Architettura dei calcolatori - Riassunto libro "Computer Organization and Design - RISC V"

Architettura dei calcolatori (Università degli Studi di Modena e Reggio Emilia)

Chapter 1

Computer Abstractions and Technology

Traditional Classes of Computing Applications

Personal Computers (PCs): computer designed for use by an individual.
Drove the evolution of computing technologies.

Servers: computer used for running larger programs for multiple users (often simultaneously), typically accessed via network.
Oriented to carrying sizeable workloads.

Supercomputers: class of computers with the higher performance and cost.
Used for high-end scientific and engineering calculations.

Embedded Computers: computer inside another device used for running one predetermined application or collection of software.
Largest class of computers, span the widest range of applications.

Post-PC Era:

Personal Mobile Device: (smartphone and tablets) small wireless devices to connect to the internet, software is installed by downloading apps.

Cloud Computing: large collection of servers that provides services to the Internet.
Relies on datacenters known as Warehouse Scale Computers.

Software as a service: delivers software and data as a service over the Internet

Below Your Program

System software: software that provides services that are useful like operating systems, compilers, loaders, assemblers.

Operating system: supervising program that manages the resources of a computer for the benefit of the programs that run on the computer.

- Handles basic input and output operations
- Allocates storage and memory
- Provides for protected sharing of the computer among multiple applications using it simultaneously.

Compilers: program that translates high-level language statements into assembly language statements.

Assembler: program that translates a symbolic version of the instruction (assembly language) into the binary version (machine language).

Under the Covers

Classic components of a computer:

- **Input:** mechanism that feds information to the computer.

- **Output:** mechanism that conveys the result to a user.
  Graphic display (Liquid Crystal Display): output device that uses and *active matrix* that has a transistor switch at each *pixel* (the smallest individual picture element) to control the transmission of light.
  The image is composed of a matrix of bits called bit map which is stored in the frame buffer and read out to the graphics display at the refresh rate.

- **Memory:** storage area in which the programs are kept when they are running and that contains the data needed by the running programs.
  It is built from ***DRAM (dynamic random access memory***, provides random access to any location) chips.

Cache memory: slower memory that acts as a buffer, it is built from *SRAM (static random access memory)* which is less dense and faster.

CPU (central processing unit or processor)

- **Datapath:** component of the processor that performs arithmetic operations.

- **Control:** part of the processor that commands the datapath, memory, I/O devices according to the program instruction.

ISA

Instruction Set Architecture: abstract interface between the hardware and the lowest-level software that encompasses all the information necessary to write a machine language program that will run correctly.
Application Binary Interface: combination of the instruction set and the operating system interface provided for application programmers. It defines a standard for binary portability across computers.
Implementation: hardware that obeys to the architecture abstraction.

A Safe Place for Data

Volatile memory: retains data only if receiving power.

Main memory: used to hold programs while they are running (DRAM)

Non-volatile memory: retains data even in the absence of power.

Secondary memory: used to store programs between runs (hard disk, flash memory)

Communicating with Other Computers

- **Communication:** Information exchanged between computers at high speed.
- **Resource sharing:** computer on the network can share I/O devices.
- **Nonlocal access:** connecting computers over long distances.

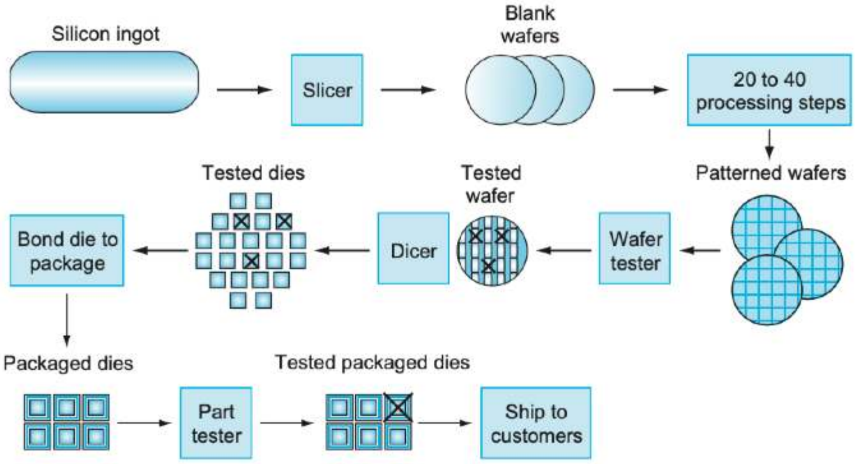
Technologies for Building Processors and Memory

Transistor: on/off switch controlled by an electric signal.
IC combines transistors on a single chip.

Possible to add materials to silicon that allow tiny areas to transform into:

- *Excellent conductors of electricity*
- *Excellent insulators from electricity*
- *Areas that can conduct or insulate under specific conditions*

Manufacturing process for integrated circuits is critical to the cost of the chips.



Defects: flaws in a wafer that can result in the failure of the die (individual section cut from a wafer known as chip).
Dicing enables to discard only those dies that contain the flaws.

Yield: percentage of good dies from the total number of dies on the wafer.

Cost of an IC:

$$\text{Cost per die} = \frac{\text{Cost per wafer}}{\text{Dies per wafer} \times \text{yield}}$$

$$\text{Dies per wafer} \approx \frac{\text{Wafer area}}{\text{Die area}}$$

$$\text{Yield} = \frac{1}{(1 + (\text{Defects per area} \times \text{Die area}/2))^2}$$

Performance

- **Response time (Execution time):** total time required for the computer to complete a task.
- **Throughput:** total amount of tasks completed per unit time.

CPU time: actual time the CPU spends computing for a specific task.
User CPU time: time spent in a program.
System CPU time: time spent in the operating system performing tasks on behalf of the program.

Clock cycle: time for one ***clock period*** (length of a clock cycle), which run at a constant rate.

$$\begin{matrix}\text{CPU execution time}\\ \text{for a program}\end{matrix} = \begin{matrix}\text{CPU clock cycles}\\ \text{for a program}\end{matrix} \times \text{Clock cycle time}$$

Alternatively, because clock rate and clock cycle time are inverses,

$$\begin{matrix}\text{CPU execution time}\\ \text{for a program}\end{matrix} = \frac{\text{CPU clock cycles for a program}}{\text{Clock rate}}$$

Clock Cycles per Instruction (CPI): average number of clock cycles per instruction for a program.

$$\text{CPU clock cycles} = \text{Instruction for a program} \times \text{Average clock cycles per instruction}$$

Instruction count: number of instructions executed by the program.

$$\text{CPU time} = \text{Instruction count} \times \text{CPI} \times \text{Clock cycle time}$$

$$\text{CPU time} = \frac{\text{Instruction count} \times \text{CPI}}{\text{Clock rate}}$$

The Power Wall

- **Dynamic Energy:** energy consumed when transistors switch state.
  It depends on the capacitive loading of each transistor and the voltage applied.
  Problem: remove the heat cooling the chip BUT we can't lower the voltage too much.

$$\textit{Energy} \propto \textit{Capacitive load} \times \textit{Voltage}^2$$

The energy of a single transition is then

$$\textit{Energy} \propto 1/2 \times \textit{Capacitive load} \times \textit{Voltage}^2$$

The power required per transistor is just the product of energy of a transition and the frequency of transitions:

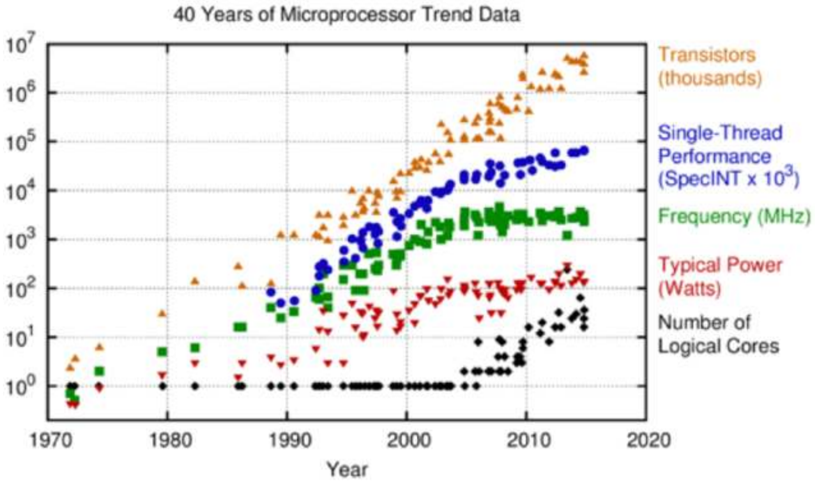
$$\textit{Power} \propto 1/2 \times \textit{Capacitive load} \times \textit{Voltage}^2 \times \textit{Frequency switched}$$

- **Static Energy:** energy consumed because of leakage current that flows even when a transistor is off.
  Increasing the number of transistors increases power dissipation.

Switch from Uniprocessor to Multiprocessor

Since the power wall response time has decreased so we need to switch from single processors to microprocessors with multiple processors or core.

- **Core:** processors placed on a microprocessor.



Benchmarking

SPEC CPU Benchmark

To evaluate two computer systems we need to compare the execution time of the workload.

- **Workload:** set of programs run on a computer that is either the actual collection of applications run by a user or constructed from real programs to approximate such mix.
- **Benchmark:** program selected for use in comparing computer performance.
  They form a workload that will predict the performance of the actual workload.

SPEC (System Performance Evaluation Cooperative): created a sets of benchmarks for modern computer systems.

$$\sqrt[n]{\prod_{i=1}^{n} \text{Execution time ratio}_i}$$

SPEC Power Benchmark

Specific benchmark to measure power.

Fallacies and Pitfalls

- **Pitfall:** Expecting the improvement of one aspect of a computer to increase overall performance by an amount proportional to the size of improvement.
  Amdahl's Law: The opportunity for improvement is affected by how much time the vent consumes (Make the common case fast).

- **Fallacy:** Computers at low utilization use little power.

- **Fallacy:** Designing for performance and designing for energy efficiency are unrelated goals.
  Energy is power over time, often hardware and software optimization that take less time save energy overall even if the optimization takes more energy when used.

- **Pitfall:** Using a subset of the performance equation as a performance metric.
  MIPS (million instruction per second): measurement of program execution speed based on the number of millions of instruction.

$$\text{MIPS} = \frac{\text{Instruction count}}{\text{Execution time} \times 10^6}$$

$$\text{MIPS} = \frac{\text{Instruction count}}{\dfrac{\text{Instruction count} \times \text{CPI}}{\text{Clock rate}} \times 10^6} = \frac{\text{Clock rate}}{\text{CPI} \times 10^6}$$

Chapter 2

Instructions: Language of the Computer

Operands of the Computer Hardware

1. **Design Principle: Simplicity favours regularity.**

Every arithmetic instruction has 3 operands (two operands added and a place where to put the sum)

1. **Design Principle: Smaller is faster.**

Limited number of operands.
Registers: primitives used in hardware design.
RISC-V has 32-64 bits registers (64 bits = doubleword) numbered from 0 to 31 prefixed by X.
Registers can contain only small amount of data.

Memory Operands

Data transfer instructions: command that moves data between memory and registers.
Address: value used to delineate the location of a specific data element within a memory array.

- *Load*: data transfer instruction that copies data from memory to a register.

```
ld   x9, 64 (x22)
```

x22: array's base address.
64: offset (must be multiplied by 8).
x9: register to be loaded.

- *Store*: instruction that copies data from a register to memory.

```
sd   x9, 96 (x22)
```

x22: array's base address.
98: offset (element number 12 multiplied by 8).
x9: register to be stored.

Attention! More variables than registers, keep the most frequently used ones in registers and place the rest in memory.
Process of putting less frequently used variables from memory into registers is called ***spilling registers***.

Constant or Immediate Operands

Constant: an operation may use a constant to perform quicker operations.
In an arithmetic operation one operand may be a constant which is called immediate.

```
addi x22, x22, 4
```

x22: rd.
x22: rs1.
4: immediate.

Signed and Unsigned Numbers

Binary digit (BIT): one of the two numbers in base 2, 0 or 1, that are the components of information.
Fundamental building block that can be low or high, on or off, true or false, 1 or 0.

- **Least Significant Bit (LSB):** rightmost bit.
- **Most Significant Bit (MSB):** leftmost bit.

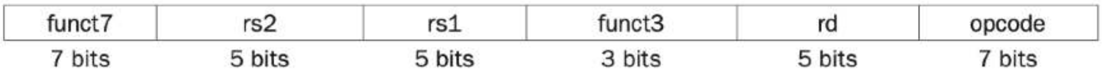
Overflow: occurs when adding two numbers that give a result that can't be represented by the hardware.
Underflow: occurs when adding two negative numbers that give a result that can't be represented.

Two's complement: 0 means positive, 1 means negative. Bit flip, add 1.
(2^n-1): positive numbers.
-2^n: negative numbers.

Representing Instructions in the Computer

Instruction Format: form of representation of an instruction composed of fields of binary numbers.
Machine Language: binary representation used for communications within a computer system (numeric version of the instruction).

RISC-V Fields

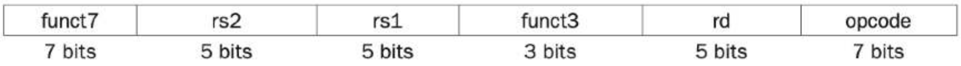


funct7	rs2	rs1	funct3	rd	opcode
7 bits	5 bits	5 bits	3 bits	5 bits	7 bits

- ***Opcode:*** field that denotes the operation and format of an instruction.
- ***rd:*** register destination.
- ***rs1:*** first register source operand.
- ***rs2:*** second register source operand.
- ***funct3/funct7:*** additional opcode field.

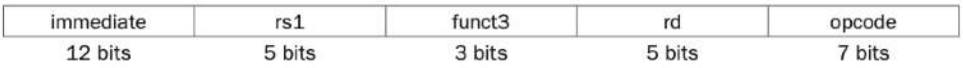
Design principle 3: good design demands good compromises.
Keep all instruction the same length using distinct instruction formats for different kinds of instructions.

R-Type (Register Type)



funct7	rs2	rs1	funct3	rd	opcode
7 bits	5 bits	5 bits	3 bits	5 bits	7 bits

I-Type (Immediate Type)

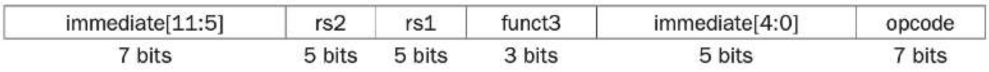


immediate	rs1	funct3	rd	opcode
12 bits	5 bits	3 bits	5 bits	7 bits

Immediate: interpreted as a two's complement value (-2^{11}, 2^{11}-1).

When used for load instruction the immediate represents a byte offset (can refer to a doubleword with a region of + -2^{11}, or 2048 bytes of the base address in the base register rd.

S-Type (Store Type)



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

12 bit immediate is split into two fields to keep the rs1 and rs2 feels in the same place (reduces hardware complexity).

Logical Operations

- **Shift**: operation that moves all the bits to the left or the right, filling the emptied bits with 0s.
  - *Shift left logical immediate (slli):* like multiplying.
  - *Shift right logical immediate (srli):* like dividing.

```
slli  x11, x19, 4
```

`x11`: rd.
`x19`: rs1.
4: immediate.

- **AND:** 1 only if there's a 1 in both operands.

```
and   x9, x10, x11
```

- **OR:** 1 if there is a 1 in either operand.

```
or    x9, x10, x11
```

- **XOR:** calculates 1 only if the values are different (instead of NOT).

```
xor   x9, x10, x12
```

Instructions for Making Decisions

Conditional branches: instruction that tests a value and, based on the outcome of the test, transfers control to a new address in the program.

- **Branch if equal:**

```
beq  rs1, rs2, L1
```

- **Branch if not equal:**

```
bne  rs1, rs2, L1
```

- **Branch if less than:**

```
blt  rs1, rs2, L1 (rs1 < rs2 takes the branch)
     bltu (rs1/2 treated as unsigned numbers)
```

- **Branch if greater than or equal:**

```
bge  rs1, rs2, L1 (rs1 >= rs2 takes the branch)
     bgeu (rs1/2 treated as unsigned numbers)
```

Basic block: sequence of instruction without branches, branch target or branch labels.

Supporting Procedures in Computer Hardware

Procedure: stored subroutine that performs a specific task based on the parameters which it is provided. Can pass values and return results. Allows code to be reused.

Steps:

1. Put parameters where the procedure can access them.
2. Transfer control to procedure.
3. Acquire storage resources needed for the procedure.
4. Perform the task.
5. Put the result value where the calling program can access it.
6. Return control.

- **x10-x17:** parameter registers in which to pass parameters and return values.
- **x1:** return address register to return to the point of origin.

Jump-and-link: instruction that branches to an address and saves the address of the following instruction to the destination register rd.

```
jal  x1, ProcedureAddress
```

x1: return address

```
jal  x0, Label Unconditional branch
```

Return address: link to the calling site that allows a procedure to return to the proper address stored in x1 (Program counter + 4: following instruction).

Jump-and-link-register: instruction that branches to the address stored in register x1 (branches to the return address, returns to the caller).

```
jal  x0, 0(x1)
```

Program Counter (PC): register containing the address of age instruction being executed in the program.

Using More Registers

Stack: data structure for spilling register organized as a last-in-first-out queue (grows from higher to lower addresses).

Stack pointer (sp, x2): value denoting the most recent allocated address in a stack (pointer that shows where the next procedure should place the registers to be spilled or where old register values are found).

- **x2:** stack pointer register (sp).

Push: add element.
Pop: remove element.

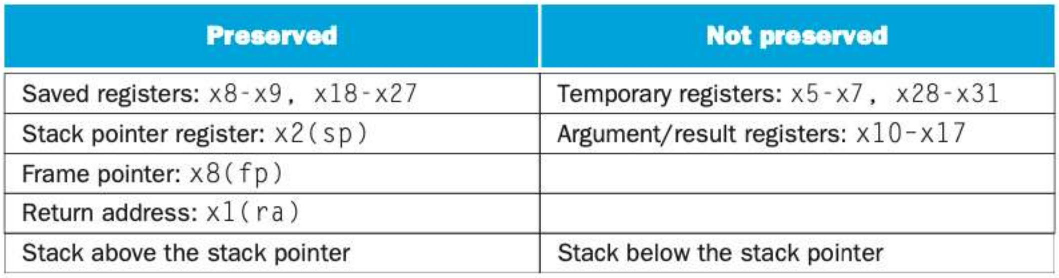
- **X5-x7, x28-x31:** temporary registers not preserved by the callee.
- **X8-x9, x18-x27:** saved registers that must be preserved on a procedure call (saved and restored by the callee).

Nested Procedures

Non-Leaf procedure: procedure that calls other procedures.
Need to save on the stack the return address, arguments and temporaries needed after the call.

```
EXAMPLE

fact:
     addi sp, sp, -16      // adjust stack for 2 items
     sd x1, 8(sp)          // save the return address
     sd x10, 0(sp)         // save the argument n
     addi x5, x10, -1      // x5 = n - 1
     bge x5, x0, L1        // if (n - 1) >= 0, go to L1
     addi x10, x0, 1       // return 1
     addi sp, sp, 16       // pop 2 items off stack
     jalr x0, 0(x1)        // return to caller
L1:  addi x10, x10, -1     // n >= 1: argument gets (n − 1)
     jal x1, fact          // call fact with (n − 1)
     addi x6, x10, 0   // return from jal: move result of fact (n - 1) to x6:
     ld x10, 0(sp)         // restore argument n
     ld x1, 8(sp)          // restore the return address
     addi sp, sp, 16       // adjust stack pointer to pop 2 items
     mul x10, x10, x6      // return n * fact (n − 1)
     jalr x0, 0(x1)        // return to the caller
```

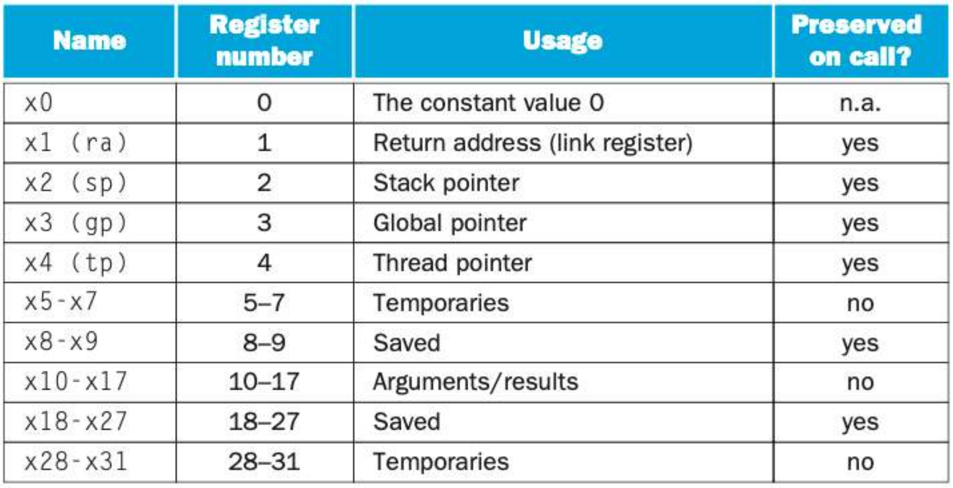


Preserved	Not preserved
Saved registers: x8-x9, x18-x27	Temporary registers: x5-x7, x28-x31
Stack pointer register: x2(sp)	Argument/result registers: x10-x17
Frame pointer: x8(fp)	
Return address: x1(ra)	
Stack above the stack pointer	Stack below the stack pointer

Allocating Space for New Data on the Stack

Procedure frame (activation record): segment of the stack containing a procedure's saved registers and local variables.

Frame pointer (fp, x8): value denoting the location of the saved registers and local variables for a given procedure.



Name	Register number	Usage	Preserved on call?
x0	0	The constant value 0	n.a.
x1 (ra)	1	Return address (link register)	yes
x2 (sp)	2	Stack pointer	yes
x3 (gp)	3	Global pointer	yes
x4 (tp)	4	Thread pointer	yes
x5-x7	5–7	Temporaries	no
x8-x9	8–9	Saved	yes
x10-x17	10–17	Arguments/results	no
x18-x27	18–27	Saved	yes
x28-x31	28–31	Temporaries	no

Communicating with People

ASCII: American Standard Code for Information Interchange.
Unicode: universal encoding of alphabets.

- `lbu:` load byte unsigned, loads byte from memory placing it in the 8 rightmost bit of a register.
- `sb`: store byte, takes a byte from the rightmost 8 bit of a register and writes it to memory.
- `lhu:` load half unsigned, loads halfword from memory placing it in the 16 rightmost bit of a register.
- `sh:` load half unsigned, takes halfword from the rightmost 16 bit of a register and writes it to memory.

RISC V:

- **Load** byte/halfword/word con segno esteso per occupare bit vuoti
  - `lb    rd, offset (rs1)`
  - `lh    rd, offset (rs1)`
  - `lw    rd, offset (rs1)`
- **Load unsigned** byte/halfword/word ***0 esteso a 64 bit***
  - `lbu    rd, offset (rs1)`
  - `lhu   rd, offset (rs1)`
  - `lwu    rd, offset (rs1)`
- **Store**
  - `sb    rs2, offset (rs1)`
  - `sh    rs2, offset (rs1)`
  - `sw    rs2, offset (rs1)`

Addressing for Wide Immediates and Addresses

Wide Immediate Operands

Lui (Load upper immediate): loads a 20-bit constant into bits 12-31 of a register.
Can create a ***32-bit constant*** with two instructions (`lui`: load bits 12-31, `addi`: load rightmost 12 bits).
Uses the U-Type instruction format.

Addressing in Branches

Branch instructions uses **SB-Type** instruction format.
Can branch addresses from -4096 to 4094 (2^{12}).

SB-Type

0	111110	01011	01010	001	1000	0	1100111
imm[12]	imm[10:5]	rs2	rs1	funct3	imm[4:1]	imm[11]	opcode

Unconditional jump-and-link instruction (`jal`) uses **UJ-Type** instruction format (20 bit address immediate).
Can jump to to addresses not bigger than 2^{20}.

UJ-Type (Unconditional Jump Type)

0	1111101000	0	00000000	00000	1101111
imm[20]	imm[10:1]	imm[11]	imm[19:12]	rd	opcode

Program counter = Register + Branch offset

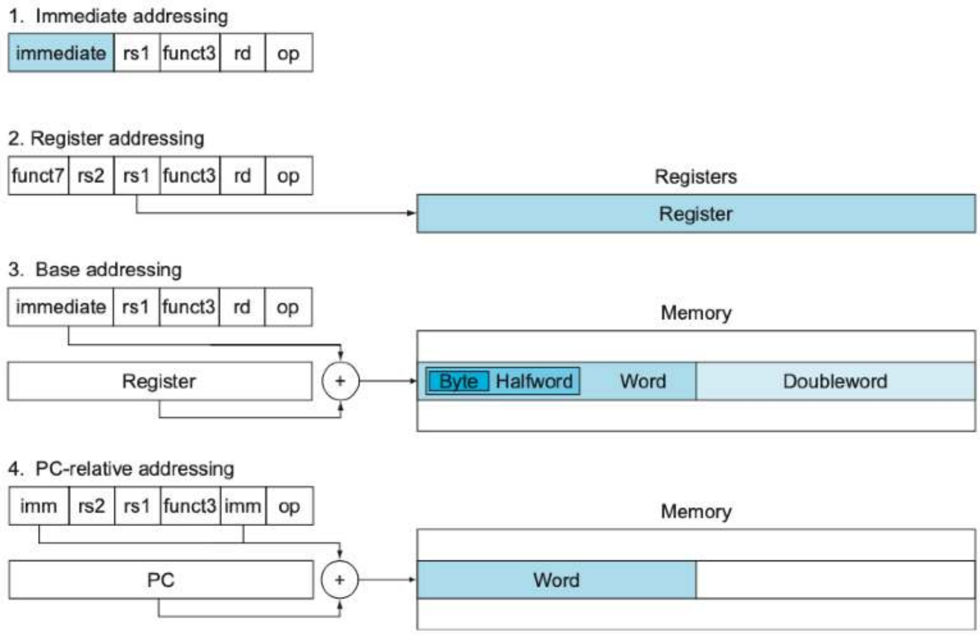
Programs can be as large as 2^{64}.

PC relative addressing: address is the sum of the program counter and a constant in the instruction.
Offset is relative to the actual PC.
Target Address = PC + Immediate*2

Jumping further: combining `lui` (writes bits 12-31) ad `jalr` (adds lower 12 bits and jumps to the sum).

RISC V Addressing Mode Summary

- **Immediate addressing**: operand is a constant within the instruction.
- **Register addressing:** operand is a register.
- **Base or displacement addressing:** operand is at the memory location (address is the sum of a register and a constant).
- **PC relative addressing:** branch address is the sum of the PC and a constant.



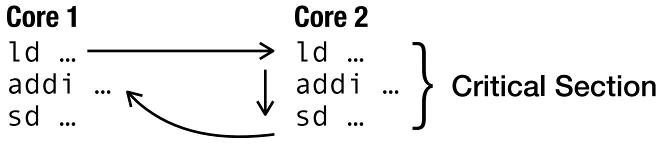
Name	Field						Comments
(Field size)	7 bits	5 bits	5 bits	3 bits	5 bits	7 bits	
R-type	funct7	rs2	rs1	funct3	rd	opcode	Arithmetic instruction format
I-type	immediate[11:0]		rs1	funct3	rd	opcode	Loads & immediate arithmetic
S-type	immed[11:5]	rs2	rs1	funct3	immed[4:0]	opcode	Stores
SB-type	immed[12,10:5]	rs2	rs1	funct3	immed[4:1,11]	opcode	Conditional branch format
UJ-type	immediate[20,10:1,11,19:12]				rd	opcode	Unconditional jump format
U-type	immediate[31:12]				rd	opcode	Upper immediate format

Parallelism and Instructions: Syncronization

Parallel Execution: two processors sharing an area of memory.
Multi cores means *multi thread code* that implement the same application: faster but problem with data coming from the shared memory (*shared resource*).
Data Race: two memory access form a data race if they are from different thread to the same location (creates inconsistency problems cause result depends on order of access).

Solution: hardware support for ***atomic*** read/write memory operation (nothing can interpose between the read and write of the memory location).

`Example:`



Lock/Unlock:
Busy = 1
Free = 0
A processor tries to set the lock doing an exchange of 1. If the value returned is 1 another processor has already claimed access. If the value returned is 0 the value is changed to 1 to prevent any other processor to retrieve a 0.

Syncronization in RISC-V

Two instruction used in sequence:

- **lr.d (load-reserved doubleword)**

```
lr.d       rd, (rs1)
```

- **sc.d (store-conditional doubleword)**

```
sc.d       rd, rs2, (rs1) // rs1 must have the same value of lr.d
```

If the contents of the memory location specified by the load-reserved are changed before the store-conditional to the same address occurs, the store conditional fails and does not write the value to memory.

***Store-conditional* fails**: returns nonzero value in rd.
***Store-conditional* succeed**: returns 0 value in rd.

```
Example:
```

Atomic Swap (to test/set lock variable)

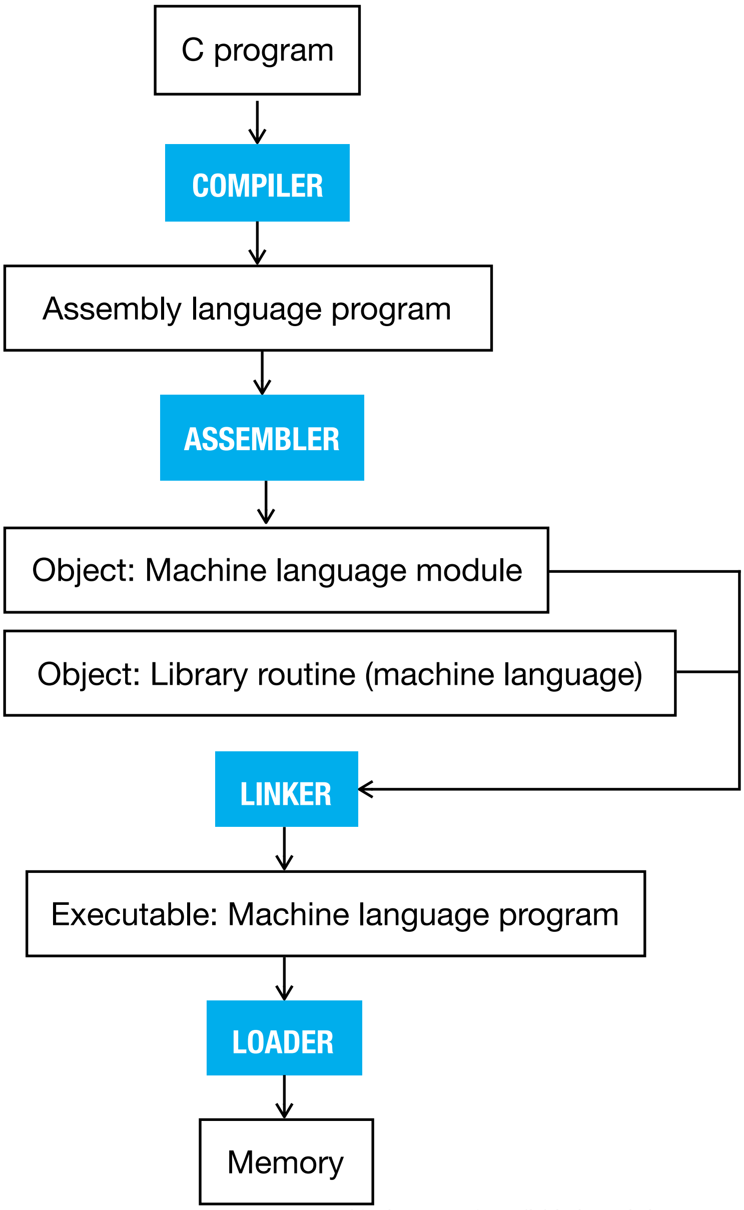
```
again:   lr.d  x10, (x20)       // load-reserved
         sc.d  x11, x23, (x20)  // store-conditional x11 = status
         bne   x11, x0, again   // branch if store fails
         addi  x23, x10, 0      // x23 = loaded value
```

Spin lock (keeps spinning as long as it is busy, execution when free)

```
         addi  x12, x0, 1       // copy locked value
again:   lr.d  x10, (x20)       // load-reserved to read lock
         bne   x10, x0, again   // check if it is 0
         sc.d  x11, x12, (x20) // attempt to store new value
         bne   x11, x0, again   //branch if store fails
         sd    x0, 0(x20)       // free lock by writing 0
```

Translating and Starting a Program

Translation hierarchy:


C program
COMPILER
Assembly language program
ASSEMBLER
Object: Machine language module
Object: Library routine (machine language)
LINKER
Executable: Machine language program
LOADER
Memory

COMPILER: transforms the program into an assembly language program.

LINKER: links pending reference in the code with libraries to complete it (resolved all undefined labels).

Executable: loaded from secondary memory to DRAM.

LOADER: does addresses translation.

Linking:

Static linking: library is part of the executable.
Fast but problems:

- if a new version is released the program uses the old version
- All routines that are called in the executable are loaded even the ones that are not executed.

Dynamic linking: library routines are linked and loaded only when the program is running.
Link/load library procedure when it is called.

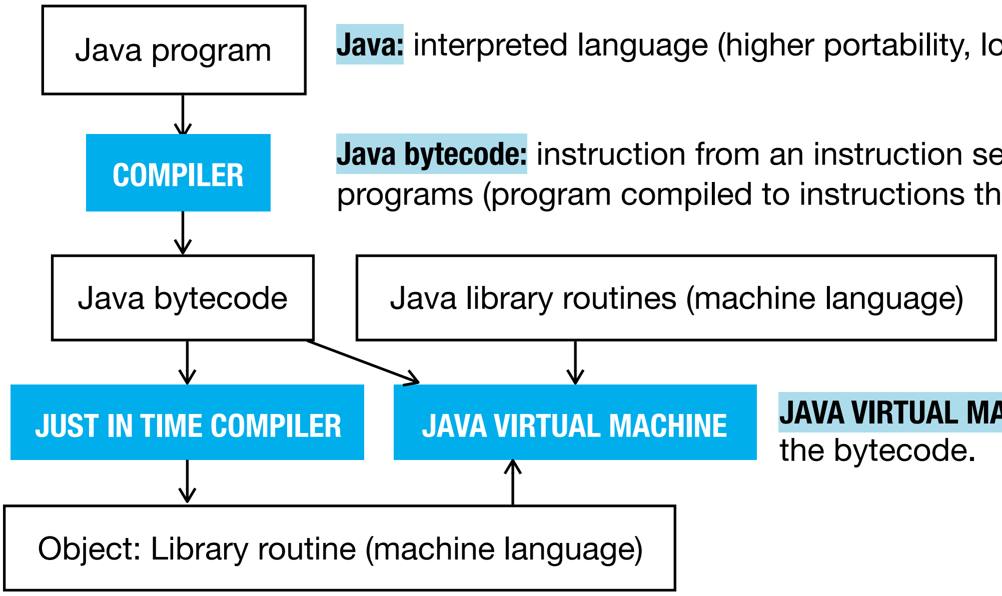
> ***Lazy linkage:*** each routine is linked only after it is called.
> The first time the library routine is called the program branches to the dynamic linker after putting a number in a register to identify the desired library routine.
> After the linker/loader finds the wanted routine, it remaps it and changes the address in the branch to point at it.
> The next calls to the library routine branches indirectly.

Additional space needed but whole libraries don't need to be copied or linked.
Pays overhead the first time a routine is called.

Physical memory (DRAM): *physical address.*
Virtual memory: *more addresses,* ***MMU (Memory Management Unit)*** *translates virtual addresses into physical addresses.*

Virtual address
MMU
Physical address

Starting a Java Program


Java program
COMPILER
Java bytecode
Java library routines (machine language)
JUST IN TIME COMPILER
JAVA VIRTUAL MACHINE
Object: Library routine (machine language)

Java: interpreted language (higher portability, lower performance).

Java bytecode: instruction from an instruction set designed to interpret Java programs (program compiled to instructions that re easy to interpret).

JAVA VIRTUAL MACHINE: program that interprets the bytecode.

JUST IN TIME COMPILER: compiler that translates while the program is running compiling the hot methods into the native instruction set and then saves it for the next time the program is run (increases performance).

Swap Example (leaf)

```
void swap(long long int v[], size_t k)
{
      long long int temp; (8 byte = doubleword, 8 bytes apart from each other)
      temp = v[k];
      v[k] = v[k+1];
      v[k+1] = temp;
}
```

Parameters:

v (address) = x10

k (value) = x11

```
swap:
     slli  x6, x11, 3       // reg x6 = k * 8 (gets address of v[k] multiplying by 8
     add   x6, x10, x6      // reg x6 = v + (k * 8)
     ld    x5, 0(x6)        // reg x5 (temp) = v[k]
     ld    x7, 8(x6)        // reg x7 = v[k + 1]
     sd    x7, 0(x6)        // v[k] = reg x7
     sd    x5, 8(x6)        // v[k+1] = reg x5 (temp)
     jalr  x0, 0(x1)        // return to calling routine
```

Sort Example (Non-leaf)

Non-leaf: calls swap.

```
void sort (long long int v[], size_t int n)
{
     size_t i, j;
     for (i = 0; i < n; i += 1) {
          for (j = i – 1; j >= 0 && v[j] > v[j + 1]; j -= 1) {
               swap(v, j);
          }
     }
}
```

Parameters:

v (address) = x10

n (number of elements of the array) = x11

i = x19

j = x20

```
Sort:
     addi  sp, sp, -40      // make room on stack for 5 registers
     sd    x1,  32(sp)      // save x1 (ra) on stack
     sd    x22, 24(sp)      // save x22 (n) on stack
     sd    x21, 16(sp)      // save x21 (byte address) on stack
     sd    x20, 8(sp)       // save x20 (j) on stack
     sd    x19, 0(sp)       // save x19 (i) on stack

     mv    x21, x10         // copy parameter x10 (v) in x21
     mv    x22, x11         // copy parameter x11 (n) in x22
     li    x19, 0           // initialize x19 (i) to 0 (i=0)

for1tst:
     bge   x19, x22, exit1 // branch if x19 >= x11 (i >= n), as long as i < n
                              loop
     addi  x20, x19, -1     // initialize x20 (j) decreasing x19 by -1 (j = i-1)

for2tst:
     slli  x5,  x20, 3      // byte address (x5) shifting left x20 (j) by 3
     add   x5,  x21, x5     // v + (j*8)
     ld    x6,  0(x5)       // load in x6 v[j]
     ld    x7,  8(x5)       // load in x7 v[j+1], offset 8 for next element
     ble   x6,  x7, exit2   // branch if x6 (v[j]) <= x7 (v[j+1])
     mv    x10, x21         // copy parameter x21 in x10 for swap (v)
     mv    x11, x20         // copy parameter x20 in x11 for swap (j)
     jal   x1,  swap        // call swap
     addi  x20, x20, -1     // decrement x20 (j) by 1
     j for2tst              // go to for2tst
```

```
exit2:
      addi x19, x19, 1        // i += 1
      j for1tst               // go to for1tst

exit1:
      ld   x19, 0(sp)         // restore x19 (i) from stack
      ld   x20, 8(sp)         // restore x20 (j) from stack
      ld   x21, 16(sp)        // restore x21 from stack
      ld   x22, 24(sp)        // restore x22 from stack
      ld   x1,  32(sp)        // restore x1 (ra) from stack
      addi sp, sp, 40         // restore stack pointer
      jalr x0,  0(x1)         // return to calling routine
```

Effect of Compiler Optimization

__**Optimization:**__ the compiler makes adjustments to the code to increase performance.

1. *Function in-lining (text substitution instead of procedure calling)*
2. *Basic block*
3. *Optimization among different basic blocks*

Attention! *Instruction count and CPI are not good performance indicators in isolation!*

Arrays versus Pointers

Clear1 uses indices:

```
clear1(long long int array[], size_t int size)
{
      size_t i;
      for (i = 0; i < size; i += 1)
            array[i] = 0;
}

      li x5, 0                // initialize i = 0
loop1:
      slli x6, x5, 3          // x6 = i * 8 (byte address)
      add  x7, x10, x6        // x7 = address of array[i]
      sd   x0, 0(x7)          // array[i] = 0
      addi x5, x5, 1          // i = i + 1
      blt  x5, x11, loop1     // if (i < size) go to loop1
```

Clear2 uses pointers: the address of the first element of array to the pointer p.
Fewer instruction than clear1, reduced from 5 top 3 inside the loop.

```
clear2(long long int *array, size_t int size)
{
      long long int *p;
      for (p = &array[0]; p < &array[size]; p = p + 1)
            *p = 0;
}

      mv   x5, x10            // p = address of array[0]
      slli x6, x11, 3         // x6 = size * 8 (last element of the array)
      add  x7, x10, x6        // x7 = address of array[size]
loop2:
      sd   x0, 0(x5)          // [p] = 0
      addi x5, x5, 8          // p = p + 8 (increment p to go to the next element)
      bltu x5, x7, loop2      // if (p < array[size]) go to loop2
```

Real Stuff: MIPS Instructions

Reduced Instruction Set Architecture.

Basic set of instruction similar to RISC-V:

- 32 bit instruction;
- 32 general purpose register, register 0 is 0;
- 32 floating point registers;
- Load and Store to access memory.

Conditional branches (different to RISC-V): comparison between two registers, whether the comparison is true a register is set to 1 or 0 (flag), then branch depending on the outcome.

Real Stuff: x86 Instructions

Evolution

- *1978*
  - ➡ 16-bit architecture
  - ➡ 16-bits wide registers (not general purpose)
- *1980*
  - ➡ Floating-point coprocessor
- *1982*
  - ➡ 24-bits address
  - ➡ MMU
- *1985*
  - ➡ 32-bits architecture
  - ➡ 32-bits wide registers
  - ➡ 32-bits address
- *1989*
  - ➡ Pipelined
  - ➡ On chip caches and FPU (Floating Point Unit)
- *1999*
  - ➡ SIMD (Single Instruction Multiple Data)
- *2003*
  - ➡ 64-bits address
  - ➡ 64-bits wide registers
- *2004*
  - ➡ 256-bits wide registers

Golden Handcuffs: increase compatibility adding instructions.

X86 Registers and Data Addressing Mode

Registers:

- 16 registers
- 8 x 32-bits General Purpose Registers (fewer GPR than RISC-V means more memory spills)
- Not all registers can be used for all operations

Arithmetic/logical instruction:

- One operand must act as both source and destination
- One of the operands can be in memory (in RISC-V load and store)

X86 Integer Operations

Data size: can be 8, 16, 32 bits.
Different size of instruction, override with an 8-bit prefix in the instruction
Conditional branches: check the result of a previous operation (flag).

X86 Instruction Encoding

Instruction: may vary from 1 to 15 bytes.
Byte-aligned PC (can have 1 byte operations).

Real Stuff: The Rest of the RISC-V Instruction Set

M: integer multiply, divide, reminder
A: atomic operations
F: single precision floating point
D: double precision floating point
C: compressed instructions (16-bit encoding)

Chapter 3

Arithmetic for Computers

Addition and Subtraction

Overflow: occurs when the result from an operation cannot be represented with the available hardware *(i.e. when a number needs 65 bits to be represented, the bit sign is set with the value of the result instead of the proper sign).*

Cannot occur when adding a positive and a negative number, when subtracting two operands with the same sign (cause we negate the second operand and then we add —> i.e. adding two operands of different signs).

Occurs when:

- subtracting a negative number from a positive number and get a negative result;
- subtracting a positive number from a negative number and get a positive result.

Multiplication

Multiplicand: first operand.

Multiplier: second operand.

Product: result.

➡ Must cope with **overflow** *(n-bit multiplicand, m-bit multiplier = n+m bits product).*

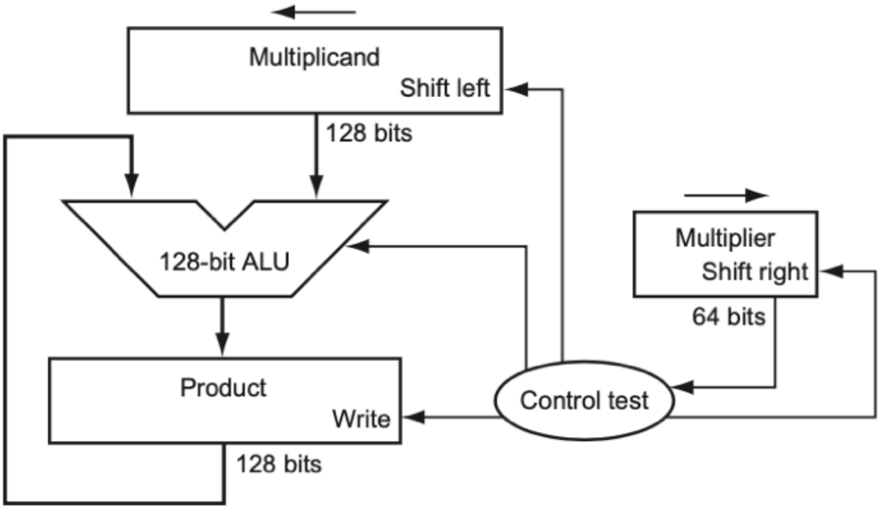
Steps:

1. ***Multiplier* = *1:*** place a copy of the multiplicand;
2. ***Multiplier* = *0:*** place 0.

Sequential Version of the Multiplication Algorithm and Hardware

- **Multiplicand:** 128-bit register initialized with the 64-bit multiplicand in the right half (0 in the left half). Shifted left 1 bit each step to align the multiplicand with the sum accumulated in the product register.
- **Multiplier:** 64-bit register.
  Shifted right 1 bit each step.
  The least significant bit (LSB) determines whether the multiplicand is added.
- **Product:** 128-bit register (initialized to 0).

Slow: takes 64 clock cycles.

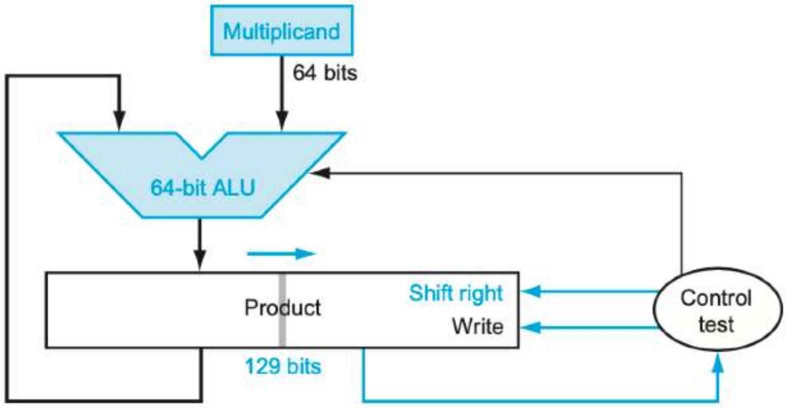


Optimized version

3 improvements:

1. Smaller multiplicand
2. Smaller ALU
3. Multiplier is contained in product and then shifted left

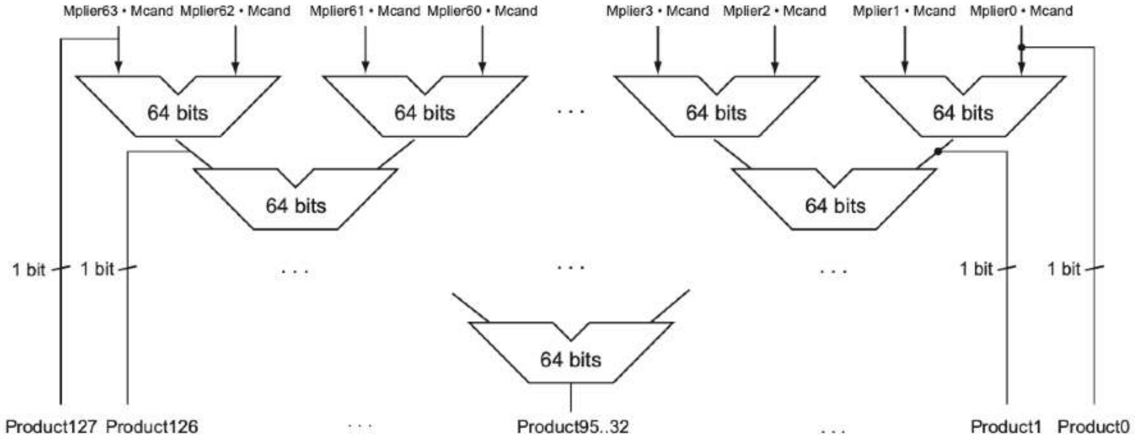
Takes 64 clock cycles but less hardware.



Faster Multiplication

Using a 64-bit adder for each bit of the multiplier: one input is the multiplicand ANDed with a multiplier and the other is the output of the prior adder.

Faster: $\log_2(64) = 6$ BUT more hardware (more dynamic energy).



Multiply in RISC-V

- **Multiply:** integer 64-bit product.

  `mul`

- **Multiply high:** upper 64 bits of the 128-bit product (both signed).

  `mulh`

- **Multiply high unsigned:** upper 64 bits of the 128-bit product (both unsigned).

  `mulhu`

- **Multiply high signed-unsigned:** upper 64 bits of the 128-bit product (one signed, other unsigned).

  `mulhsu`

Division

Dividend: first operand.

Divisor: second operand.

Quotient: first result.

Reminder: second result.

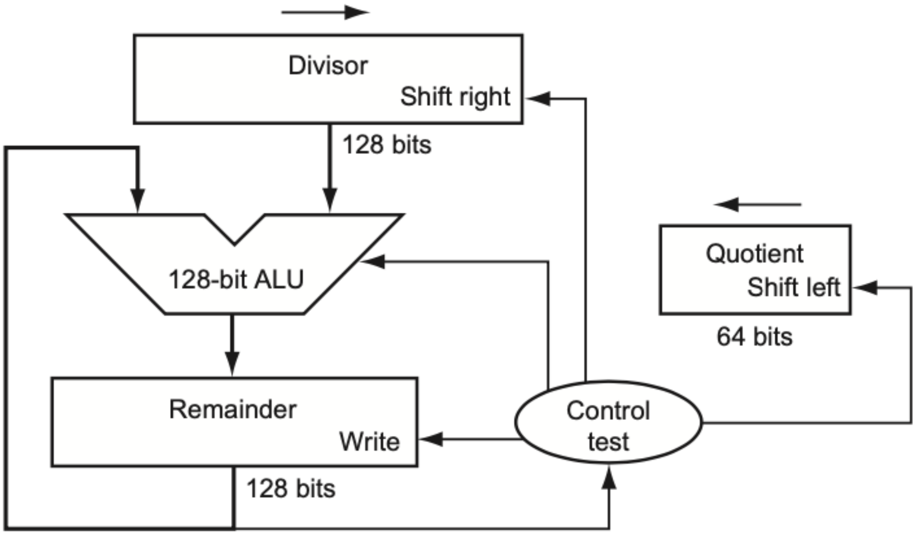
➡ Ignores overflow

A Division Algorithm and Hardware

- **Divisor:** 128-bit register initialized with the 64-bit divisor in the left half (0 in the right half). Shifted right 1 bit each step to align the divisor with the dividend.
- **Reminder:** 128-bit register initialized with the dividend.
- **Quotient:** 64-bit register initialized to 0. Shifted right 1 bit each step.

The divisor is subtracted from the reminder:

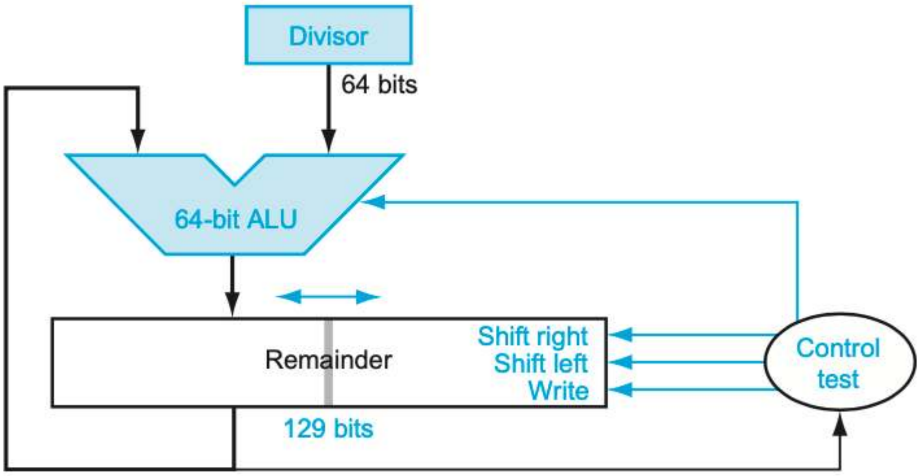
- *positive result:* place 1 in quotient;
- *negative result:* place 0 in the quotient and restore the value (add divisor and reminder) and place it in the reminder register.



Optimized version

2 improvements:

1. Smaller divisor
2. Smaller ALU
3. Reminder register initialize with the dividend, combined with the quotient.



Division cannot be optimized (need to know the sign of the difference before we can perform the next step of the algorithm).

Divide in RISC-V

- **Divide**

  `div`

- **Divide unsigned**

  `divu`

- **Reminder**

  `rem`

- **Reminder unsigned**

  `remu`

Floating Point

Floating point: computer arithmetic that represents numbers in which the binary point is not fixed. Numbers are represented as a single nonzero digit to the left of the binary point.
$1.xxxxx_{two}$ x 2^{yyy}

Floating-Point Representation

Fraction (*mantissa*): value placed in the fraction field.
Increasing the size of the fraction enhances the precision of the fraction.
Exponent: value placed in the exponent field.
Increasing the size of the exponent increases the range of numbers that can be represented.
$(-1)^{s}$ x F x 2^{E}

Single precision: floating-point value represented in a 32-bit word.

1 bit: *sign* **8 bits:** *exponent* **23 bits:** *mantissa*

Overflow (floating-point): occurs when the exponent is too large to be represented.
Underflow (floating-point): occurs when the negative exponent is too large to be represented.

Double precision: floating-point value represented in a 64-bit doubleword.

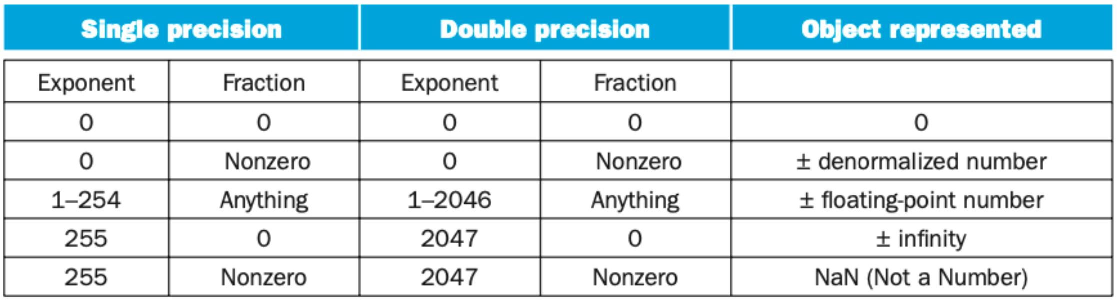
1 bit: *sign* **11 bits:** *exponent* **52 bits:** *mantissa*

Exceptions and Interrupts

Exception: unscheduled event that disrupts program execution, used to detect overflow.
Interrupt: exception that comes from outside the processor.
RISC-V computers do not raise an exception on overflow or underflow.

IEEE 754 Floating-Point Standard

The leading 1 bit is implicit (**single precision:** 1 + 23 bit fraction = *significand*, **double precision:** 1 + 52 bit fraction = *significand*).
$(-1)^{s}$ x (1 + F) x $2^{E\ \text{(exponent - bias)}}$



Single precision		Double precision		Object represented
Exponent	Fraction	Exponent	Fraction	
0	0	0	0	0
0	Nonzero	0	Nonzero	± denormalized number
1–254	Anything	1–2046	Anything	± floating-point number
255	0	2047	0	± infinity
255	Nonzero	2047	Nonzero	NaN (Not a Number)

Biased Notation: bias is the number being added to the exponent.
Single precision: 127
Exponent max = 127
Exponent min = -126
Double precision: 1023
Exponent max = 1023
Exponent min = -1022

Floating-Point Addition

Steps:

1. *Align binary points*
   i.e. shift numbers with smaller exponents until it matches the larger one.
2. *Add significands*
3. *Normalize the result* (shifting right incrementing the exponent or left decrementing the exponent)
4. *Round and renormalize if necessary*

Floating-point adder:
Floating-point additions can’t be done in one clock cycle cause the hardware is more complex.
Can be *pipelined*.

1 clock cycle	op 4	exp
2 clock cycle	op 3	sum
3 clock cycle	op 2	norm
4 clock cycle	op 1	round

Floating-Point Multiplication

Steps:

1. *Calculate the exponent adding the exponent together*
   Attention! *To get the correct biased sum we must subtract the bias from the sum.*
2. *Multiplication of the significand*
3. *Normalization and check for overflow or underflow*
4. *Rounding*
5. *Check the sign*
   if the signs are the same = positive, if they differ = negative

Floating-Point Instructions in RISC-V

Single precision: `.s` 32-bit word.
Double precision: `.d` 64-bit doubleword.

- **Addition:**
  `fadd.s`
  `fadd.d`
- **Subtraction:**
  `fsub.s`
  `fsub.d`
- **Multiplication:**
  `fmul.s`
  `fmul.d`
- **Division:**
  `fdiv.s`
  `fdiv.d`
- **Square root:**
  `fsqrt.s`
  `fsqrt.d`
- **Equals:**
  `feq.s`
  `feq.d`
- **Less than:**
  `flt.s`
  `flt.d`
- **Less than or equals:**
  `fle.s`
  `fle.d`

Comparison instructions: `feq`, `flt`, `fle` set an integer register to 0 if the comparison is false and 1 if it’s true then it branches using the integer branch instructions.
Comparison with 2 instructions like MIPS.
Attention! *Avoid comparison in floating-point cause normalization and rounding can modify the number.*

Registers: 32 floating-point registers (`f0 - f31`).
Can hold a single-precision floating-point number or a double-precision floating-point number.
Single precision register is the lower half of a double precision register.

Data transfer instructions:

- **Load word:**
  `flw  // loads single precision`
- **Load doubleword:**
  `fld  // loads double precision`
- **Store word**:
  `fsw  // stores single precision`
- **Store doubleword:**
  `fsd  // stores double precision`

Accurate Arithmetic

Floating-point numbers are approximations for a number that can't be represented so we need to get the floating-point representation close to the actual number.
Rounding modes: IEEE 754 keeps two extra bits on the right during intermediate calculations to improve rounding accuracy.
Guard: first bit.
Round: second bit.
Units in the last place (ulp): the number of bits in error in the least significant bits of the significand between the actual number and the number that can be represented.

Parallelism and Computer Arithmetic: Subword Parallelism

SIMD: *Single Instruction Multiple Data*, permits operative parallelism.
ALU with SIMD can perform the same operation on different data partitioning the adder (graphics and audio application can take advance of performing simultaneous operations on shorter vectors).

Example

128-bit adder can perform:

- 16 8-bit operands
- 8 16-bit operands
- 4 32-bit operands
- 2 64-bit operands

Real Stuff: Streaming SIMD Extensions and Advanced Vector Extensions in x86

MMX: *MultiMedia eXtension*, included instructions that operate on short vectors of integers.
SSE: *Streaming SIMD Extension*, provided instructions that operate on short vectors of single-precision floating-point numbers (SSE2 includes double precision floating-point).

16 registers called XMM: xmm means one operand is a 128-bit SSE2 register (registers code name).

- **SS:** ***scalar single*** precision floating point ***(one 32-bit operand)***;
- **SD:** ***scalar double*** precision floating point ***(one 64-bit operand)***;
- **PS:** ***packed single*** precision floating point ***(four 32-bit operands)***;
- **PD:** ***packed double*** precision floating point ***(two 64-bit operands)***.

YMM: width of the register is doubled with AVX (advanced Vector eXtension).

- Eight 32-bit floating point operations;
- Four 64-bit floating point operations.

DGEMM: *Double precision GEneral Matrix Multiply.*
The AVX version of DGEMM is faster than the unoptimized one cause it performs more operations using subword parallelism.

Fallacies and Pitfalls

Fallacy: shifting right is like dividing.
Only for unsigned integers.
Pitfall: floating-point addition is not associative.
Floating point numbers have limited precision and result in approximations of real results.
On parallel computer floating point sums can be calculated in different orders getting different answers despite running identical code cause of the number of processors running.

		(x+y)+z	x+(y+z)
x	-1.50E + 38		-1.50E + 38
y	1.50E + 38	0.00E + 00	
z	1.0	1.0	1.50E + 38
		1.00E + 00	**0.00E + 00**

—> adding 1 the result remains the same

Chapter 4
The Processor

Basic RISC-V Implementation

Three classes of instruction:

- ***Memory-reference;***
- ***Arithmetic-logical;***
- ***Branches.***

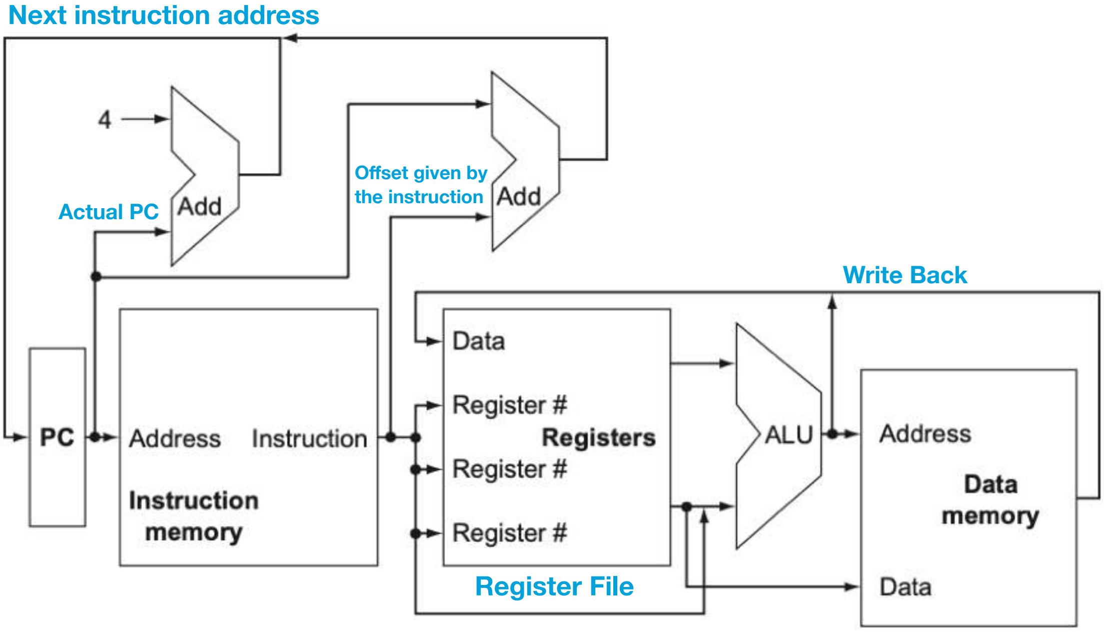
Every class use the ALU.

Three identical steps for every instruction:

1. *Send the program counter to the memory to fetch the instruction;*
2. *Read one or two registers using fields of the instruction to select them;*
3. *Use of the ALU.*
   Every class uses the ALU to perform ***different operations***:
   - ***Memory-reference:*** address calculation;
   - ***Arithmetic-logical:*** perform operations;
   - ***Branches:*** equality test.

Fourth step differ for every instruction class:

- ***Memory-reference:*** access memory to read (load) or write (store) data;
- ***Arithmetic-logical:*** write data into a register;
- ***Branch:*** change instruction address or increment by 4 the PC to go to the next instruction.



Explanation:

All instructions start by using the PC to supply the instruction address to the instruction memory. After the instruction is fetched the register operands used by an instruction (specified by the field of that instruction) are fetched.

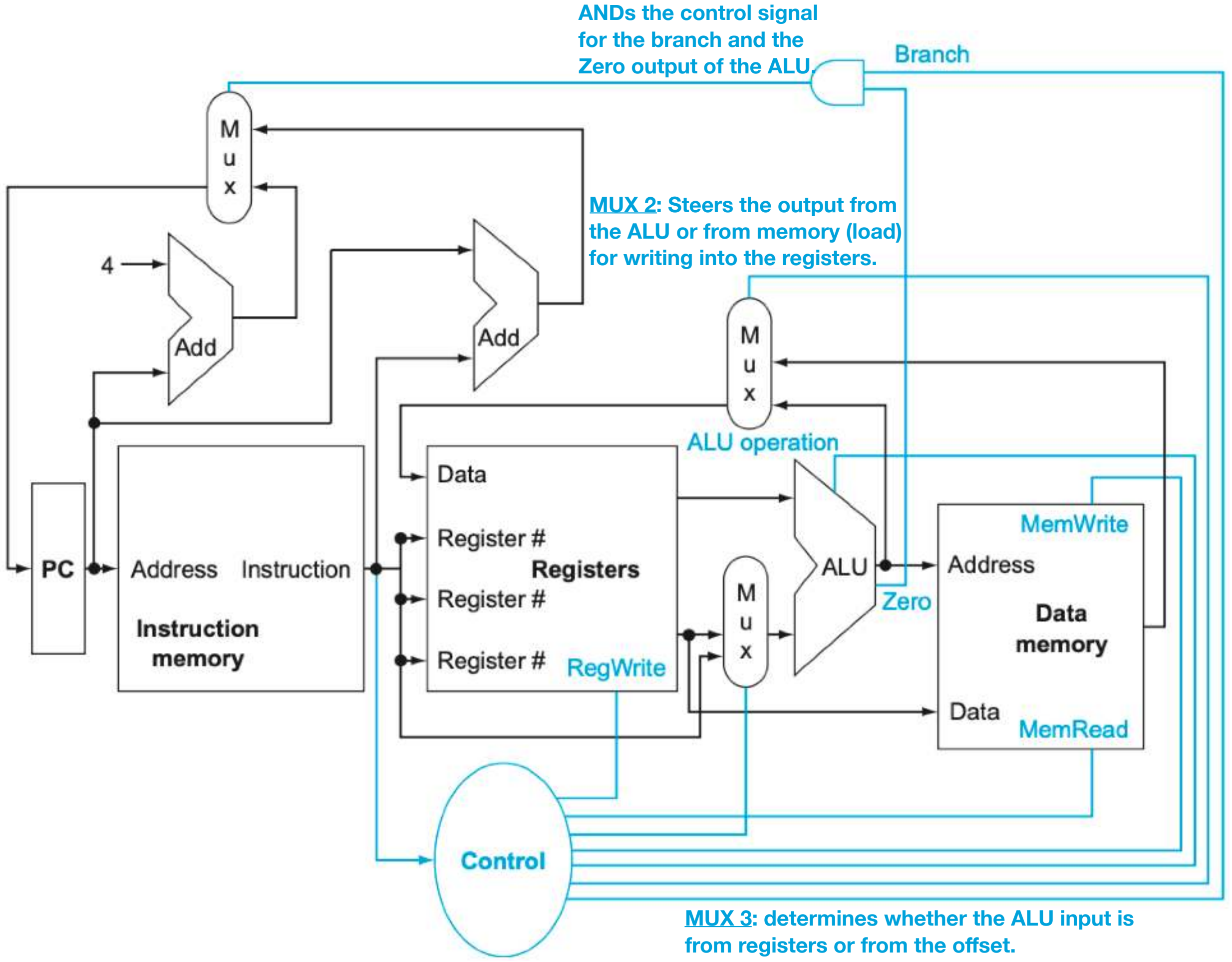
After being fetched they can be operated to compute a memory address, to compute an arithmetic result, or an equity check.

If the instruction is an arithmetic-logical instruction the result must be written to a register.

If the instruction is a load or store the result is used as an address to either store a value from the registers or load a value from memory.

If the instruction is a branch the result determines the next instruction address.

Control Unit: (instruction is the input) determines how to set the control lines for the functional unit and two of the multiplexers.



ANDs the control signal for the branch and the Zero output of the ALU.

MUX 2: Steers the output from the ALU or from memory (load) for writing into the registers.

MUX 3: determines whether the ALU input is from registers or from the offset.

Logic Design Conventions

Combinational element: outputs depend on the current inputs (such as ALU).
State element: elements containing a state i.e. internal storage (such as registers or memory).
Also called ***sequential*** cause their outputs depend on both their inputs and the contents of the internal state.
At least two inputs:

- data value;
- **clock** (determines when the data is written).

Clocking Methodology

Clocking methodology: approach used to determine when data are valid and stable relative to the clock.
It defines when signals can be read and when they can be written.
Edge-triggered clocking: clocking scheme in which all state changes occur on a clock edge.
Any value stored in a sequential logic element are updated only on a clock edge (quick transition from low to high and vice versa).
Allows a state element to be read and written in the same clock cycle without creating a race that could lead to indeterminate data values.

Building a Datapath

Datapath element: unit used to operate on or hold data within a processor.

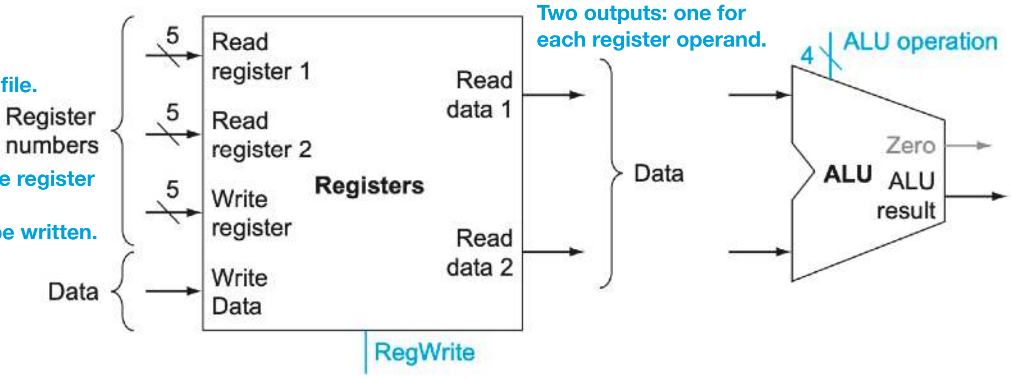
Access instructions:

Program counter: register containing the address of the instruction in the program being executed.
Must be incremented by 4 bytes to point at the next instruction (performed with an ALU).
Register file: state element consisting of a set of registers that can be read and written by supplying a register number to be accessed.

➡ **Arithmetic-logical instructions** (`add, sub, and, or`)
Register number inputs are 5 bits wide ($2^5 = 32$).

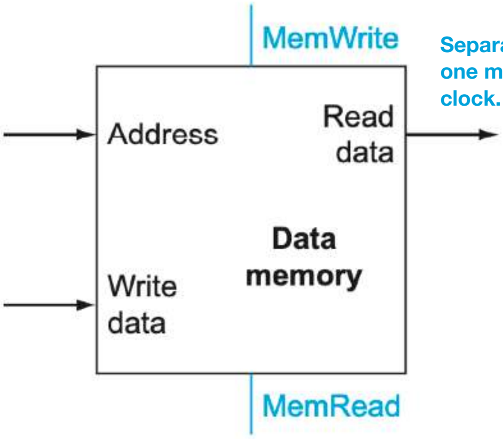
READ
Two read ports: read the 2 operands from the register file.

WRITE
One write port: specifies the register number to be written.
Data: supplies the data to be written.

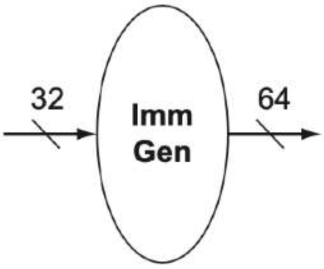


Two outputs: one for each register operand.

➡ **Memory-reference instructions** (`ld, sd`)

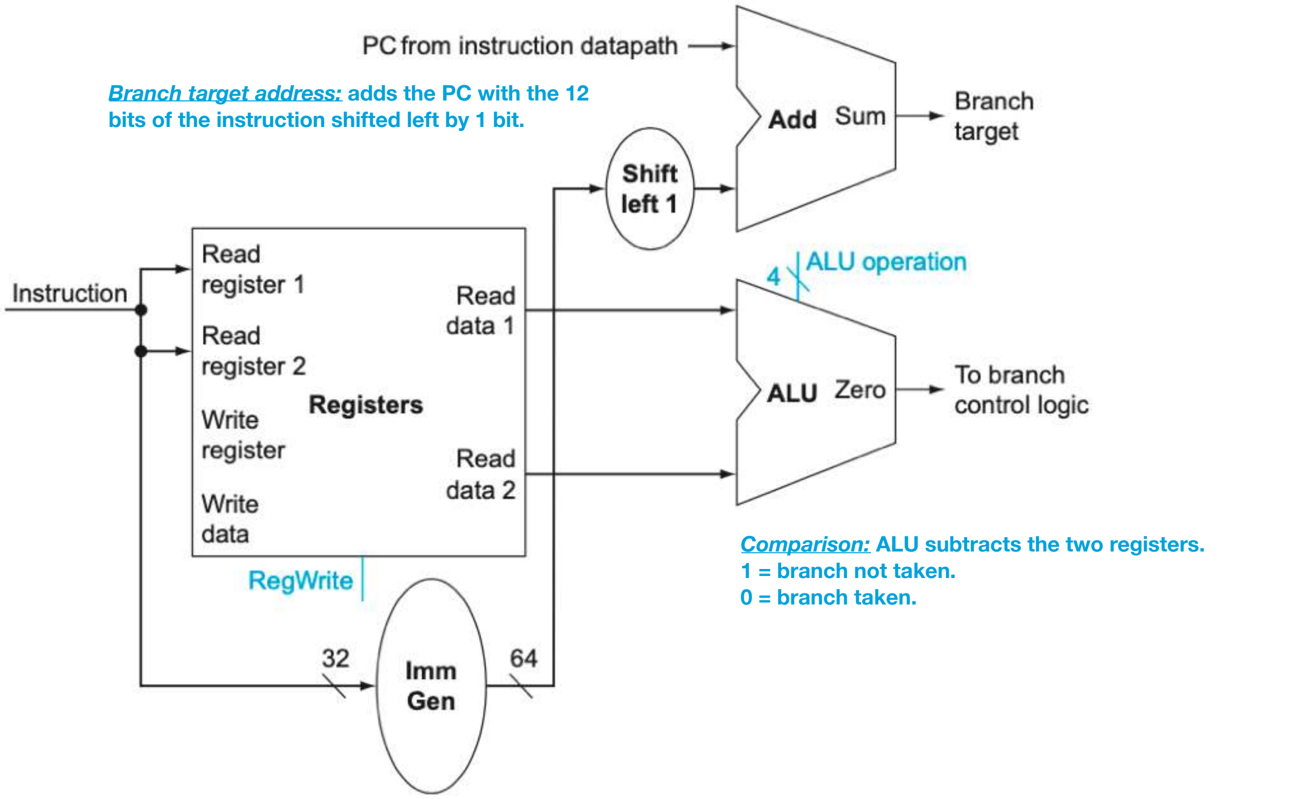


Separate read and write controls (only one may be asserted on any given clock.



Immediate generation unit: input is a 32-bit instruction, selects the 12-bit field and extends it into a 64-bit result (*sign-extend*).

➡ **Branch instructions**
Branch target address: address specified in a branch which becomes the new program counter (given by the sum of the offset field and the address of the branch).
Branch taken: the branch condition is satisfied.
Branch not taken: the branch condition is false and the PC becomes the address of the next instruction that sequentially follows the branch.

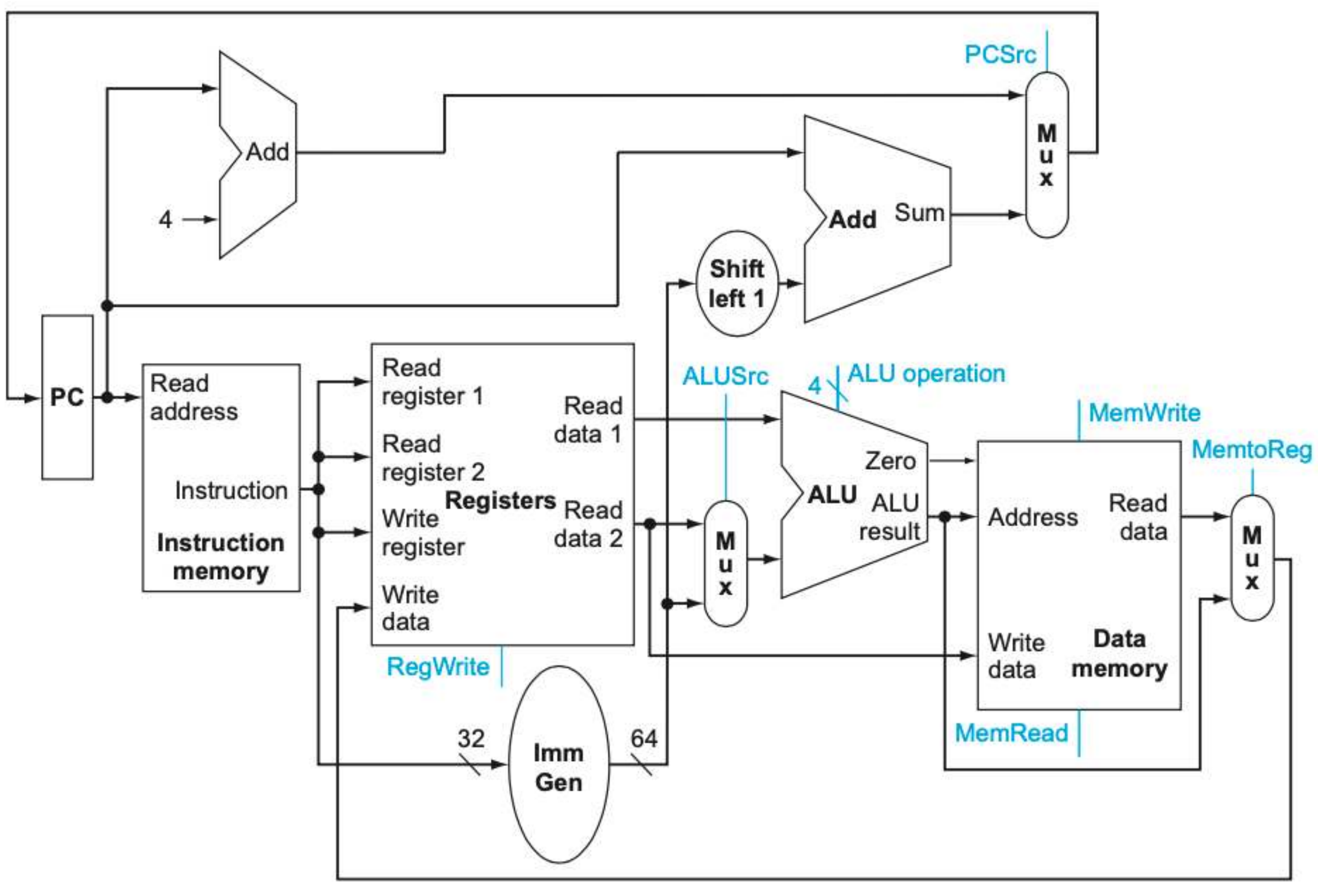


Branch target address: adds the PC with the 12 bits of the instruction shifted left by 1 bit.

Comparison: ALU subtracts the two registers.
1 = branch not taken.
0 = branch taken.

Creating a Single Datapath

Execute all instruction in a single clock cycle ⟶ *separate memory for instruction and data*



A Simple Implementation Scheme

The ALU Control

➡ **<u>Arithmetic-logical instructions:</u>** use the ALU to perform 4 operations.

ALU CONTROL	FUNCTION
0000	**AND**
0001	**OR**
0010	**Add**
0110	**Sub**

<u>ALU control:</u> inputs are the funct3 and funct7 fields of the instruction and a 2-bit control field called ALUOp.

<u>ALUOp:</u> indicates whether the operation to be performed is:

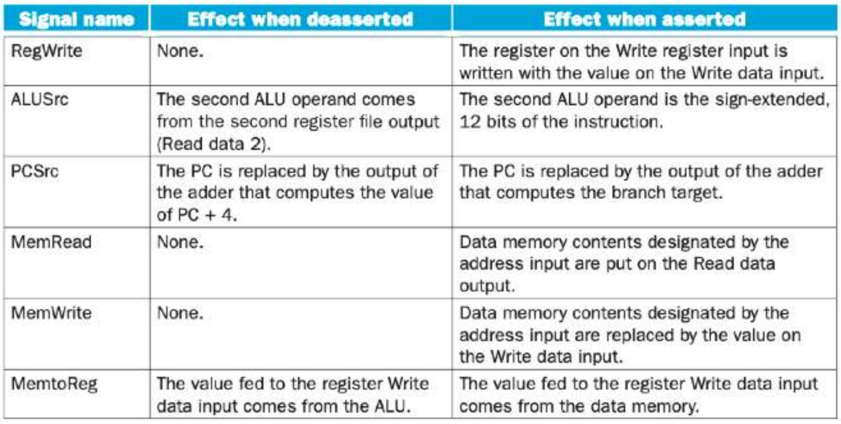
- add (00) for ld/sd;
- sub (01) for beq;
- determined by the funct fields (10).

Instruction opcode	ALUOp	Operation	Funct7 field	Funct3 field	Desired ALU action	ALU control input
ld	00	load doubleword	XXXXXXX	XXX	add	0010
sd	00	store doubleword	XXXXXXX	XXX	add	0010
beq	01	branch if equal	XXXXXXX	XXX	subtract	0110
R-type	10	add	0000000	000	add	0010
R-type	10	sub	0100000	000	subtract	0110
R-type	10	and	0000000	111	AND	0000
R-type	10	or	0000000	110	OR	0001

Designing the Main Control Unit

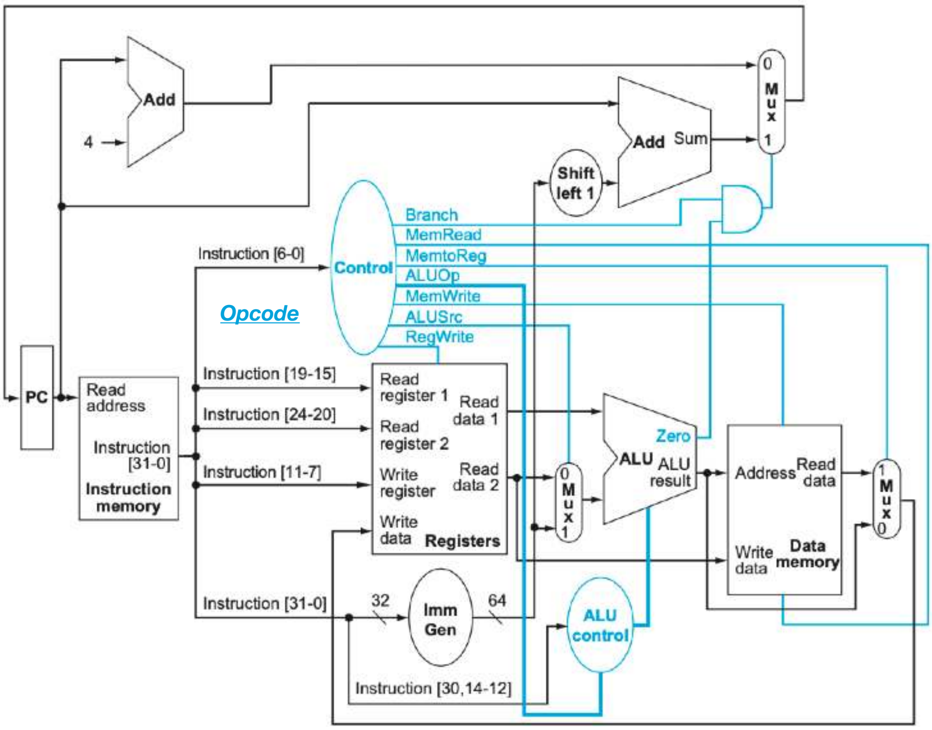
1-bit control to a two way multiplexer:

- ***Asserted:*** selects input corresponding to 1;
- ***Deasserted:*** selects input corresponding to 0.



Signal name	Effect when deasserted	Effect when asserted
RegWrite	None.	The register on the Write register input is written with the value on the Write data input.
ALUSrc	The second ALU operand comes from the second register file output (Read data 2).	The second ALU operand is the sign-extended, 12 bits of the instruction.
PCSrc	The PC is replaced by the output of the adder that computes the value of PC + 4.	The PC is replaced by the output of the adder that computes the branch target.
MemRead	None.	Data memory contents designated by the address input are put on the Read data output.
MemWrite	None.	Data memory contents designated by the address input are replaced by the value on the Write data input.
MemtoReg	The value fed to the register Write data input comes from the ALU.	The value fed to the register Write data input comes from the data memory.

Operation of the Datapath



Operation of the datapath for an Arithmetic-logical instruction:

```
add  x1, x2, x3
```

Steps:

1. Instruction is fetched, PC is incremented.
2. Two registers (x2, x3) are read from the register file.
3. ALU operates on the data read from the register file, portions of the opcode are used to generate the ALU function.
4. Result written in the destination register (x1).

Operation of the datapath for a Memory-reference instruction:

```
ld   x1, offset(x2)
```

Steps:

1. Instruction is fetched, PC is incremented.
2. A register (x2) value is read from the register file *(not two cause second operand is immediate)*.
3. ALU computes the sum of the value read from the register file and the sign-extended 12 bits of the instruction (offset).
4. Sum from the ALU is used as the address for the data memory.
5. Data from the memory unit Is written into the register file (x1).

Attention! *Store has no write back (no step 5).*

Operation of the datapath for a Branch instruction:

```
beq  x1, x2, offset
```

Steps:

1. Instruction is fetched, PC is incremented.
2. Two registers (`x1`, `x2`) are read from the register file.
3. ALU subtracts one data value from the other.
   The value of PC is added to the sign-extended 12 bits of the instruction (`offset`) left shifted by one; the result is the branch target address.
4. Zero status information from the ALU is used to decide which adder result to store in the PC.

Why a Single-Cycle Implementation is not Used Today

Single-Cycle: too inefficient cause the *clock cycle* must have the *same length* for every instruction.
The longest possible path in the processor determines the clock cycle.

> ***Load instruction:*** uses five functional units (instruction memory, register file, ALU, data memory, register file).

An Overview of Pipelining

Pipelining: implementation technique in which multiple instructions are overlapped.
Pipelined approach ***improves throughput*** (*single instruction latency = remains the same*).
Faster cause everything works in *parallel*.

- If all stages take about the same amount of time and there is enough work to do (*full pipeline*), then the ***speed-up*** due to pipelining is ***equal to the stages in the pipeline***.
- **Beginning/End pipeline is not full:** start-up and wind-down affects performance when the number of tasks is not large compared to the number of stages in the pipeline.

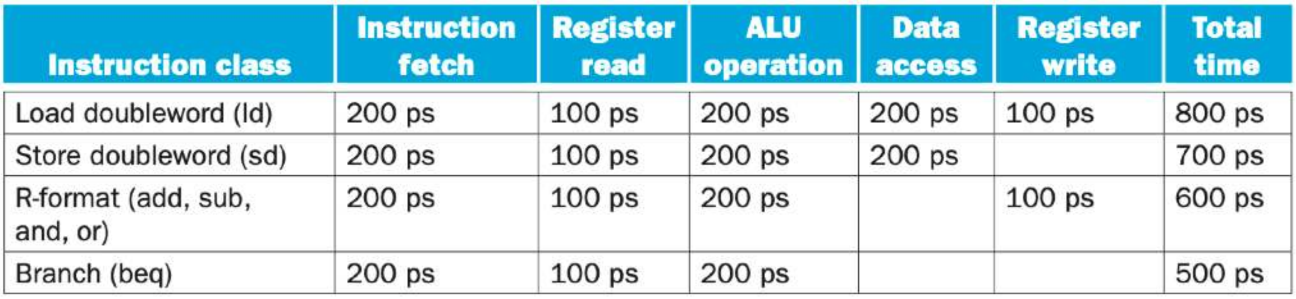
Pipeline stages:

1. IF: Instruction Fetch from memory.
2. ID: Instruction Decoding (read register and decode instruction).
3. EX: Execute the operation or calculate an address.
4. MEM: access an operand in data memory (if necessary).
5. WB: Write the result into a register (if necessary).

Worst-case clock cycle: pipelined execution clock cycle must have the worst-case clock cycle even though some stages take less time.

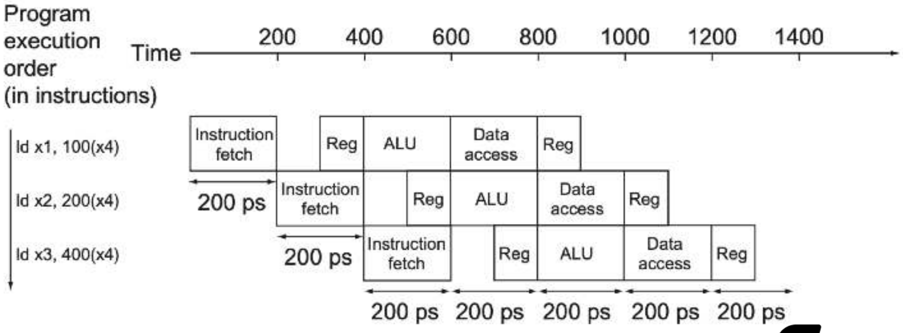
$$\text{Time between instructions}_{\text{pipelined}} = \frac{\text{Time between instructions}_{\text{nonpipelined}}}{\text{Number of pipe stages}}$$

Total time for each instruction:



Instruction class	Instruction fetch	Register read	ALU operation	Data access	Register write	Total time
Load doubleword (ld)	200 ps	100 ps	200 ps	200 ps	100 ps	800 ps
Store doubleword (sd)	200 ps	100 ps	200 ps	200 ps		700 ps
R-format (add, sub, and, or)	200 ps	100 ps	200 ps		100 ps	600 ps
Branch (beq)	200 ps	100 ps	200 ps			500 ps

➡ ***When pipeline is full a new instruction is executed at every clock cycle.***



Designing Instruction sets for Pipelining

- Same length instructions (easier IF and ID).
- Few instruction formats (source and destination register field located in the same place).
- Memory operands only in load and store (EX to calculate address, MEM to access memory). Keeps the design simple (x86 is more complex cause it operates on memory operands).

Pipeline Hazards

Hazards: the next instruction cannot execute in the following clock cycle.

Structural Hazards

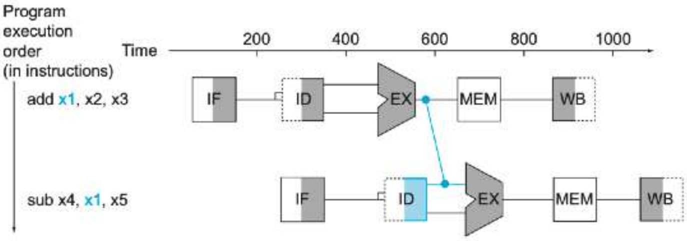
Structural hazard: when a planned instruction cannot execute in the proper clock cycle because the hardware does not support the combination of instructions that are set to execute.
(*Example:* using a single memory for instruction and data).

Data Hazards

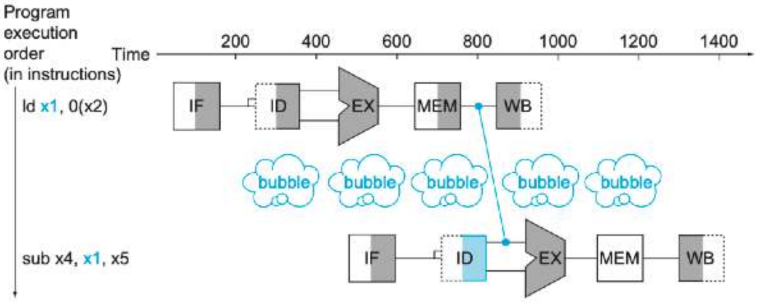
Data hazard: when a planned instruction cannot execute in the proper clock cycle because data that are needed to execute the instruction are not available.
Pipeline must be stalled because on step must wait for another to complete.
Arise from the dependence of one instruction on an earlier one that's still in the pipeline.

Forwarding: method of resolving a data hazard by retrieving the missing data element from internal buffers rather than waiting for it.

Not always useful! Valid only if the destination stage is later in time than the source stage.



Pipeline stall: stall initiated in order to resolve a data hazard.



```
Example:

a = b + e;
c = b + f;
```

```
ld    x1, 0(x31)     // Load b
ld    x2, 8(x31)     // Load e
add   x3, x1, x2     // b + e   Stall
sd    x3, 24(x31)    // Store a
ld    x4, 16(x31)    // Load f
add   x5, x1, x4     // b + f   Stall
sd    x5, 32(x31)    // Store c
```

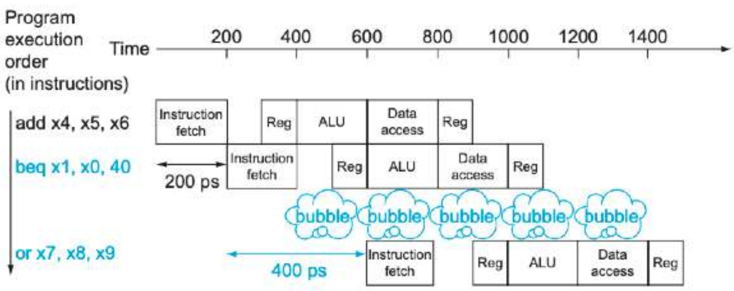
13 clock cycles

```
ld    x1, 0(x31)
ld    x2, 8(x31)
ld    x4, 16(x31)  Load instead of
add   x3, x1, x2   bubble: operation
sd    x3, 24(x31)  without dependency
add   x5, x1, x4
sd    x5, 32(x31)
```

11 clock cycles

Control Hazards

Control hazard: when the proper instruction cannot execute in the proper pipeline clock cycle because the instruction that was fetched in not the one needed.



Conditional branch instruction: IF must wait one clock cycle (stall) because we need the ALU result to know what instruction is needed.

Branch prediction: method of resolving a branch hazard that assumes a given outcome for the conditional branch and proceeds from that assumption rather than ascertain the actual outcome.
Predicts outcome of the branch and stalls if prediction is wrong.

- ***Static branch prediction:*** based on typical branch behavior (ex. loops)
- ***Dynamic branch prediction:*** keeps history for each conditional branch as taken or untaken and then uses the recent past behavior to predict the future, when wrong restart the pipeline.

RISC-V branch prediction

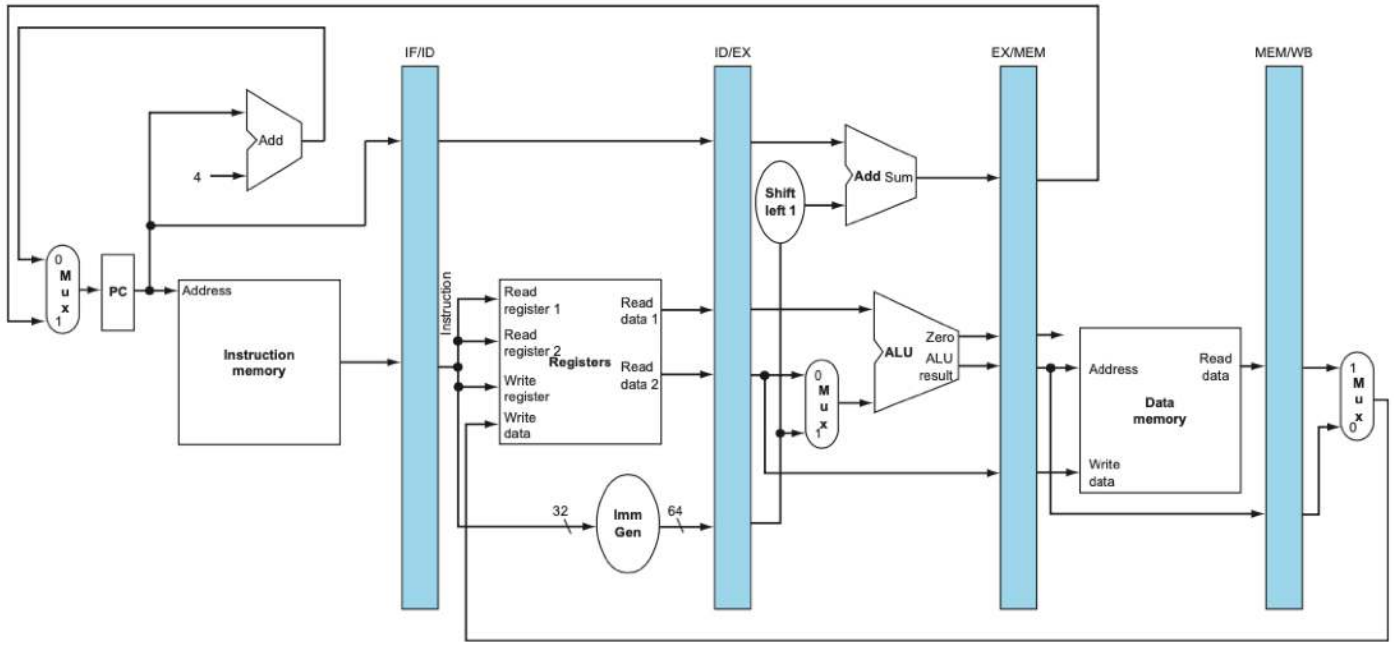
Start executing next instruction (IF and ID).
EX: knows whether branch is:

- ***Taken:*** continues.
- ***Untaken:*** cancels wrong instruction fetched before (not yet executed).

Pipelined Datapath and Control

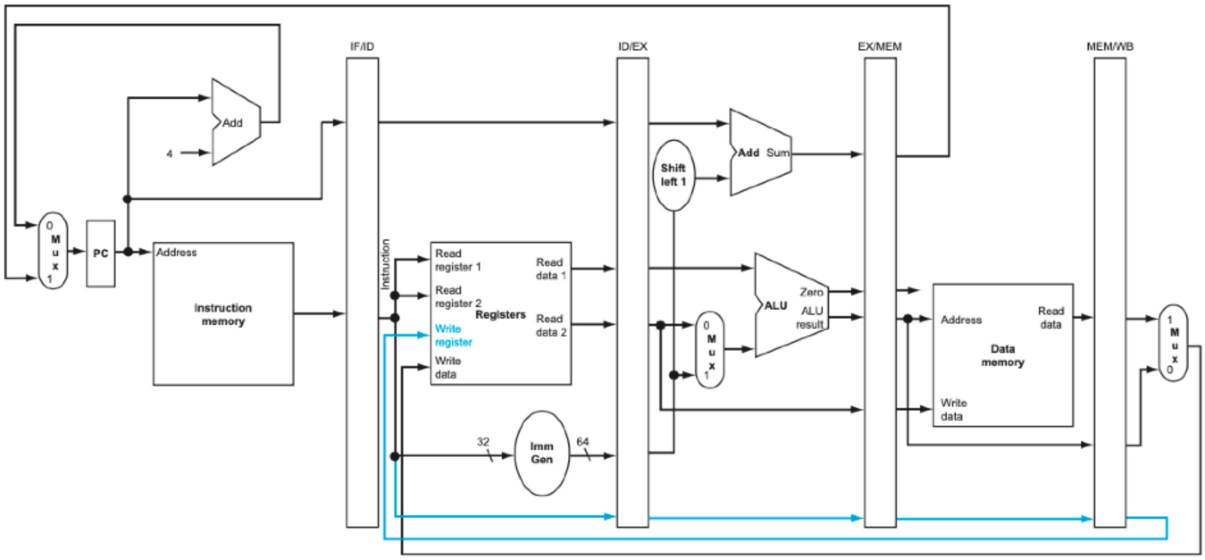
Instruction is divided into five stages: IF, ID, EX, MEM, WB.
Data flows from left to right except:

- WB places the result into the register file (can lead to data hazard);
- Selection of the next PC value, incremented PC or branch address from the MEM stage (can lead t control hazard).



Place register between the sages to save the arguments otherwise the information is lost when the next instruction enters that pipeline stage.

Correct pipelined datapath to handle the load instruction: write register number must be passed from the ID pipe stage until it reaches the MEM/WB pipeline register otherwise it will be lost.

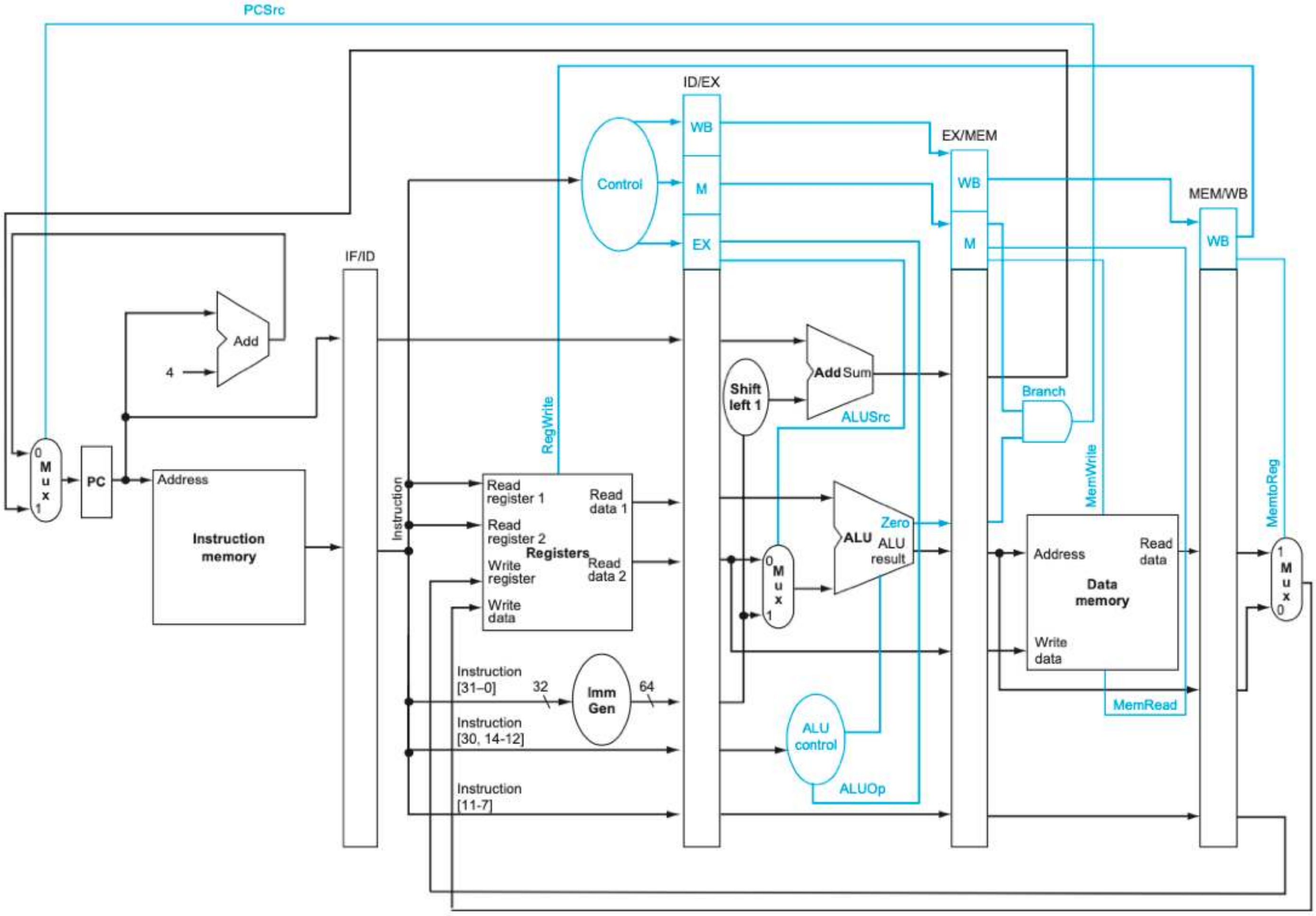


Pipelined Control

Set control lines during each pipeline stage. Five groups:

1. IF: read instruction, increment the PC always asserted.
2. ID
3. EX: signals to be set are ***ALUOp*** and ***ALUSrc***.
4. MEM: control lines are ***Branch***, ***MemRead***, ***MenWrite***.
5. WB: control lines are ***MemtoReg*** (sending the ALU result or the memory vale to the register file) and ***RegWrite*** (writes the chosen value).

Pass control signals extending the pipeline registers to include control information.



Chapter 5

Large and Fast: Exploiting Memory Hierarchy

Introduction

Memory bottleneck: adding cores improves memory bandwidth (bus) but latency remains the same.

Principle of Locality

Temporal locality: if an item is referenced, it will tend to be referenced again soon. (use cache)
Spartial locality: if an item is referenced, items whose addresses are close by will tend to be referenced soon.

Memory Hierarchy

Memory hierarchy: structure that uses multiple levels of memories with different speeds and sizes, as the distance from the processor increases, the size of the memories and the access time both increase.

Faster memories: near the processor, more expensive per bit thus smaller.

Slower memories: far from the processor, take more time to access.

Block (line): minimum unit of information that can be either present or not in a cache.

Hit and miss

- **Hit:** if the data requested by the processor appears in the upper level's blocks.
- **Miss:** if the data requested by the processor is not found in the upper level's blocks. Need to retrieve the requested data from the lower level's block to access it.

Miss rate: fraction of memory accesses not found in a level of the memory hierarchy.
Hit rate: fraction of memory accesses found in the upper level.

Hit time: time required to access a level of the memory hierarchy, including the time to determine whether the access is a hit or a miss.
Miss penalty: time required to fetch a block into a level of the memory hierarchy from the lower level, including the time to deliver the requested block to the processor.

Memory Technologies

DRAM: dynamic random access memory (implements main memory).
SRAM: static random access memory (implements levels closer to the processor i.e. caches).

SRAM Technology

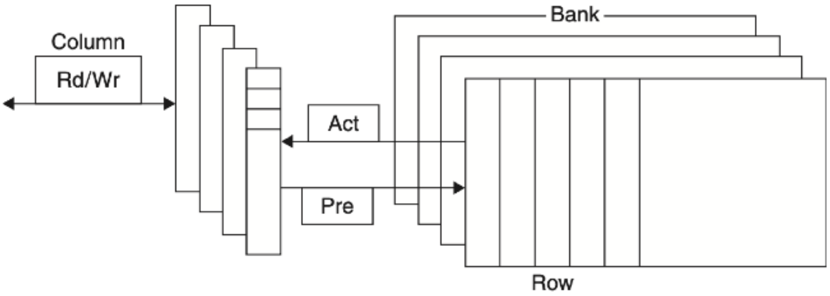
- Use 6 to 8 transistor per bit (prevents information from being disturbed when read).
- Access time close to the clock cycle.
- Value kept indefinitely as long as power is applied.

DRAM Technology

- One transistor per bit (much denser and cheaper).
- Value kept is stored in a cell as a charge in a capacitor (cannot be kept indefinitely). **Refresh:** read the contents and write them again.

Organization

Banks *divided in* **Rows** (share a word line).
Buffer: like a SRAM (acts like a cache on the RAM side), contains the last row read from a bank. Changing the address, random bits can be accessed in the buffer until the next room access.
Each bank can have its own row buffer (faster).



Precharge signal: opens/closes a bank.
Activate signal: sends row address transferring the row to the buffer.
Column: each slot in a row (cache line).

SDRAM: synchronous DRAM.
Burst mode: supply successive words from a row with reduced latency (faster reading sequential data). SDRAM can transfer bits in the burst.
DDR: *double data rate SDRAM*, transfers data on both the rising and falling edge of the clock getting twice as much bandwidth (can transfer 8 data for each bank).

Flash Memory

Flash memory: type of electrically erasable programmable read-only memory.

Flash bits: can wear out after being written many times (limited number of writes for a single location).
Flash controller: changes address of most frequently used location with the ones that have been written less times ⟶ **Wear leveling**

Disk Memory

Magnetic disk: collection of metal platters (covered with magnetic recording material) which rotate on a spindle at 5400 to 15000 revolutions per minute.
Reads and writes information with a movable arm containing a coil called read-write head.

- ***Tracks:*** concentric circles that make up the surface of a magnetic disk.
- ***Sectors:*** segment that make up a track (smallest amount of information that is read/written on a disk).

Stages:

1. ***Seek:*** process of positioning a read/write head over the proper track on a disk.
2. Wait for the desired sector to rotate under the head.
   Rotational delay: time required for the desired sector to rotate under the read/write head, half of the rotation time (sometimes already there other times wait for a complete rotation).
3. ***Transfer time:*** time to transfer a block of bits.

Slower (mechanical device), cheaper (high storage at modest cost) and non volatile.

Example

Given

- 512 B sector (1/2 K)
- 15000 rpm
- 4ms average seek time
- 100 MB/s transfer rate
- 0.2 ms controller overhead

Average read time:

	4 ms
+	1/2 (15000/60) = 2 ms rotational delay
+	512 MB / 100 MB/s = 0,005 transfer time
+	0.2 ms overhead
=	**6.2 ms**

The Basics of Caches

Cache: memory between the processor and main memory.

- **Direct mapped cache:** cache structure in which each memory location is mapped to exactly one location in the cache (only one choice).
  Mapped on the RAM: memory locations shifted by the cache size from each other go in the same cache line.
  Disclaimer! *Not all addresses can go in every cache location but a cache location can have more addresses.*

Index: bit used to select the right cache block.
(Block address) modulo (Number of blocks in the cache)

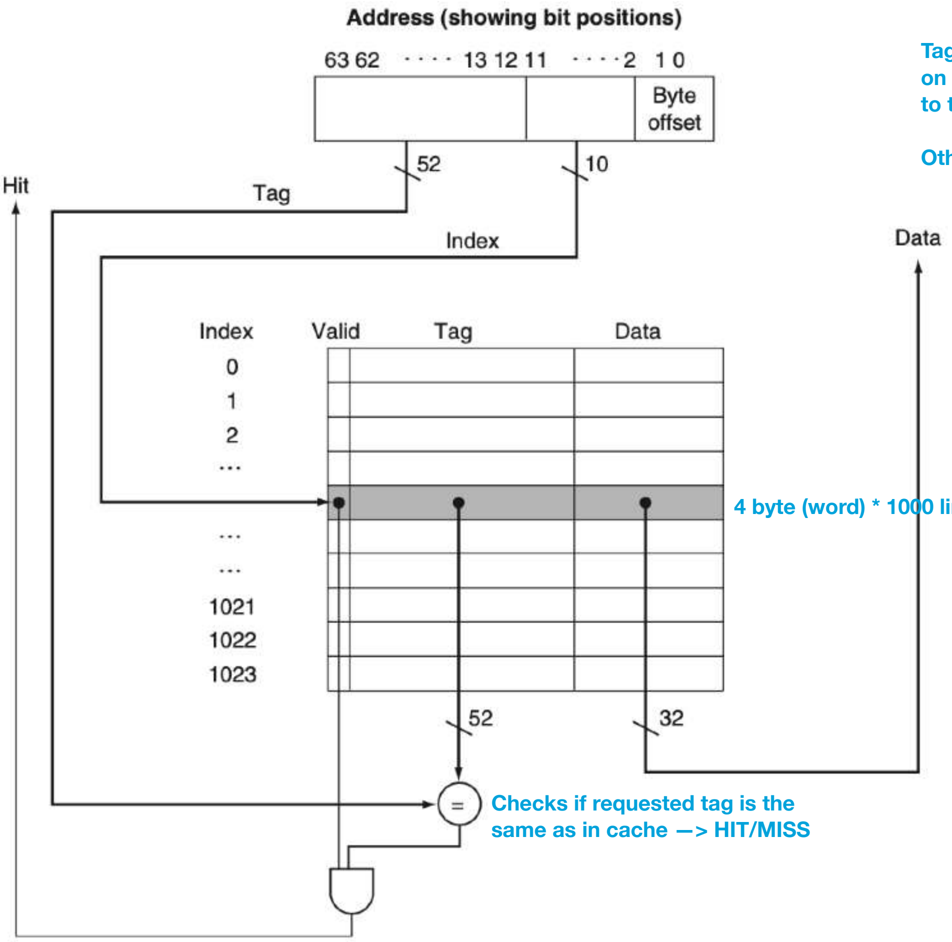
Tag: field in a table used for a memory hierarchy that contains the address information required to identify whether the associated block in the hierarchy corresponded to a requested word.
Contains the upper portion of the address (MSB).
The index together with the tag uniquely specifies the memory address of the word contained in the cache block.

What if there's no data in a location? (ex. when the processor starts up)

Valid bit: field that indicates whether an entry contains a valid data.

Accessing a Cache

Data eviction: data in cache is moved back into memory.
If the data is different need to save it back into memory (cannot subscribe the data in cache).
Index field: used as an address to reference the cache (n bit field has 2^n values = number of entries is a power of 2).
Byte offset: words are aligned to multiples of 4 byte (32 bits), the two least significant bit of every address specify a byte within a word.



Tag = upper 52 bits of the address AND valid bit is on the request hits in the cache (word is supplied to the processor).

Otherwise, a miss occurs.

4 byte (word) * 1000 lines = 4 KB

Checks if requested tag is the same as in cache —> HIT/MISS

Block Size Considerations

Larger blocks should reduce miss rate exploiting burst effect due to spartial locality (every time a byte is accessed, the nearest bytes are brought together).

Problem!

1. ***More cache misses due to increased competition.***
   Larger blocks means few of them: number of blocks that can be held in the cache become small (more competition).
2. ***Increased miss penalty.***
   Transfer time increase as the block size expands (decreases cache performance).
   Miss penalty reduction:
   - **Early restart:** as soon as the requested word arrives, forward it to the processor (don't wait for cache line to be full).
   - **Critical word first:** start fetching the block with the required (critical) word.

Handling Cache Misses

On cache hit: CPU proceeds normally.
On cache miss: creates a pipeline stall.
Stall the processor while waiting for the block to be fetched from the next level of memory hierarchy.

- *Instruction cache miss:* restart IF.
- *Data cache miss:* complete data access.

Out-of-Ordering: allows execution of instructions while waiting for a cache miss.

Handling Writes

Inconsistency: when data is written only into the cache without changing main memory (different data).

Write-through: scheme in which writes always update both the cache and the next lower level of memory hierarchy, ensuring that data are always consistent between the two.
Problem! Can slow down the processor cause accessing memory takes time.
Solution: *write buffer between cache and RAM.*
Write buffer: queue that holds data while the data are waiting to be written in memory.
Processor continues execution after writing data into cache and write buffer.
Only ***stalls*** when the buffer is full.

Write-back: scheme that updates values only to the block in the cache, then writes the modified block to the lower level of hierarchy when the block is replaced.
Keeps track of dirty blocks and writes it into RAM only when it is replaced (eviction dirty line).
Can improve performance BUT more complex to implement.

Write miss policies

Write miss: store but data is not in cache.

- ***Allocate-on-miss:*** block is fetched from memory and then is overwritten.
- ***Write around:*** block updated in memory but not in cache.

Measuring and Improving Cache Performance

CPU time: ***program execution cycles*** and ***clock cycles spent waiting for memory*** (memory stall cycles from cache misses).

$$\text{Memory-stall clock cycles} = \frac{\overset{\text{Load/Store}}{\text{Memory accesses}}}{\text{Program}} \times \text{Miss rate} \times \text{Miss penalty} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Misses}}{\text{Instruction}} \times \text{Miss penalty}$$

```
Example: calculating cache performance
```

Given

- I-cache miss rate = 2%
- D-cache miss rate = 4%
- Miss penalty = 100 cycles
- Base CPI = 2
- Ld/Sd = 36% of all instructions

Miss cycles per instruction

- **I-cache:** 0.02 x 100 = 2
  Takes longer cause not all instructions need data but all instructions need themselves!
- **D-cache:** 0.36 x 0.04 x 100 = 1.44

Actual CPI

- 2 + 2 + 1.44 = **5.44**
- 5.44 / 2 = 2.72 *Ideal CPU is 2.72 times faster of the actual one.*

CPU performance increase (faster processor) but memory system remains the same
➡ Miss penalty more significant (amount of time spent on memory stalls will take up an increasing fraction of the execution time)
Increased clock rate maintaining the same memory system
➡ Cache misses decrease performance

AMAT: ***average memory access time***.

$$\text{AMAT} = \text{Time for a hit} + \text{Miss rate} \times \text{Miss penalty}$$

```
Example: calculating average memory access time
```

Given

- CPU with 1ns clock (1 GHz)
- Hit time = 1 cycle
- Miss penalty = 20 cycles
- I-cache miss rate = 5%

AMAT

- 1 + 0.05 x 20 = 2ns (2 cycles per instruction)

Reducing Cache Misses by More Flexible Placement of Blocks

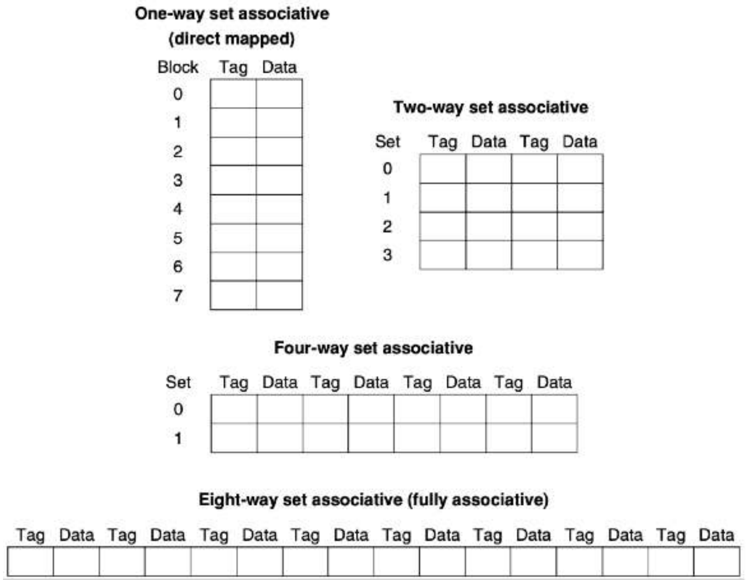
Fully associative: cache structure in which a block can be placed in any location in the cache (block may be associated with any entry, ***1 set only***).

Problem! All entries must be searched cause a block can be placed anywhere.

Solution: *use of comparators associated with each cache entry BUT expensive.*

Set associative: cache with a fixed number of locations where each block can be placed.

n-way set-associative: each set consists of n blocks.
Each block in the memory maps to a unique set given by the index field and a block can be placed in any element of that set.
Needs n comparators.
Set containing a memory block: (Block number) modulo (Number of sets)


One-way set associative
(direct mapped)
Block Tag Data
0
1
2
3
4
5
6
7
Two-way set associative
Set Tag Data Tag Data
0
1
2
3
Four-way set associative
Set Tag Data Tag Data Tag Data Tag Data
0
1
Eight-way set associative (fully associative)
Tag Data Tag Data Tag Data Tag Data Tag Data Tag Data Tag Data Tag Data

Increasing associativity decreases miss rate BUT with diminishing returns (Amdahl's Law)

Locating a Block in the Cache

- ***Direct mapped:*** access by indexing (only one comparator).
- ***Fully associative:*** no index, the entire address is compared against the tag of every block.
- ***Set associative:*** index value used to select the set, tags of all blocks in the set must be searched in parallel.

Choosing Which Block to Replace

- ***Direct mapped:*** no choice cause a block can go in one position only.
- ***Fully associative:*** each block can be replaced.
- ***Set associative:*** choice of which block to replace.

Replacement policies

Non-valid entry: checks the valid bit to see if there's data or not.

Least recently used (LRU): block replaced is the one that has been unused for the longest time (keeping track with a bit to indicate whenever an element it is referenced).
Problem with larger caches.

Random

Reducing the Miss Penalty Using Multilevel Caches

Multilevel cache: memory hierarchy with multiple levels of caches.
Additional level of caching supported by many microprocessor.

Second-level cache is accessed whenever a miss occurs in the primary cache. If it contains the requested data miss penalty first level = access time second level.
If it needs to go to main memory:

total miss penalty = second level cache access time + main memory access time