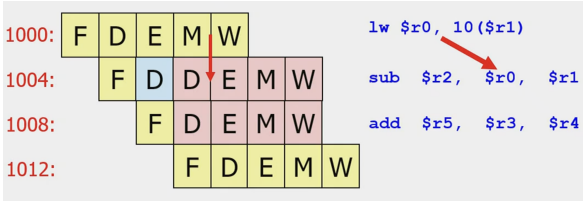
A notable special case is *register-file forwarding* (**W → D**). In this scheme:

- During the **W** stage, registers are written in the *first half* of the clock cycle.
- During the **D** stage, registers are read in the *second half* of the same cycle.

This timing ensures that a register value written in **W** is immediately available for reading in **D**, avoiding a read-after-write hazard for consecutive instructions.
In the classic MIPS pipeline, the forwarding paths described (**E → E**, **M → E**, **W → D**) are sufficient to resolve most data hazards. However, some instruction sequences may still require additional stall cycles if the needed forwarding path is unavailable.

15.6 Structural Hazards

A **structural hazard** occurs when multiple instructions simultaneously require the same hardware resource (e.g., pipeline stage or memory port), potentially causing pipeline stalls or incorrect execution.



Example

Consider the following instruction sequence:

```
1. lw   $r0, 0($r1)      # Load word into $r0
2. lw   $r2, 4($r1)      # Load word into $r2
3. add  $r3, $r0, $r2    # Add $r0 and $r2, store in $r3
```

In a pipelined processor with a single memory port, both `lw` instructions (1 and 2) attempt to access memory during the Memory (M) stage simultaneously. This conflict creates a structural hazard, forcing the processor to stall one instruction until the memory resource becomes available, thereby reducing throughput.

Handling Cache Misses

Cache misses can introduce structural hazards by stalling pipeline stages:

- A miss in the Memory (M) stage can block access to memory or the bus, causing stalls in the Execute (E), Decode (D), and Fetch (F) stages.

Stalling Strategies

- **Single Cache Miss**:
  - Stall the Memory (M) stage and all preceding stages (E, D, F).
  - Allow the Write-Back (W) stage to proceed to avoid blocking completed operations.
- **Concurrent Misses**:
  - Prioritize data cache misses over instruction cache misses to minimize overall stall time.
  - This may temporarily cause resource contention on main memory.

Consequences of Unresolved Structural Hazards

- Instructions may execute out of order or prematurely, leading to resource conflicts.
- This can result in incorrect execution or pipeline corruption.

Prevention in Our Pipeline

- Careful resource allocation ensures that structural hazards are **avoided**.
- Adequate provisioning of each pipeline stage and proper management of shared resources enable concurrent operation without contention.

Dependency Example

In the example above, the `sub` instruction depends on the result of the `lw` instruction ($r0). If $r0 is not ready when `sub` reaches the Execute stage, a structural hazard can occur unless stalls or forwarding mechanisms are implemented to handle the dependency.

15.7 Control Hazards in Pipelined Processors

15.7.1 The Problem

A *control hazard* (or *branch hazard*) occurs in a pipelined CPU whenever the next instruction to fetch depends on the result of a branch. Since pipeline stages operate in parallel, the processor often fetches an instruction *before* it knows whether the branch will be taken or not. If the branch is taken, the speculatively fetched instruction is wrong and must be *flushed* (or invalidated); if the branch is not taken, execution continues normally.

Pipeline Stages: F (Fetch), D (Decode), E (Execute), M (Memory), W (Write-back)

Example

```
Address   Instruction
---------------------------------
1000      beq   $r0, $r1, loop
1004      sub   $r2, $r0, $r1
1008      ...   (next instruction)
```

When the CPU starts to fetch the instruction at `1004`, it may not yet know if the branch at `1000` is taken. If the branch is taken, the fetched instruction at `1004` is incorrect and must be discarded.

15.7.2 Stalling (Flushing) the Pipeline

One straightforward solution is to *stall* the pipeline until it is known whether the branch will be taken. Conceptually:

1. Fetch the branch instruction.
2. Stall new instruction fetches until the branch outcome is computed (e.g., by the end of the E stage).
3. If the branch is taken, jump to the correct target address; if not, continue with the sequential address.

Stalling ensures correctness, but it **wastes several cycles** whenever a branch is encountered.

- **Fetching & Decoding a wrong instruction** is not harmful as long as it is not *executed*.
- If the branch resolves in the E stage (cycle 3 for a 5-stage pipeline), two extra instructions may have been fetched speculatively. If the branch is taken, those two instructions are *killed* (flushed); if not, they continue without delay.

15.7.3 Delay Slots

Another approach, historically used by MIPS and a few others, is to define *delay slots* after a branch. In a 1-slot design:

1. The instruction *immediately* following the branch is always executed (the "delay slot").
2. The branch effect (taken or not taken) occurs *after* that delay slot instruction completes.

Hence, the architecture enforces that instructions in the delay slot *always* run, regardless of whether the branch is taken. In a 2-slot design, the next two instructions always execute, and so on.

Example of inserting NOPs (no-ops) in delay slots:

```
1:   beq  $r0, $r1, loop  <-- branch
2:   nop                  <-- delay slot #1
3:   nop                  <-- delay slot #2
4:   sub $r2, $r0, $r1    <-- next real instruction
```

While legal, using NOPs in delay slots merely shifts the stall problem into software. More sophisticated compilers attempt to move independent instructions into these slots so that the extra cycles are not wasted.

15.8 Summary

Pipelining in computer architecture can be hindered by three primary types of hazards:

- **Data Hazards** (*data dependences*): These occur when instructions depend on the results of previous instructions.
  - *Solutions:*
    - Forwarding paths, wherever possible.
    - Stalls, in all other cases.
- **Control Hazards** (*jumps and branches*): These arise from the control flow of instructions, such as branches and jumps.
  - *Solutions:*
    - Delay slots, if the architecture allows it.
    - Branch prediction, to try to do the right thing.
    - Stalls, if the above are not feasible.
- **Structural Hazards** (*conflicting need for a resource*): These occur when multiple instructions require the same hardware resource simultaneously.
  - *Solutions:*
    - Design rigid pipelines that avoid structural hazards by construction.
    - Use stalls in cases where hazards cannot be avoided.

Chapter 16

Part IV(d) - Instruction Level Parallelism - Scheduling

16.1 Dynamic Scheduling and Out-of-Order Execution

Modern processors often employ **dynamic scheduling** to exploit more instruction-level parallelism (ILP). Instead of strictly following the program's original instruction order (*in-order* execution), the processor can:

- **Fetch and decode** instructions as early as possible, even if previous instructions have not completed.
- **Reorder** instruction execution to avoid idle functional units when long-latency instructions (e.g., divides) are in progress.
- **Ensure correctness** by respecting true data dependencies and properly writing results back in program order (via a *Reorder Buffer (ROB)*).

16.1.1 Motivating Example

Consider the following sequence of floating-point operations:

```
divd  $f0, $f2, $f4    # Long-latency division
addd  $f10, $f0, $f8   # Depends on divd's result
subd  $f12, $f8, $f14  # Independent of divd's result
```

- In a strict in-order pipeline, the `subd` could be stalled until `divd` completes its execution (because `addd` is waiting on $f0).
- With dynamic scheduling, the processor can reorder `subd` before `addd` as soon as it identifies that `subd` does *not* depend on `divd`.
- This reordering utilizes available resources without violating correctness.

16.1.2 Breaking the Rigidity of Basic Pipelines

In a standard five-stage pipeline (Fetch, Decode, Execute, Memory, Writeback), stalls are common when earlier instructions hold resources or have unresolved data dependencies. Dynamic scheduling mitigates these stalls by:

1. **Continuing to fetch and decode** new instructions (even if some are stalled in execution).
2. **Deferring writeback** until resources become available or dependencies are resolved, using dedicated hardware structures (e.g., reservation stations, reorder buffers).
3. **Allowing out-of-order completion**: instructions finish as soon as they can, but their results are committed in-order to preserve program semantics.

16.1.3 Dynamically Scheduled Processor Overview

A typical dynamically scheduled CPU integrates:

- **Fetch/Decode** units that feed instruction information into *reservation stations* (RS).
- Multiple functional units (ALU, FPU, Memory pipelines) operating in parallel.

- A **Reorder Buffer (ROB)** to track instruction completion and to ensure in-order retirement (commit) of results.
- **Forwarding paths** to provide operands directly to waiting instructions without requiring all results to be written back to the register file first.

By decoupling instruction fetch/decode from their actual execution, a **dynamically scheduled processor** allows more effective use of hardware resources and improves overall performance, particularly for workloads with long-latency operations or frequent stalls in in-order pipelines.

16.1.4 Reservation Stations

Reservation stations are hardware queues that temporarily hold instructions before they are sent to an execution unit. They are a key component in enabling **out-of-order execution** within a processor. This mechanism allows the CPU to execute instructions as soon as their necessary resources are available, rather than strictly adhering to the program's original order.

How Reservation Stations Work

Reservation stations facilitate several critical functions in the CPU:

1. **Check Operand Availability:** They verify that all input operands required by an instruction have been computed and are available, either in the register file or through bypassing from another execution unit.
2. **Prevent Structural Hazards:** They ensure that the targeted execution unit is free and ready to accept a new operation, thereby avoiding conflicts over shared functional units.
3. **Enable Dynamic Scheduling:** They allow instructions to be dispatched to available execution units as soon as both their operands and the necessary functional resources are ready, rather than waiting for earlier instructions to complete.

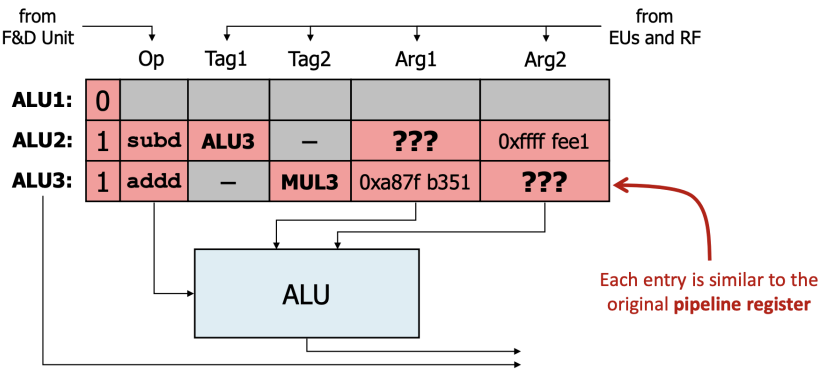
Components of a Reservation Station

Each entry within a reservation station typically contains the following elements:

- *Operation Code* (e.g., `add`, `sub`, `mul`): Specifies the operation to be performed.
- *Operands or Tags*: Contains the actual operands needed for the operation or tags that reference future instructions responsible for producing those operands.
- *Status Bits*: Indicate whether the operands are ready and whether an appropriate execution unit has been reserved.

Execution Process

1. **Issuing Instructions:** When an instruction enters a reservation station, it waits until both its operands are available and the required execution unit is free.
2. **Executing Instructions:** Once these conditions are met, the reservation station issues the instruction to the execution unit.
3. **Broadcasting Results:** After execution, the result is forwarded to all reservation stations that are waiting for that particular value, updating their entries to reflect that the operand is now valid.



Analogy: Kitchen Order Management

Think of reservation stations as a **kitchen's order management system** in a restaurant:

- **Orders as Instructions:** Each customer's order is an instruction that needs to be prepared.
- **Ingredients as Operands:** The ingredients required for each dish represent the operands. An order can only be prepared if all necessary ingredients are available.
- **Chefs as Execution Units:** The chefs are the execution units that prepare the dishes.

Process Flow:

1. **Taking Orders:** Orders are placed in the reservation station (waiting area) as they come in.
2. **Checking Ingredients:** The system verifies that all ingredients for an order are available.
3. **Assigning to Chefs:** If ingredients are ready and a chef is available, the order is handed off to the chef for preparation.
4. **Updating Availability:** Once a dish is prepared, the result is available to fulfill other orders that might depend on it.

Summary

Reservation stations decouple the dispatching of instructions from the availability of their operands and the execution units. By doing so, they enhance the processor's ability to execute instructions out of order, thereby improving overall performance and efficiency.

16.1.5 Register Renaming and Data Dependencies

Register renaming is a technique used to eliminate **Write-After-Write (WAW)** and **Write-After-Read (WAR)** dependencies, collectively referred to as *name dependencies*. These dependencies occur because registers are reused across multiple instructions, despite the absence of actual data flow between them.

Pipeline Hazards and Dependency Types

Pipeline execution is prone to the following dependencies:

- **Read-After-Write (RAW):** True data dependence, where an instruction requires the output of a previous one.
- **Write-After-Write (WAW):** Name dependence, resolved by renaming.
- **Write-After-Read (WAR):** Name dependence, resolved by renaming.

In dynamic pipelines, out-of-order execution can introduce hazards like WAW and WAR. Renaming ensures correctness by maintaining unique register identifiers, enabling both in-order and out-of-order pipelines to avoid conflicts.

Example

Consider the following instructions:

```
divd  $f0, $f1, $f2
addd  $f3, $f0, $f4
subd  $f4, $f5, $f6
adddi $f0, $f5, 10
```

- `addd` has a **RAW** dependence on `divd`.
- `subd` has a **WAR** dependence on `addd`.
- `adddi` has a **WAW** dependence on `divd`.

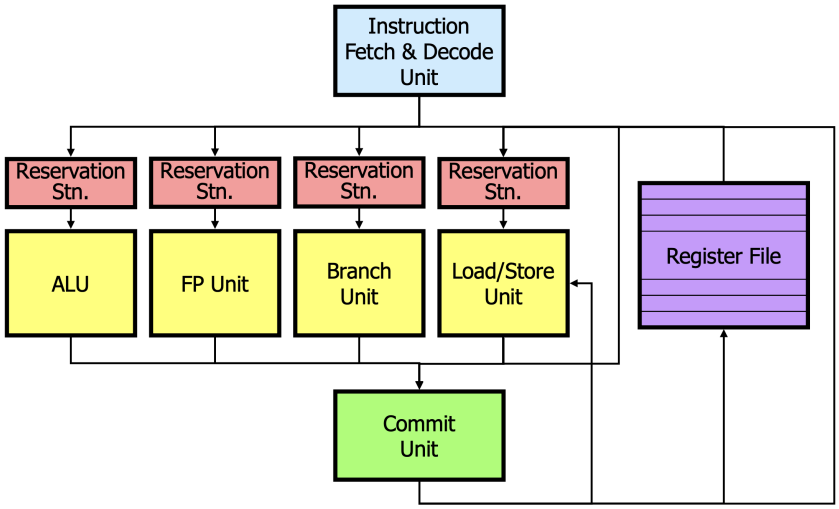
By renaming, these dependencies are resolved:

```
divd  $f0, $f1, $f2
addd  $f3, $f0, $f4
subd  $f30, $f5, $f6
adddi $f29, $f30, 10
```

This ensures the pipeline executes efficiently without conflicts, improving instruction throughput.

16.2 Dynamically Scheduled Processor

A *dynamically scheduled processor* uses hardware mechanisms to exploit *out-of-order* execution, allowing instructions to proceed as soon as their operands become available. This contrasts with a strictly pipelined MIPS design, where all instructions flow in *order* through the five pipeline stages (F, D, E, M, W). By dynamically scheduling instructions, the processor can reduce stalls and more effectively utilize hardware resources.



- **Instruction Fetch & Decode Unit:** Fetches and decodes instructions, dispatching them to the appropriate *reservation stations* once the instruction type is identified.
- **Reservation Stations:** Buffers that hold instructions waiting for the required operands or execution unit to become available. Each functional unit (e.g., ALU, floating-point, branch, or load/store) typically has its own set of reservation stations.
- **ALU, FP Unit, Branch Unit, Load/Store Unit:** Execution units where instructions are actually carried out. The Load/Store Unit also manages memory operations. Because these units operate in parallel, multiple independent instructions can be serviced simultaneously.
- **Register File:** Stores the architectural registers. Instructions read from and write to this file (potentially out of program order), but the final states are committed in order, preserving correct program semantics.
- **Commit Unit:** Also referred to as the *retirement* or *write-back stage*. It ensures that the processor's *architectural state* is updated in the correct program order, even though internal execution may be out of order.

Example Execution

Suppose we have the following sequence of instructions:

$$I1 : \text{R1} \leftarrow \text{R2} + \text{R3}$$
$$I2 : \text{R4} \leftarrow \text{R1} \times \text{R5}$$
$$I3 : \text{R6} \leftarrow \text{R7} + \text{R8}$$

In a strictly pipelined MIPS processor, I2 would have to stall while waiting for R1 (produced by I1) to be written back. However, a dynamically scheduled processor can place I2 into a reservation station, and simultaneously issue I3 to the ALU, because I3 does not depend on I1 or I2. This allows overlapping execution and reduced idle cycles.

16.2.1 Precise vs. Imprecise Exceptions

Exceptions occur when the processor encounters an event requiring special handling (e.g., page fault, unsupported instruction). They can be categorized as *precise* or *imprecise* based on whether the exact instruction that caused the exception—and the architectural state associated with that point in the instruction stream—can be precisely identified.

- **Precise Exceptions**
  - The processor enforces an in-order view of instruction completion at the point of the exception.
  - This implies that all instructions before the faulting instruction have completed, and none of the subsequent instructions have begun or altered the architectural state.
  - Reordering or out-of-order execution may still happen internally, but when an exception occurs, the processor "commits" instructions in a way that appears strictly in-order.
  - This behavior simplifies error handling, as the operating system (OS) or exception handler knows exactly where the problem occurred and which instructions completed.
- **Imprecise Exceptions**
  - Out-of-order execution becomes visible to the user (or OS), meaning the faulting instruction might not be clearly identified at the time of the exception.
  - The OS or programmer must assume that instructions have partially or fully executed in a different order than expected.
  - Correcting the architectural state becomes more complex; the system may need to re-execute an entire subroutine to ensure correctness.
  - Modern architectures generally avoid imprecise exceptions because of these complexities (especially for critical features such as virtual memory or I/O).

Out-of-Order Commitment and Exceptions

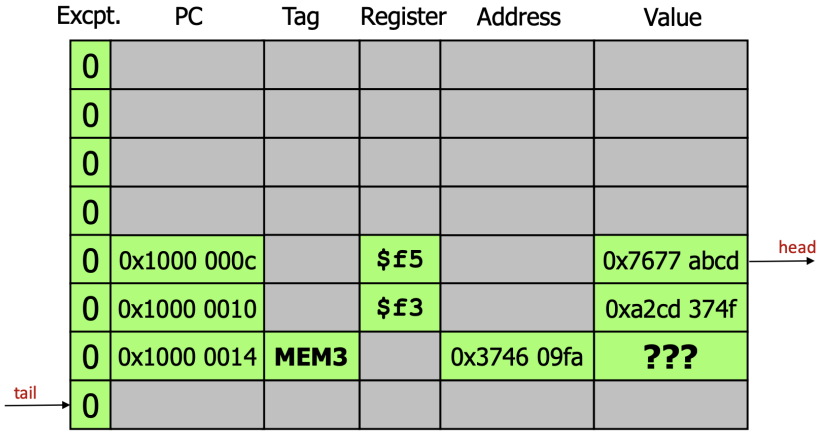
Dynamic (out-of-order) execution complicates exception handling because the processor may complete some instructions after the faulting instruction if it issued them early. In a precise exception model, the hardware automatically rolls back or defers the effects of later instructions so that:

1. Everything before the faulting instruction is guaranteed to have completed.
2. No instructions after the faulting instruction have committed any state.

This exact commitment model allows the exception handler to identify the precise location of the fault. When exceptions are imprecise, the program may need to be restarted from an earlier point to restore correct state, making it challenging for both the hardware and software to manage.

16.2.2 Reordering Instructions at Writeback

A *reorder buffer* (ROB) is used in out-of-order processors to maintain correct program order when writing back results to the architectural register file and memory. While instructions may execute in parallel or out of order, the ROB ensures that their visible effects (writes to registers/memory) occur in the original program order. This mechanism preserves the logical behavior of the program while also taking advantage of pipeline parallelism.



High-Level Overview.

- **Out-of-Order Execution:** After fetching and decoding, instructions are dispatched to execution units as soon as their operands become available. This enables the processor to exploit instruction-level parallelism.
- **Reorder Buffer (ROB):** Each fetched instruction is allocated an entry in the ROB. The entry holds:
  1. The instruction's *program counter* (PC) and a unique *tag*.
  2. The *destination* (register or memory address).
  3. The *result* value (once the execution unit produces it).
  4. An *exception status* field to indicate whether an exception occurred.
- **Writeback & Commit:** Although execution finishes out of order, the ROB enforces an *in-order* commit. The oldest (head) entry in the ROB is checked first:
  1. If its result is ready and no exception has occurred, the commit unit writes the result to the destination register or memory location.
  2. The ROB entry is then freed, and the *head* pointer moves to the next instruction.
- **Preserving Program Correctness:** If an exception flag is set for the head entry, the pipeline can be flushed and the exception handled in program order, ensuring correct state recovery.

Execution Steps.

1. *Fetch & Decode:* Instructions are fetched in program order and assigned entries in the ROB. Each ROB entry records necessary metadata (PC, destination, etc.).
2. *Dispatch to Execution Units:* As soon as sources for an instruction are ready, it can be sent to an available execution unit. Meanwhile, the ROB entry remains allocated to that instruction.
3. *Receive Results in ROB:* Once an execution unit finishes, it writes the result (along with the instruction's tag) back to the ROB. The destination register or memory is *not* updated yet.
4. *Commit in Program Order:* The reorder buffer's head entry is checked:
   - If the head instruction's result is available and no exception is flagged, that value is *committed* to the architectural register file or memory in correct program order.
   - The head pointer is advanced, retiring the entry from the ROB.
   - This process repeats as subsequent instructions at the head become ready and valid.

Why It Improves Performance.

Even though instructions are effectively *reordered* before the final writeback, the pipeline overlaps multiple steps of different instructions. The reorder buffer allows:

- **Parallel Execution:** Independent instructions execute simultaneously in different pipeline stages, reducing overall latency.
- **Hazard Resolution:** The ROB tracks which instructions have completed and can manage data hazards by forwarding results as soon as they are produced.
- **In-Order Commit Guarantee:** The programmer-visible state updates in strict order (the ROB's head-to-tail sequence), preserving correct semantics without stalling earlier instructions for later ones.

Thus, the reorder buffer provides the illusion of in-order execution while allowing out-of-order performance gains. As soon as an instruction completes, its result is available for subsequent instructions; however, to ensure correctness, final commitment of these results to the architectural state occurs strictly in the order of the original program.

Dynamically Scheduled Processor: Step-by-Step Execution

This outlines the operation of a dynamically scheduled processor, broken down into modular stages that can be adapted for various design choices (e.g., with or without forwarding paths, reservation stations, etc.). Each step is presented using a structured **if-then-else** format and emphasizes how instructions flow through the pipeline under different conditions.

Basically an algorithm to answer this chapter's exercises.

1. Instruction Fetching

1. **Fetch Attempt**
   - **IF** the instruction cache (I-cache) is ready to serve a new instruction **THEN**
     - Fetch the next instruction address from the Program Counter (PC).
     - Send the address request to the I-cache.
     - Increment or update PC for the next instruction (or branch target if known).
   - **ELSE** (*I-cache miss* or *pipeline stall condition*)
     - Stall the fetch stage until the I-cache responds, or the pipeline is cleared.
2. **Branch Misprediction Handling**
   - **IF** a branch misprediction is detected **THEN**
     - Flush the fetched instructions after the mispredicted branch.
     - Update PC with the correct branch target.
     - Re-fetch instructions from the correct location.
   - **ELSE**
     - Continue normal fetching.

2. Instruction Decoding

1. **Decode Phase**
   - **IF** the decode (or dispatch) hardware and any necessary pipeline registers are available **THEN**
     - Read the fetched instruction.
     - Decode the opcode and identify operands and destination register(s).
   - **ELSE**
     - Stall decode until resources become free.
2. **Hazard Checks**
   - **IF** there is a structural hazard (e.g., decode hardware busy) **THEN**
     - Stall the decode stage until the hazard is cleared.
   - **IF** there is a data hazard (register not yet available or pending in the Reorder Buffer) **THEN**
     - Mark the instruction as needing operands from future writes or forwarding paths.
   - **ELSE**
     - Proceed to place the instruction into an appropriate reservation station.

3. Reservation Stations

1. **Instruction Buffering and Dependency Resolution**
   - **IF** a reservation station matching the instruction type (ALU, FP, Branch, or Load/Store) is free **THEN**
     - Place the instruction in the station, along with operand tags or values.
     - Check which operands are currently valid (available in the register file, or forwarded) and which are pending.
   - **ELSE**
     - Stall the instruction until a reservation station becomes available.
2. **Operand Availability**
   - **IF** all input operands are ready **THEN**
     - Dispatch the instruction to the appropriate execution unit immediately.
   - **ELSE**
     - Wait for forwarding signals or for the Reorder Buffer (ROB) to broadcast the result.

4. Execution Units

1. **Instruction Execution**
   - **IF** the execution unit is free and all operands are valid **THEN**
     - Execute the instruction (e.g., perform ALU operation, load/store, floating-point operation, or branch evaluation).
   - **ELSE**
     - Stall in the reservation station until the execution unit is available and any missing operands are forwarded.
2. **Forwarding and ROB Updates**
   - **IF** the processor supports forwarding **THEN**
     - Immediately broadcast the result on the Common Data Bus (CDB) so dependent instructions can receive the value without waiting for it to be written to the register file.
   - **ELSE**
     - Write the result into the reorder buffer and/or register file.
     - Dependent instructions must wait until the write is complete to read the result.

5. Commit Unit

1. **Reorder Buffer (ROB) and In-Order Commit**
   - **IF** the instruction is at the head of the ROB and has completed execution with no exceptions **THEN**
     - Commit the result to the architectural register file or memory (for store operations).
     - Remove the instruction entry from the ROB.
   - **ELSE** (*instruction not yet at head of ROB* or *exception detected*)
     - Stall the commit stage until the head instruction is fully ready.
     - **IF** an exception or misprediction is detected **THEN**
       - Flush instructions in the ROB after the faulting or mispredicted instruction.
       - Recover architectural state from the last known good state or from checkpoints.

6. Register File

1. **Register Access and Dynamic Scheduling Support**
   - **IF** the result is not yet committed (i.e., it resides in the ROB) **THEN**
     - Dependent instructions obtain the result from forwarding paths or by listening to the ROB broadcast.
   - **ELSE**
     - Read the committed value from the register file in the usual manner.

7. Execution Scenarios (Decision Tree)

Scenario 1: With Forwarding Paths

1. **IF** an instruction completes in the execution unit
   - **THEN** broadcast the result immediately to all reservation stations listening for that tag.
   - Dependent instructions that had this operand pending can now proceed to execution in the next cycle (if their other operands are also ready and an execution unit is free).

Scenario 2: Without Forwarding Paths

1. **IF** an instruction completes in the execution unit
   - **THEN** the result is first written to the reorder buffer (and eventually to the register file).
   - Dependent instructions wait until the value is visible in the register file or the ROB can broadcast the commitment.

Lastly, general rule, if something is busy, or if it doesn't have enough space to add an instruction/operation etc. . . , Stall.

8. Performance Comparison

- Dynamic scheduling allows multiple instructions to be *in-flight*, decoding and executing out of order as their operands become available.
- **Compared to a simple in-order pipeline**:
  - More instructions can execute in parallel if they do not depend on each other.
  - Hazards are resolved dynamically, leading to fewer pipeline stalls.
  - Forwarding paths further reduce stalls by providing immediate data to dependent instructions.

- Overall, the utilization of hardware resources is improved and throughput (instructions per cycle) is increased.

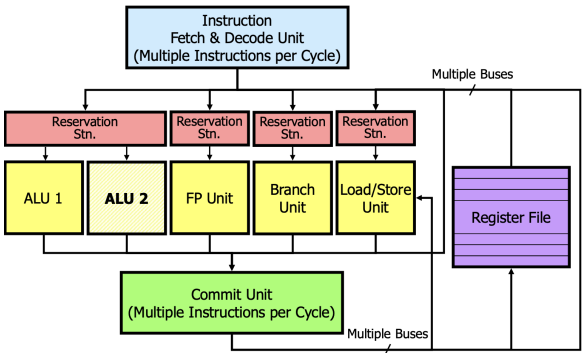
Chapter 17

Part 4f: Instruction Level Parallelism (ILP) Besides and Beyond Superscalars

Instruction-Level Parallelism (ILP) is the measure of how many operations in a computer program can be performed simultaneously. The goal of ILP is to exploit parallel execution within a single thread to improve performance. Traditional *superscalar* architectures achieve ILP by issuing multiple instructions in one clock cycle, but new challenges arise when hardware and program behavior limit the amount of parallelism that can be extracted.

17.1 Superscalar Execution

Modern high-performance processors often implement *superscalar* execution, where multiple instructions are issued (i.e., started) in the same clock cycle to increase throughput. However, to realize sustained parallelism, several requirements must be met:



- **Fetch more instructions per cycle.**
  An instruction cache with sufficient bandwidth is needed to supply multiple instructions each cycle. Without adequate fetch capacity, the pipeline stalls and cannot exploit superscalar capabilities.
- **Commit more instructions per cycle.**
  To retire (commit) multiple instructions per cycle, the *reorder buffer* (ROB) and the *register file* must have enough ports and resources. A lack of commit bandwidth creates a bottleneck, undoing the benefits of parallel execution in earlier stages.
- **Obey data and control dependencies.**
  Even if hardware can fetch and commit multiple instructions per cycle, *data hazards* and *control hazards* must be respected. Modern superscalar designs typically use out-of-order (dynamic) scheduling to track dependencies and reorder instructions for higher throughput while maintaining correctness.

Despite advanced hardware techniques, data and control hazards impose fundamental limits on how much ILP can be extracted. In practice, balancing fetch, execute, and commit bandwidth with careful hazard management remains the central challenge of superscalar execution.

17.2 Dealing with Control Hazards

Superscalar processors can issue multiple instructions each cycle, but they are especially sensitive to branching instructions that disrupt the flow of fetched instructions. This section outlines several techniques that help mitigate the performance penalties of branches.

17.2.1 Dynamic Branch Prediction

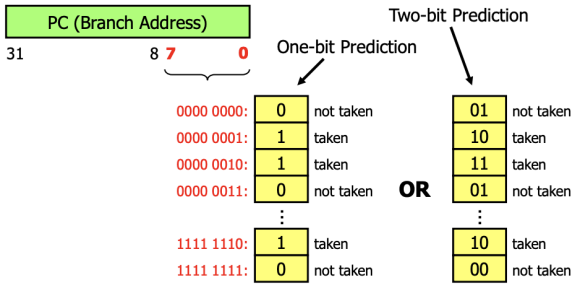
Branches create uncertainty: the processor needs to know which instruction addresses to fetch next. Two main problems arise when exploiting ILP:

- **True data dependencies**, which are unavoidable since some instructions inherently depend on the results of others.
- **Branches**, which determine *where* to look for further instructions.

Static Prediction
Early branch prediction methods (e.g., always-taken, never-taken, or simple compiler hints) often fail to anticipate dynamic behavior accurately because they cannot adapt to runtime conditions.
Dynamic Prediction
Dynamic branch prediction learns from past behavior. By using hardware structures that track how often a branch was taken, the predictor can adapt and refine its guesses over time, increasing overall accuracy and reducing wasted work from mispredicted branches.

17.2.2 Branch History Table (BHT)

A key component in many dynamic branch predictors is the *Branch History Table (BHT)*, which uses part of the *Program Counter* (PC) as an index. Each BHT entry stores one or more bits that represent the likelihood of a branch being taken or not taken.



- **Address Indexing:** The lower bits of the PC (e.g., bits 7:0) index into the BHT.
- **Prediction Storage:**
  - *One-bit Prediction:* Each entry stores a single bit:
    - `0`: Not taken
    - `1`: Taken
  - *Two-bit Prediction:* Each entry stores two bits:
    - `00`: Strongly not taken
    - `01`: Weakly not taken
    - `10`: Weakly taken
    - `11`: Strongly taken
- **Update Mechanism:** Predictions are updated based on whether the branch was actually taken, helping the hardware adapt to program behavior.

17.2.3 Speculative Execution

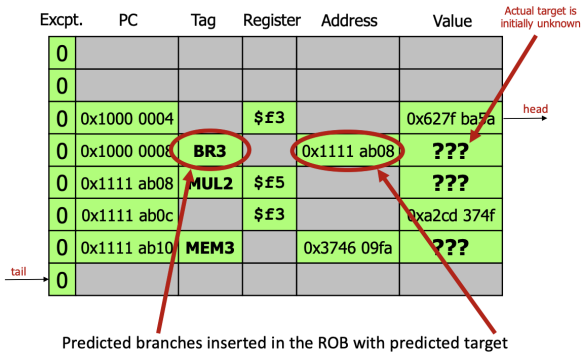
Speculative execution is a performance optimization that issues and executes instructions *before* a branch outcome is resolved. This reduces pipeline bubbles that occur when the processor must otherwise wait.

- **Dynamic Branch Prediction:** The processor fetches and decodes along the predicted path, using minimal resources so that incorrect speculations are easily discarded.

- **Results Handling:** Computed results remain *uncommitted* until the branch outcome is confirmed.
- **Recovery:** If a misprediction occurs, the processor *squashes* any partially executed instructions from the wrong path and resumes from the correct path.

17.2.4 Branches in the Reorder Buffer

Modern out-of-order processors track branch instructions and their outcomes in a *Reorder Buffer (ROB)*. Because branches are speculative, the ROB must handle both correct and incorrect predictions gracefully.

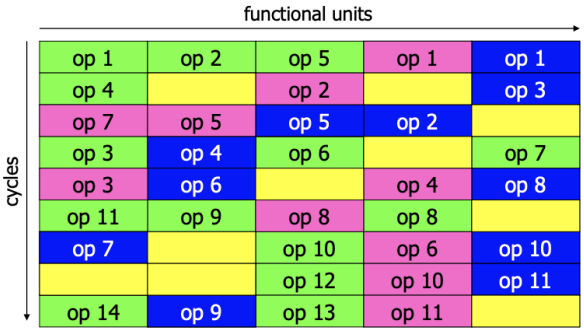


- **Correctly Predicted Branches:** When the actual branch target matches the prediction, the branch instruction is marked *resolved* and can be retired normally.
- **Mispredicted Branches:** If the actual branch target differs from the prediction, all instructions following that branch in the ROB are invalidated (*squashed*), and fetch restarts from the correct target address.

These mechanisms ensure the processor can continue running instructions *speculatively*, reaping the benefits of ILP while safeguarding correctness.

17.3 Beyond Superscalars: Simultaneous Multithreading (SMT)

When a superscalar pipeline is unable to find sufficient parallelism within a single thread, an alternative is to keep the hardware busy by issuing instructions from multiple threads. *Simultaneous Multithreading (SMT)* does exactly this, allowing multiple independent threads to utilize the same execution resources in parallel.



SMT vs. Single-Thread Superscalar

A dynamically scheduled superscalar typically manages only one thread at a time. It has:

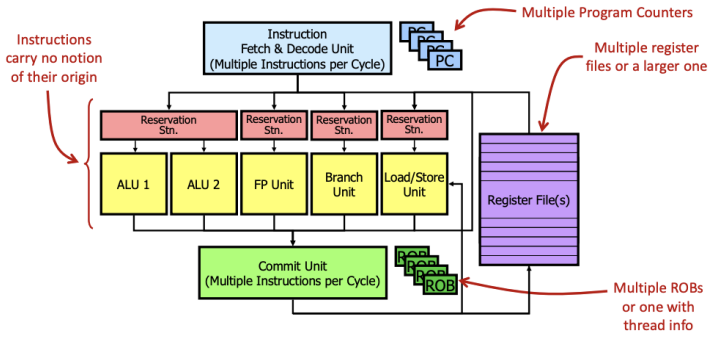
- A single set of *reservation stations* to track operand availability.
- One *reorder buffer* to handle out-of-order execution and in-order retirement.

Adding SMT Support

An SMT processor can simultaneously issue instructions from multiple threads in the same cycle:

- **Multiple PCs:** Each hardware thread needs its own program counter.
- **Extended Register File:** Either a separate set of registers per thread or a unified file with thread IDs.

- **Multiple or Extended ROBs:** Each thread must track and retire its instructions in order; thread IDs are often added to each entry.



Because thread instructions share the same functional units, SMT can achieve higher resource utilization, especially when one thread stalls or has limited ILP on its own. However, hardware becomes more complex, and caches and other shared structures must be carefully managed.

17.4 Memory Considerations: Nonblocking Caches

Even with a highly parallel core, performance may be bottlenecked by slow memory operations. If a load instruction misses in the cache, superscalar and SMT designs benefit from continuing other work rather than stalling immediately. *Nonblocking caches* help achieve this.

Example of Memory Stall

```
lw   $t2, 0($t0)     # t2 = mem[t0]
lw   $t3, 0($t1)     # t3 = mem[t1]
addi $t3, $t3, 123
andi $t3, $t3, 0xff
```

If the first load (`lw` $t2, 0$(`t0`)) results in a cache miss, a simple blocking cache would stall the pipeline until the data returns from memory. However, a *nonblocking cache* lets subsequent instructions continue if they do not depend on the missing data.

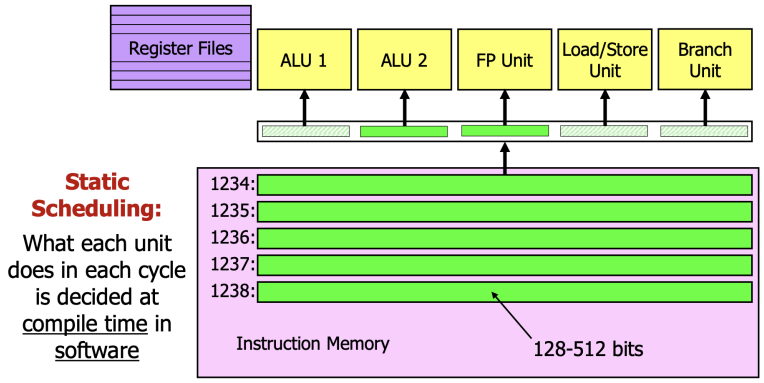
Hit Under Miss and Miss Under Miss

- **Hit Under Miss:** Serves new cache requests from different addresses while a miss is being resolved.
- **Miss Under Miss:** Handles multiple outstanding misses, overlapping latencies from the memory system.

These mechanisms greatly improve throughput by allowing the processor to exploit ILP (and multiple threads in SMT) while memory requests are pending.

17.5 VLIW: Very Long Instruction Word Architecture

VLIW (Very Long Instruction Word) architecture exploits *instruction-level parallelism* (ILP) by bundling multiple operations into a single long instruction word. Unlike pipelined processors, which execute instructions sequentially (albeit overlapped in time), VLIW delegates the responsibility of scheduling parallel operations to the compiler. The compiler groups independent operations that can be executed simultaneously into fixed-width instruction bundles.



17.5.1 Core Concepts

- **Fixed Instruction Format:** Each instruction consists of multiple slots, where each slot is assigned to a specific functional unit (e.g., arithmetic logic unit (ALU), memory unit).
- **Static Scheduling:** The compiler analyzes dependencies in the code and schedules instructions into bundles. Dynamic dependency checks and scheduling hardware are not needed.
- **Compiler-Driven:** The compiler must ensure that operations within one bundle are independent (or safe) to execute in parallel.

17.5.2 Example: VLIW vs. Pipelined Execution

Consider the following simple code fragment in a C-like pseudocode:

```
a = b + c;   // Operation 1
d = e - f;   // Operation 2
g = h * i;   // Operation 3
x = arr[j];  // Operation 4 (memory load)
```

Pipelined Processor Execution

In a pipelined processor, these instructions may be overlapped in execution stages (fetch, decode, execute, etc.). However, they are still issued one after another. For example:

1. **Cycle 1:** Fetch Operation 1
2. **Cycle 2:** Decode Operation 1, Fetch Operation 2
3. **Cycle 3:** Execute Operation 1, Decode Operation 2, Fetch Operation 3
4. **Cycle 4:** Execute Operation 2, Decode Operation 3, Fetch Operation 4
5. **Cycle 5:** Execute Operation 3, Execute Operation 4

The pipeline overlaps different stages of separate instructions but does not execute multiple operations simultaneously in the same cycle.

VLIW Processor Execution

Assume a simple VLIW processor with 3 functional units:

- ALU1: Arithmetic operations.
- ALU2: Arithmetic operations.
- MEM: Memory load/store operations.

The VLIW compiler analyzes dependencies and groups independent operations into a single long instruction. Given the code, the compiler may schedule as follows:

VLIW Instruction Format:

```
| ALU1   |  ALU2   |  MEM |
```

Scheduled VLIW Instructions:

1. **Cycle 1:**
   - `ALU1: ADD r1, r2, r3` (compute `a = b + c`)
   - `ALU2: SUB r4, r5, r6` (compute `d = e - f`)
   - `MEM: LOAD r7, [arr + r8]` (perform memory load for `x = arr[j]`)
2. **Cycle 2:**
   - `ALU1: MUL r9, r10, r11` (compute `g = h * i`)
   - `ALU2: NOP` (no operation)
   - `MEM: NOP` (no operation)

Key Differences

- **Instruction Issue:**
  - *Pipelined:* Issues one instruction per cycle and overlaps the stages.
  - *VLIW:* Issues a bundle of operations in one cycle, executing them concurrently.
- **Scheduling:**
  - *Pipelined:* Hardware manages instruction overlapping.
  - *VLIW:* The compiler statically schedules independent instructions into a single, long instruction word.

17.5.3 Summary

VLIW architectures shift complexity from hardware to the compiler, allowing multiple operations to execute in parallel within a single clock cycle. This enables efficient exploitation of ILP but requires careful compile-time analysis to ensure that parallel execution is both possible and correct.

Chapter 18

Part 5a. Multiprocessors Cache Coherence

In the last chapter, we focused on how to get the most out of a single processor by exploring advanced parallelism techniques. Now, we're moving to systems with multiple processors, where keeping their caches in sync is key to making everything work smoothly. This chapter introduces the basics of cache coherence and why it's so important in shared-memory systems.

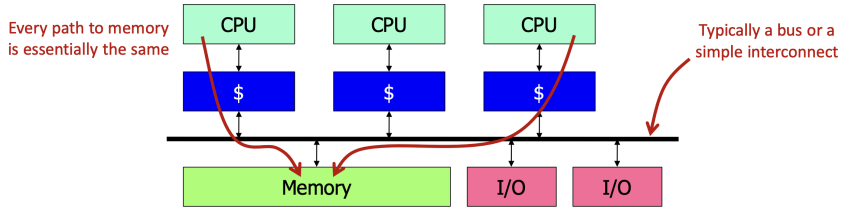
18.1 Flynn's Taxonomy (1966)

Flynn's Taxonomy classifies computer architectures based on the number of instruction streams and data streams they support. It is divided into the following categories:

- **SISD (Single Instruction, Single Data):** Represents uniprocessors where a single instruction stream operates on a single data stream. This is the traditional architecture of most early computers.
- **SIMD (Single Instruction, Multiple Data):** A single program executes on multiple data sets simultaneously. Classic examples include vector architectures used in high-performance computing, which are now less common. Modern x86 architectures support SIMD through various Instruction Set Architecture (ISA) extensions such as:
  - MMX (1996)
  - SSE (1999–2008)
  - AVX (2011–2016)
- **MIMD (Multiple Instruction, Multiple Data):** The general form of parallelism where each processor executes its own program on its own data. This is the most flexible and widely used parallel computing model.

18.1.1 Shared-Memory Multiprocessors (UMA)

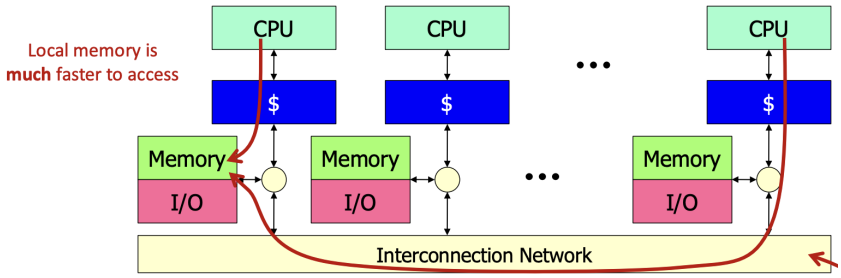
Uniform Memory Access (UMA) is a shared-memory multiprocessor architecture where all processors have equal access time to the shared memory. This architecture is characterized by:



- **Uniform memory access:** Every path to memory is essentially the same, ensuring consistent performance across all processors.
- **Simple interconnect:** Typically, a bus or a basic interconnect is used to connect CPUs, caches, memory, and I/O devices.
- **Limited scalability:** UMA systems generally support 4 to 16 processors due to bottlenecks in the interconnect and memory access.
- **Traditional design:** UMA represents a simple and fairly traditional multiprocessor architecture suitable for small-scale parallel systems.

18.1.2 Distributed-Memory Multiprocessors (NUMA)

Nonuniform Memory Access (NUMA) is a distributed-memory multiprocessor architecture where each processor has its own local memory.



- **Local memory access:** Each processor accesses its local memory much faster than the memory of other processors, leading to nonuniform memory access times.
- **Interconnection network:** Processors are connected through an interconnection network, which is often implemented as a real network, to enable communication and data exchange.
- **Scalability:** NUMA systems offer a more scalable and cost-effective way to grow the memory system, making them suitable for larger parallel systems.
- **Complex communication:** Communication between processors is more complex and involves higher latency compared to UMA systems.

18.1.3 Programming Paradigms

Parallel programming paradigms are classified based on how data is exchanged between processors. The two primary paradigms are:

- **Shared-Memory:**
  - Data is exchanged *implicitly* through shared variables in a common memory space.
  - Standard libraries (e.g., `OpenMP`) simplify programming.
  - Well-suited for shared-memory architectures (e.g., SMP, NUMA).
  - Can be implemented as *Distributed Shared Memory (DSM)* on systems with physically distributed memory, using virtual memory abstractions (e.g., `TreadMarks` for DSM; `Apache Spark` for DSM-like abstractions in big data).
- **Message Passing:**
  - Data is exchanged *explicitly* by sending and receiving messages over a network or interconnect.
  - Standard libraries (e.g., `MPI`) are widely used.
  - Natural for distributed-memory systems with private memory per processor.
  - Can also be implemented on shared-memory systems (e.g., NUMA), although it may introduce unnecessary overhead compared to native shared-memory programming.

18.1.4 Why (Hardware) Shared Memory?

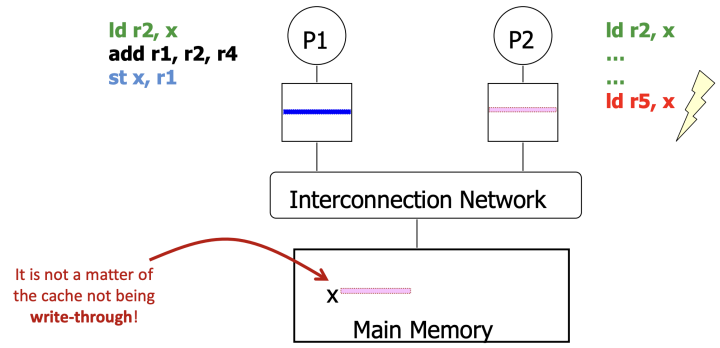
Shared memory provides a mechanism for parallel computing where multiple processors access a common memory space. Its advantages and disadvantages are as follows:

- **Advantages:**
  - Applications perceive it as a multitasking uniprocessor.
  - Requires only evolutionary extensions for operating systems.
  - Enables communication without relying on the operating system.
  - Simplifies software development by allowing correctness to be prioritized over performance.
- **Disadvantages:**
  - Communication is implicit, making optimization more challenging.
  - Synchronization between processors is complex.
  - Places implementation demands on hardware designers.
- **Result:**
  - *Symmetric Multiprocessors (SMPs):* Once the foundation of early supercomputers, SMPs have been largely replaced by distributed-memory message-passing systems due to scalability limitations.
  - *Chip Multiprocessors (CMPs) or Multicore Processors:* These dominate modern parallel computing, driving multibillion-dollar markets.

18.1.5 Cache Coherence and the Multi-Processor Problem

Cache coherence ensures that in a system with multiple processors, all caches have a *consistent* view of shared data. Without coherence mechanisms, processors could read or write stale data in their caches, leading to erroneous computation results.

Example Scenario:



1. **Processor P1 loads a value:** Suppose a shared variable `x` in main memory holds an initial value. Processor P1 loads `x` into its cache, so it now has a local copy of `x`.
2. **Processor P2 also accesses `x`:** Later, P2 tries to read `x` but experiences a cache miss (since `x` is not yet in P2's cache). It fetches `x` from main memory, storing this same value in its own local cache.
3. **P2 modifies `x`:** Processor P2 then updates `x` directly in its cache (for instance, `st x, r1`). Depending on the cache write policy (*write-back* vs. *write-through*), P2 may or may not immediately update main memory. **However, even if it does write to memory, P1's cache copy remains *unchanged*.**
4. **P1 sees stale data:** Because P1 already has a cached copy of `x`, it continues reading that older value (i.e., the one loaded earlier).

Why is this a problem? The issue here is that multiple cached copies of the same data must be kept in sync. If updates are performed in one cache without informing other caches, the system can quickly become *incoherent*. A processor might base calculations on an incorrect, outdated value of `x`, leading to unpredictable behavior or incorrect program outputs.

Importance of Coherence Protocols: To prevent such inconsistencies, cache coherence protocols enforce invalidation or updating of stale cache lines. When one processor modifies `x`, coherence messages are sent over the interconnection network so that other caches either invalidate their copies or update them with the new data. This ensures that all processors always see a consistent view of shared memory.

18.1.6 Ensuring a Coherent Memory System

A *coherent memory system* guarantees that every processor observes all shared-memory operations (reads and writes) in a manner that is logically consistent with a single, shared view of memory. The goals of such a system typically boil down to the following three properties:

1. **Preservation of program order.** If a processor P writes to a location `X` and then (without any intervening writes from other processors) reads `X`, it must read back the value that it just wrote. This ensures that each processor's own writes are visible to itself in program order.
2. **Coherent view (read values).** If processor P_1 writes `X`, and processor P_2 then reads `X`, with no other intervening writes to `X`, P_2 must see the value written by P_1. Essentially, a read in one processor cannot observe an older version of `X` if a newer version exists in the system and there have been no conflicting writes.
3. **Write serialization.** If multiple processors write to `X` (e.g., P_1 writes, then P_2 writes, etc.), all processors must observe these writes in the same order. Thus, if P_1 writes to `X` first and P_2 writes to `X` second, then no processor should be able to observe P_2's write before P_1's write.

How do we achieve coherence in practice?

- **Hardware Protocols:** Coherence is typically enforced via specialized hardware protocols (e.g., MESI, MOESI) that track and coordinate the states of cache lines. When a processor writes to `X`, the protocol ensures other copies of `X` are either invalidated or updated.
- **Snooping or Directory-Based Approaches:** In *snooping* protocols, all caches monitor a shared bus to detect writes and invalidate outdated copies. In *directory-based* protocols, a central directory keeps track of which caches hold each line, allowing precise invalidation or update messages.
- **Preserving Order:** A coherence protocol enforces the three properties above by establishing rules for when a cache line can be read, written, invalidated, or shared. This ensures every processor eventually sees writes in the correct order and never operates on stale data.

By carefully orchestrating which cache copy is valid and who has the authority to write to it, a coherent memory system prevents the classic inconsistencies shown in our earlier example. Processors remain synchronized on shared data values, avoiding stale reads and enabling correct parallel execution.

18.1.7 Snoopy Cache-Coherence Protocols

Snoopy cache-coherence protocols rely on a shared bus to serialize memory transactions and ensure data consistency across multiple caches. Each cache controller *snoops* all bus transactions and compares them against the cache lines it currently holds.

Basic Operation:

- **Bus as a Serialization Point:** All memory requests issued by processors appear on the shared bus, providing a single global ordering.
- **Snooping:** Each cache controller listens (*snoops*) to every bus transaction. If a transaction concerns a cache line that the controller contains, it takes steps to maintain coherence.

Coherence Actions: When a transaction targets a cache line, the responsible cache controller can:

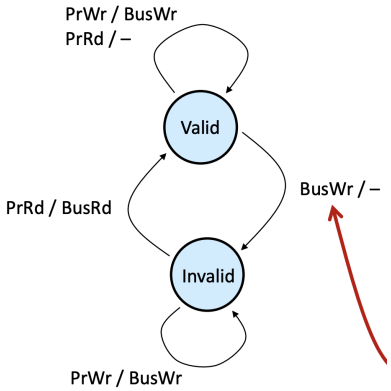
- *Invalidate* a stale copy of the line.
- *Update* its local line if another cache provides new data.
- *Supply value* to another cache or to memory.

The specific action taken depends on the protocol's finite state machine (FSM), which tracks the line's state (e.g., *Modified*, *Shared*, *Invalid*, etc.).

Simultaneous Controllers:
Each cache operates its snooping logic independently but concurrently. Because they all observe the same bus traffic, conflicts and updates are detected quickly, preserving coherence across the system.
This bus-based *snoopy* approach is conceptually simpler than directory-based methods and is effective for a moderate number of processors sharing a single bus. However, as system scale increases, the performance overhead of snooping and bus contention may become a limiting factor.

18.1.8 Simple Invalidate Snooping Protocol

The Simple Invalidate Snooping Protocol is a cache coherence protocol designed for write-through, write-no-allocate caches. It operates with two states: **Valid** and **Invalid**.
Transitions between these states are governed by processor and bus actions.



- **Valid State**:
  - A `PrWr` (Processor Write) results in a `BusWr` (Bus Write) operation.
  - A `PrRd` (Processor Read) requires no bus action.
- **Invalid State**:
  - A `PrRd` triggers a `BusRd` (Bus Read) to fetch data into the cache.
  - A `PrWr` results in a `BusWr`.
- When another processor writes to the same cache line, a `BusWr` is broadcast, transitioning the state from **Valid** to **Invalid**.
- A `BusRd` can transition the state from **Invalid** to **Valid**.

The protocol ensures coherence by invalidating or updating cache lines in response to snooped bus operations. This mechanism is essential in multiprocessor systems where caches are shared.
Note: The snooping mechanism actively monitors the bus to maintain coherence.

18.1.9 3-State Write-Back Invalidation Protocol (MSI)

The *3-State Write-Back Invalidation Protocol (MSI)* is used to maintain cache coherence in multiprocessor systems. It introduces three states for cache lines:

- **Modified (M):**
  - The cache line contains the latest copy of the data.
  - The memory is stale and not up-to-date.
  - Only one cache can have this state for a given line.
- **Shared (S):**
  - The cache line contains a valid copy of the data.
  - One or more caches may hold the same data in this state.
- **Invalid (I):**
  - The cache line is not valid and must be fetched from memory or another cache.

Features:

- Before entering the *Modified* state, all other copies of the cache line must be invalidated.
- Ensures cache coherence by enforcing bus transactions to maintain order and perform invalidations.

Comparison with 2-State Protocol:

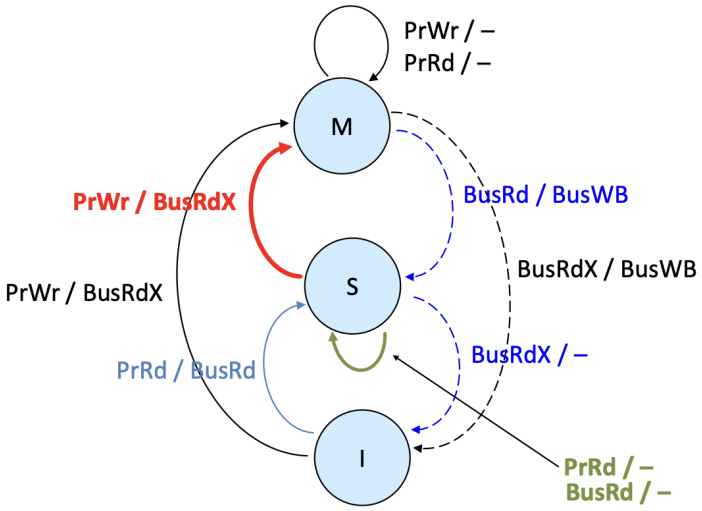
- The 2-state protocol is simpler but less efficient, as every write operation requires a broadcast on the bus.
- MSI resolves coherence issues effectively, but it can lead to higher performance overhead due to bus transactions.

This protocol ensures coherence but can impact performance, particularly in systems with high contention for the memory bus.

18.2 MSI Protocol

The *Modified, Shared, Invalid* (MSI) protocol is a classic cache coherence protocol used in multiprocessor systems with write-back caches. Its goal is to ensure that all processors observe a consistent view of memory, even though multiple caches may hold copies of the same memory block. In MSI, each cache block can be in exactly one of three states at any time:

- **M (Modified)**: The cache block holds the only valid (and most recent) copy of the data, and this copy has been modified with respect to main memory. Main memory is thus *stale* until this block is written back.
- **S (Shared)**: One or more caches may contain valid copies of the data. The copy in main memory is also valid (i.e., the same as the cache blocks).
- **I (Invalid)**: The cache block is not valid. The data in this block must not be used without first fetching a valid copy from memory or another cache.



The protocol enforces coherence by requiring certain *bus transactions* on reads and writes, which can cause cache blocks to transition from one state to another. Typical bus signals include:

- **BusRd (Bus Read)**: A read request *without* intent to modify.
- **BusRdX (Bus Read Exclusive)**: A read request *with* intent to modify (also called *Read For Ownership*); it invalidates any other copies.
- **BusWB (Bus Write Back)**: A cache writes its modified block back to memory (or supplies it to another cache) when it must give up ownership.

Below is a concise description of the main state transitions (processor actions are prefixed with `Pr` and bus actions with `Bus`):

- **I → S**: Occurs on a `PrRd`, which triggers `BusRd` if the block is not present in any cache (or must be fetched from memory). The cache obtains a shared copy.
- **I → M**: Happens on a `PrWr`, leading to `BusRdX`. All other caches invalidate their copies before one cache transitions to Modified.
- **S → M**: On a `PrWr` to a shared block, the cache issues `BusRdX`, invalidating other shared copies and gaining exclusive (modified) ownership.
- **M → S**: If another processor performs a read (`BusRd`) while the block is in M, the current cache must supply the data via `BusWB`, and the block transitions to S (now shared among caches).
- **M or S → I**: An `Invalidate` request (triggered by someone else's `BusRdX`) or a coherence miss can force the local copy to become Invalid.

By following these rules, the MSI protocol ensures that at most one cache holds a **Modified** copy and that any other copies are either **Shared** or **Invalid**. This guarantees coherence across all caches and maintains the illusion of a single, consistent memory.