HIGH PERFORMANCE COMPUTING FOR SCIENCE AND ENGINEERING

LECTURE 16

Fabian Wermelinger
Harvard University
CS205

Thursday, March 30th 2023

LAST TIME

- Performance metrics
- Using hardware performance counters to compute metrics
- Parallel speedup and parallel efficiency
- Amdahl's Law and strong scaling
- Gustafson's Law and weak scaling
- Weak efficiency and its relation to parallel overhead and compute/transfer overlap

TODAY

Main topic: ***Instruction set architecture, instruction-level parallelism and processor pipelining***

Details:

- ***Reading assignment:*** *A New Golden Age for Computer Architecture*
  - Computer architecture, RISC and CISC.
  - Challenges for new architectures with respect to energy consumption.
- Instruction set architecture (ISA): the *vocabulary of the computer*. Different classes and history.
- Operation counts and comparison with different applications.
- Interpretation of memory addresses.
- Addressing modes and instruction encoding.
- Processor pipelining (with example based on hypothetical ISA).

WHAT WE HAVE SEEN SO FAR

Software: C/C++, OpenMP, MPI

```cpp
#include <iostream>

int main(void)
{
    std::cout << "Hello, World!\n";
    return 0;
}
```

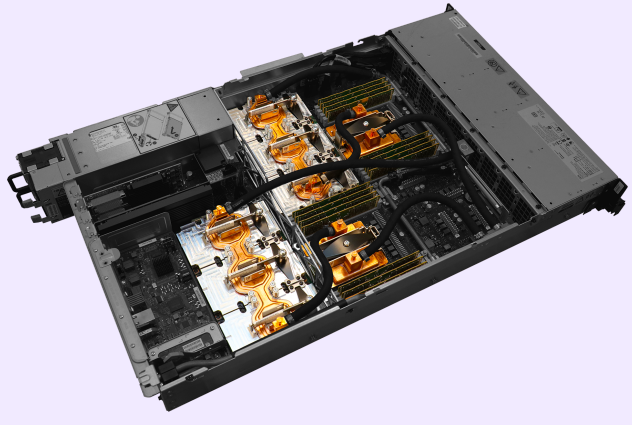
↔

Compiler

GCC, Clang, ICC, PGI, ...

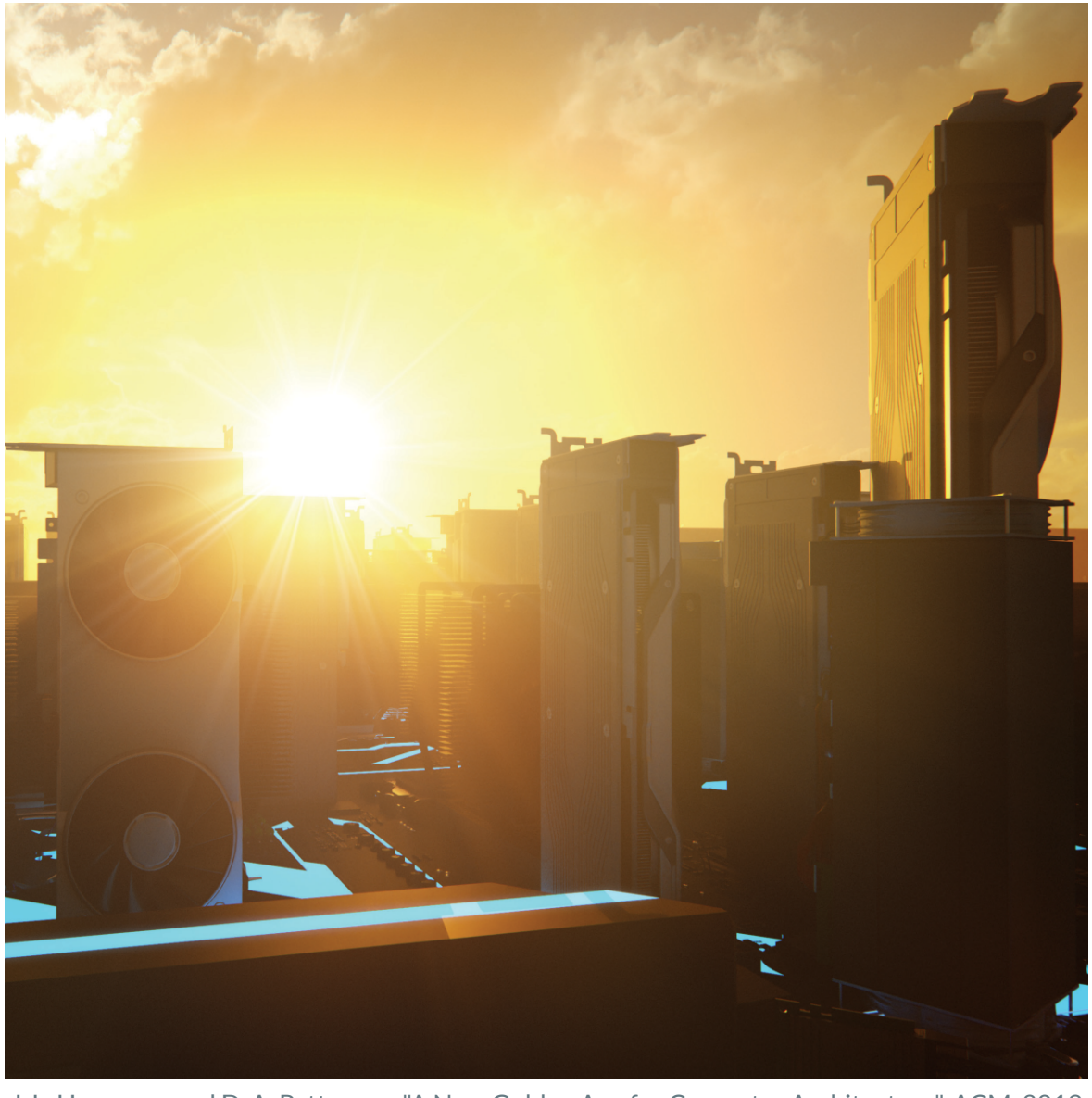
- You write software on the high-level
- For high performance you must understand how hardware works and what language it speaks.

Hardware: CPU, cores, cache, registers, memory, bus, I/O



- The compiler is the interface between high-level software and low-level machine code.
- It uses the ***Instruction Set Architecture*** (ISA) to bring high-level software and hardware together.
- There are multiple ISAs and different hardware supports different ISAs.
  → *there is no standard!*

READING ASSIGNMENT



J. L. Hennessy and D. A. Patterson, "A New Golden Age for Computer Architecture", ACM, 2019

A New Golden Age for Computer Architecture

- *"Software talks to hardware through a vocabulary called an instruction set architecture (ISA)"*
- IBM had 4(!) incompatible lines of computers, each with its own ISA.
- *ISA is messy because:* no standard, proprietary, different vendors different designs.
- RISC-V is an attempt to change that *(and it is gaining momentum!)*.

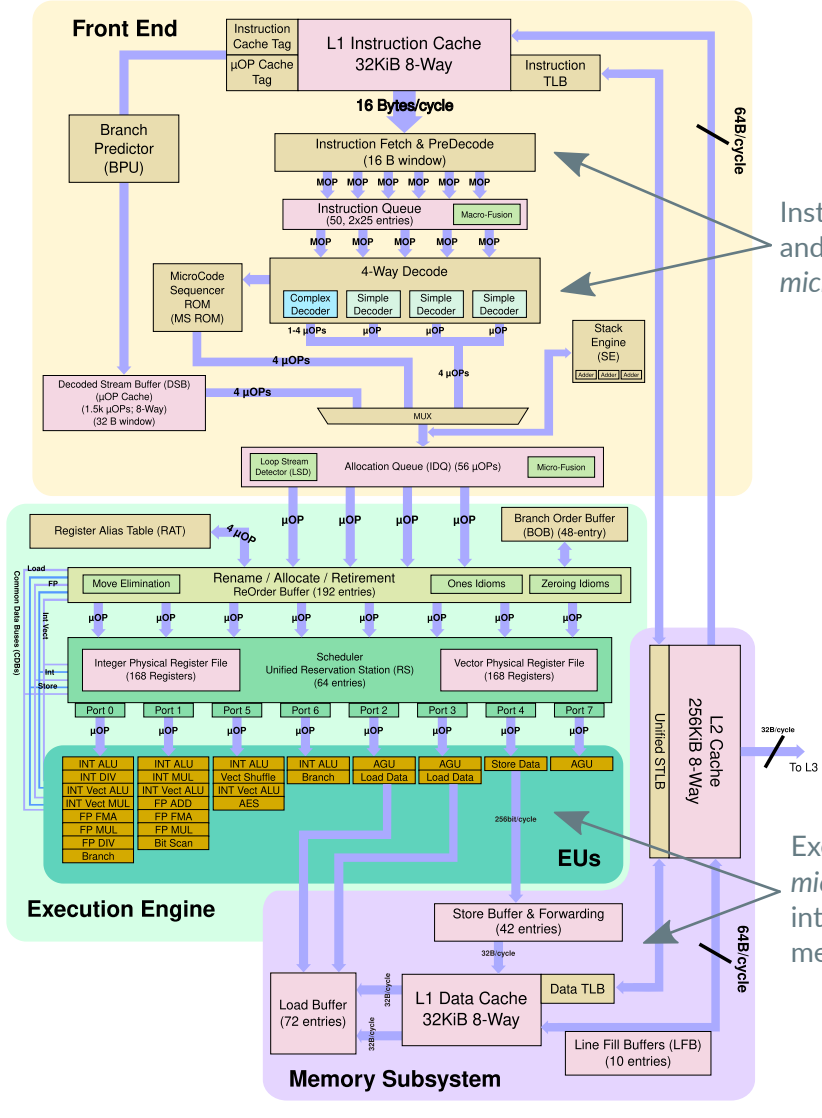
READING ASSIGNMENT: OVERVIEW

- The greatest challenge for computer designers (then and now) is the "brains", i.e., the *control hardware*.
- M. Wilkes proposed simplification of control by introducing *microinstructions* (1953).
- A conventional instruction is made up of several microinstructions
- This led to two main designs:
  1. ***CISC:*** Complex Instruction Set Computer (instructions consist of several microinstructions). ***Examples:*** Intel x86 (32-bit) and x86-64 (64-bit) dominant ISA for a long time, now less and less but still main player.
  2. ***RISC:*** Reduced Instruction Set Computer (instructions are microinstructions themselves). ***Examples:*** RISC-V, Alpha, HP-PA, MIPS, IBM Power and Sun SPARC
- Microinstuctions allow to implement a *powerful* hardware optimization called *pipelining*.
- ***No new CISC ISA has been released since decades.***

EXAMPLE: BROADWELL MICROARCHITECTURE

In-order execution

Out-of-order execution



Instruction fetch and decoding into *microinstructions*

Execution of *microinstructions* and interactions with memory subsystem

READING ASSIGNMENT: RISC VS. CISC

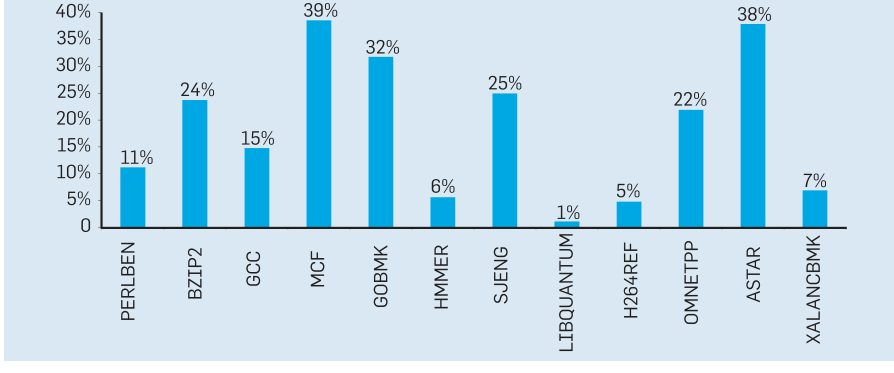
- PC sales were worldwide dominated by Intel and AMD. The products of both these vendors use the same ISA→`x86` and `x86-64` for the 32-bit and 64-bit architectures, respectively. *These are CISC ISA's.*
- Commercial RISC vendors like Alpha, HP-PA, MIPS, IBM Power or Sun SPARC were all different and suffered from fewer sales than the conventional PCs.
- The PC era peaked around 2011 and we are now in the post-PC era. *Apple helped initiate this trend by introducing the iPhone in 2007 which incorporates the technology of ARM processors (RISC machines).*
- RISC is more power efficient than CISC and consequently `x86` shipments have fallen almost 10% per year since 2011.
- On the other hand, products that contain RISC processors (phones, tablets, IoT) have skyrocketed. Today 99% of 32-bit and 64-bit processors are RISC.
- General consensus is that RISC offers better properties for a general-purpose processor. *PC era was CISC, post-PC era is RISC.*

READING ASSIGNMENT: MOORE'S LAW

- As we approach the end of Moore's Law and Dennard scaling *(power wall)*, we learned that multicore processors were introduced around 2004.
- In the paper the authors explain what exactly led to this development. One option was the further exploitation of instruction-level parallelism (ILP).
  Why did architecture designers ultimately move to multicore designs?

- ILP can be exploited through pipelining.
- Pipelines *must be full all the time* in order to be efficient.
- Architects use branch prediction to *speculatively* guess which execution path is followed.
- If prediction was wrong, the executed instructions are *wasted* and state must be reset → ***waste of energy***.
- Implementing an efficient deep pipeline is hard → ***multicore was born***.

Figure 4. Wasted instructions as a percentage of all instructions completed on an Intel Core i7 for a variety of SPEC integer benchmarks.



J. Hennessy and D. Patterson, ACM 2019

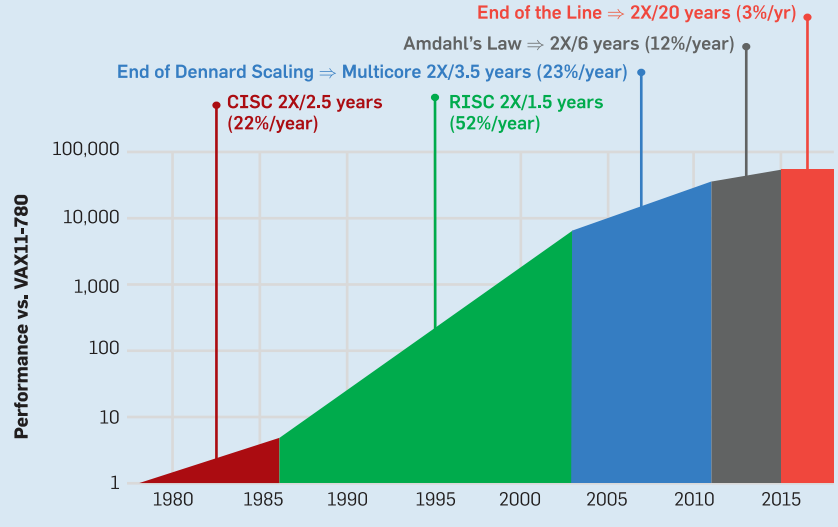
Comparison of wasted instructions due to branch misprediction using different SPEC benchmarks (https://spec.org/).

READING ASSIGNMENT: AMDAHL'S LAW

- But multicore chips do also not solve the energy efficiency problem.
- ***Where has the responsibility to identify and how to exploit parallelism shifted?***
- ***How are multicore chips not energy efficient?***

- ***The responsibility has shifted to you.***
- Idle cores consume energy too but they don't do useful work.
- ***Amdahl's Law applies without mercy:*** *If you have a 64 core processor and, if you execute on one core only 1% of the time, you waste about 40-45% of the energy! (Energy is proportional to all the 64 cores!)*
- Real programs ***are not all-or-none parallelism***, some portions of your code may execute on more than one but not all cores.

Figure 6. Growth of computer performance using integer programs (SPECintCPU).



J. Hennessy and D. Patterson, ACM 2019

Growth of the SPEC benchmark performance over time.

Free lunch is over!

READING ASSIGNMENT: THE FUTURE?

- In the paper the authors discuss the outlook for the future. ***What is the suggestion to overcome these inefficiency barriers for future architectures?***

- Two main approaches are suggested:
  1. Improve high-level languages with special compilers that generate better code. E.g. many applications written in Python would immediately benefit. What Leiserson et al. did in the first reading assignment. See Numba, PyPy or Codon for example.
  2. Must use Domain Specific Architectures (DSA) [and Domain Specific Languages (DSL)].
- Examples for DSAs: GPUs, neural network processors for deep learning (TPUs, Cerebras), SIMD, FPGA.

Benefits of DSAs:

- Exploit more efficient form of parallelism for the specific domain (e.g. SIMD is more efficient than MIMD as it only fetches one instruction stream and executes in lockstep, but SIMD is less flexible).
- Can make better use of memory hierarchy (e.g. GEMV optimized processors).
- Can use less precision when adequate (e.g. graphics and machine learning).
- Benefit from programs written in DSL that expose more parallelism.

INSTRUCTION SET ARCHITECTURE

- Defines a basic set of *"tools"* the CPU can work with → the instruction set architecture for 32-bit and 64-bit architectures is not the same (you probably realized that when you upgraded your Mac to Catalina in 2019).
- Specifies memory addressing:
  - Little endian byte order
  - Big endian byte order
- Can be a ***Reduced Instruction Set Computer*** (RISC): instruction set is limited to *simple* instructions but is *fast* (can do many instructions per cycle, possibly has to do more instructions on average since instructions are simple), *ideally the latency of one instruction is one clock cycle and the instruction format has the same size (number of bits in encoding) for all instructions*.
- Can be a ***Complex Instruction Set Computer*** (CISC): instruction set can perform complex instructions and is usually *slower* with *more complex chip circuitry*. *Instructions do not have uniform latencies and their size may differ*.
- Some example instructions we have already seen: load/store, jumps, arithmetic

INSTRUCTION SET ARCHITECTURE

- ***Key principles how computers of today work:***
  1. Instructions are represented as (binary) numbers by an encoding.
  2. Programs are stored in memory to be read or written, just like *data*.

  This is known as the *stored-program* concept and is the foundation for the multi-purpose computer.
- Programs are *compiled machine code stored in main memory*.
- Even the compiler that generates new programs is represented like that in memory (need a compiler-compiler to build a compiler 😖).
- One consequence for the binary representation of programs is that the computer that runs them must be *compatible* with the used instruction set architecture. This is one of the reasons for the dominance of `x86`. Commercial software is usually shipped in binary form that requires a matching instruction (CPU) set to be executed.

ISA COMPATIBILITY

Consider the following two cases for compatibility of a binary executable:

Case 1: incompatible on hardware level

- Consider the code:

```
1 double f(double a, double b)
2 {
3     return a + b;
4 }
```

- Compiled on RaspberryPi (*ARM, RISC*), the machine code is:

```
1 Disassembly of section .text:
2
3 0000000000000000 <f>:
4    0:   1e612800        fadd    d0, d0, d1
5    4:   d65f03c0        ret
```

- Compiled on laptop (*Intel, CISC*), the machine code is:

```
1 Disassembly of section .text:
2
3 0000000000000000 <f>:
4    0:   f2 0f 58 c1     addsd   %xmm1,%xmm0
5    4:   c3              ret
```

Observe:

- The RISC format takes *three* instruction operands, Intel's CISC only *two* (different instruction format).
- Encoding on RISC is ***fixed size***: *both instructions are encoded by 4 bytes*, `0x1e612800` and `0xd65f03c0`, respectively. The function `f` takes total of 8 bytes machine code on RISC.
- Encoding on CISC is ***variable size***: *the floating-point add is encoded by 4 bytes and the return by 1 byte*, `0xf20f58c1` and `0xc3`, respectively. On CISC `f` is encoded by 5 bytes, 38% smaller than the RISC encoding.
- ***The encodings used in RISC and CISC machines are incompatible!***

ISA COMPATIBILITY

Consider the following two cases for compatibility of a binary executable:

Case 2: *incompatible on operating system level*

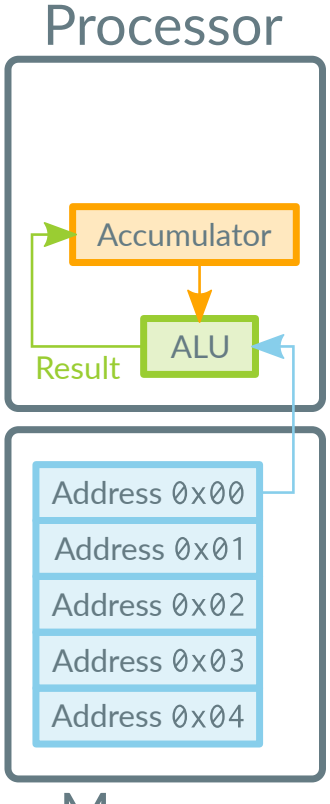
- Assume now you have two laptops, both with the *same Intel CPU* but you run *Linux* on one of them and *Windows* on the other.
- Both machines have the *same instruction set architecture* so the *machine code is compatible*.
- A *binary* compiled on Linux is however not compatible to run on Windows *because of the different operating systems*. The executable and linking formats are different on the two operating systems and the *executable binaries* are therefore incompatible (not the ISA, hardware). The ***A****pplication* ***B****inary* ***I****nterface* (ABI) defines calling conventions for machine code and linkage to shared libraries. Linux and Windows use different ***A****pplication* ***P****rogramming* ***I****nterfaces* (API) to enable the ABI on the system and are therefore *incompatible*.
- Linux uses the *Executable and Linkable Format* (ELF) and Windows uses the *Portable Executable* format (PE)
- When you compile code into *assembly* code on Linux and you would then link that assembly code to an executable on Windows, it would be able to run on that OS because the instruction set architecture (hardware) is the same.

The two machines are ISA compatible but the ABI's are different.

CLASSES OF INSTRUCTION SET ARCHITECTURES

Historical accumulator architectures:

- These architectures had *only one* register which is called the *accumulator*.
- *All operands for instructions were memory addresses*. For example, `ADD 200` would fetch the data at address `200` and add it into the accumulator.

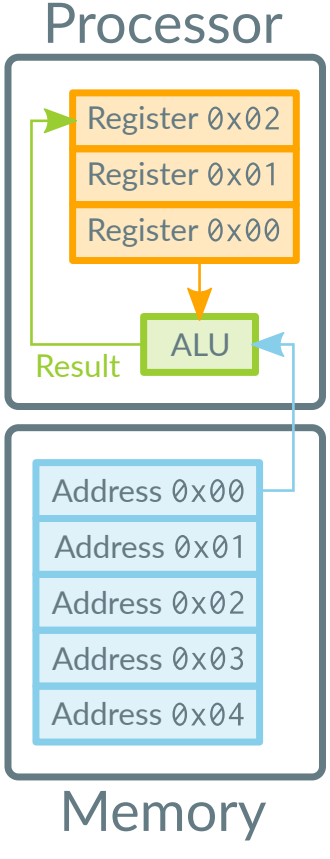


Digital Equipment Corporation (DEC) PDP-8, 12-bit accumulator architecture, 1965.

CLASSES OF INSTRUCTION SET ARCHITECTURES

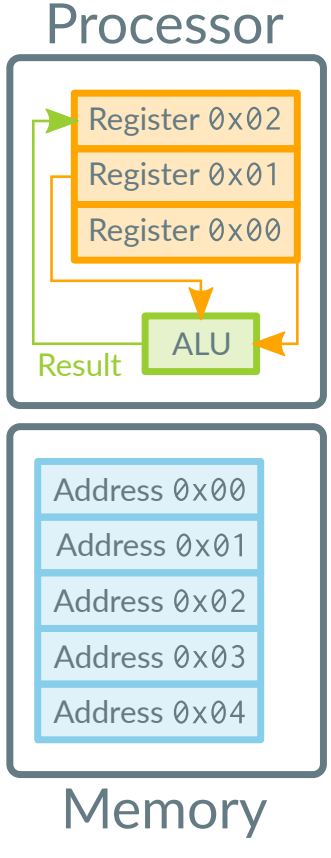
Today:

- Multiple general purpose registers available.
- RISC-V or x86 are *general-purpose register architectures*. These may be further divided:



Register-memory:

- One of the instruction operands may be a *memory address*.
- Usually CISC architectures.
- ***Examples:*** x86, x86-64, Motorola 68000



Load-store:

- *Only registers* are allowed for instruction operands.
- Usually RISC architectures.
- All data must be fetched into/from registers using `load` or `store` instructions.
- ***Examples:*** RISC-V, IBM PowerPC, Sun SPARC.

CLASSES OF INSTRUCTION SET ARCHITECTURES

Advantages and disadvantages:

Type	Advantages	Disadvantages
Load-store (0,3)	Simple, fixed-length instruction encoding. Simple code generation model. Instructions take similar numbers of clocks to execute.	Higher instruction count than architectures with memory references in instructions. More instructions and lower instruction density lead to larger programs in terms of size (may not be ideal for embedded applications).
Register-memory (1,2)	Data can be accessed without a separate load instruction first. Instruction format tends to be easy to encode and yields good density.	Operands are not equivalent since a source operand in a binary operation is destroyed. Encoding a register number and a memory address in each instruction may restrict the number of registers. Cycles per instruction vary by operand location.
Memory-memory (2,2) or (3,3)	Most compact. Does not waste registers for temporaries.	Large variation in instruction size, especially for three-operand instructions. In addition, large variation in work per instruction. Memory accesses create memory bottleneck. *(Not used today)*

The numbers in parenthesis mean:

(*number of allowed memory addresses for instruction operands*, *maximum number of instruction operands*).

CLASSES OF INSTRUCTION SET ARCHITECTURES

Historical evolution of various architectures:

Machine	Number of general-purpose registers	Architectural style	Year
EDSAC	1	Accumulator	1949
IBM 701	1	Accumulator	1953
CDC 6600	8	Load-store	1963
IBM 360	16	Register-memory	1964
DEC PDP-8	1	Accumulator	1965
DEC PDP-11	8	Register-memory	1970
Intel 8008	1	Accumulator	1972
Motorola 6800	2	Accumulator	1974
DEC VAX	16	Register-memory, memory-memory	1977
Intel 8086	1	Extended accumulator	1978
Motorola 68000	16	Register-memory	1980
Intel 80386	8	Register-memory	1985
ARM	16	Load-store *Your phone or Apple M1/2*	1985
MIPS	32	Load-store	1985
HP PA-RISC	32	Load-store	1986
SPARC	32	Load-store	1987
PowerPC	32	Load-store	1992
DEC Alpha	32	Load-store	1992
HP/Intel IA-64	128	Load-store	2001
AMD64 (EMT64)	16	Register-memory *Your PC/laptop/Cluster*	2003
RISC-V	32	Load-store	2010

Patterson and Hennessy, 2017

OPERATIONS IN AN INSTRUCTION SET

Fundamental: every computer must provide these basic instructions:

Operator Type	Examples
Arithmetic and logical	***Integer arithmetic*** and logical operations: add, subtract, and, or, multiply, divide
Data transfer	Load and stores; move instructions memory addressing
Control	Branch, jump, procedure call and return, traps
System	Operating system call, virtual memory management instructions

Computer dependent: instructions with higher complexity, often composed of multiple microinstructions (some are only available on special hardware):

Operator Type	Examples
Floating point	Floating-point operations: add, subtract, multiply, divide, compare
Decimal	Decimal add, decimal multiply, decimal-to-character conversions
String	String move, string compare, string search
Graphics	Pixel and vertex operations, compression and decompression operations

Most important for us

OPERATIONS IN AN INSTRUCTION SET

Example: SPEC92 benchmark

(http://www.mrob.com/pub/comp/benchmarks/spec.html#CPU_92)

Integer codes:

Table 9.3 CINT92—SPARC instruction counts.

Benchmarks	Memory %	Branch %	Other %	Millions of Instructions
008.espresso	28.1	20.4	51.5	2,931
022.li	**33.3**	23.7	*43.0*	**4,661**
023.eqntott	*16.5*	**26.5**	57.0	1,322
026.compress	25.7	*16.6*	**57.7**	2,699
072.sc	24.1	21.8	54.1	*1,255*
085.gcc	26.6	20.2	53.2	4,624
TOTAL	27.3	23.3	49.4	17,692

Memory = loads + stores; Other = total – memory – branch.
Italics indicates the lowest number in the column.
Bold indicates the highest number in the column.

http://jimgray.azurewebsites.net/benchmarkhandbook/chapter9.pdf

008	Optimizing tool (PLA)	14800 SLOC
022	LISP interpreter	7700 SLOC
023	Conversion of equations to truth table	3500 SLOC
026	Text compression	1500 SLOC
072	Spreadsheet application	8500 SLOC
085	GNU compiler	87800 SLOC

Average integer instructions in all of the SPECint92 benchmarks:

Rank	80x86 instruction	Integer average (% total executed)
1	load	22%
2	conditional branch	20%
3	compare	16%
4	store	12%
5	add	8%
6	and	6%
7	sub	5%
8	move register-register	4%
9	call	1%
10	return	1%
Total		**96%**

Figure A.13 The top 10 instructions for the 80x86. Simple instructions dominate this list and are responsible for 96% of the instructions executed. These percentages are the average of the five SPECint92 programs.

Hennessy and Patterson, 2011

Memory operations and branching dominate in these integer programs (which represent the majority of tasks in integer programs).

OPERATIONS IN AN INSTRUCTION SET

Example: SPEC92 benchmark

(http://www.mrob.com/pub/comp/benchmarks/spec.html#CPU_92)

Integer codes:

Table 9.3 CINT92—SPARC instruction counts.

Benchmarks	Memory %	Branch %	Other %	Millions of Instructions
008.espresso	28.1	20.4	51.5	2,931
022.li	**33.3**	23.7	*43.0*	**4,661**
023.eqntott	*16.5*	**26.5**	57.0	1,322
026.compress	25.7	*16.6*	**57.7**	2,699
072.sc	24.1	21.8	54.1	*1,255*
085.gcc	26.6	20.2	53.2	4,624
TOTAL	27.3	23.3	49.4	17,692

Memory = loads + stores; Other = total – memory – branch.
Italics indicates the lowest number in the column.
Bold indicates the highest number in the column.

http://jimgray.azurewebsites.net/benchmarkhandbook/chapter9.pdf

008	Optimizing tool (PLA)	14800 SLOC
022	LISP interpreter	7700 SLOC
023	Conversion of equations to truth table	3500 SLOC
026	Text compression	1500 SLOC
072	Spreadsheet application	8500 SLOC
085	GNU compiler	87800 SLOC

Floating-point codes:

Table 9.5 CFP92 - SPARC instruction counts.

Benchmarks	Memory %	Fpop %	Branch %	Other %	Millions of Instructions
013.spice2g6	25.0	*4.2*	**13.6**	**57.2**	**22,878**
015.doduc	29.4	25.9	8.4	36.2	*1,303*
034.mdljdp2	26.7	47.7	2.0	23.7	3,217
039.wave5	33.7	21.9	7.4	37.0	4,432
047.tomcatv	45.0	32.9	1.8	20.3	1,406
048.ora	*13.2*	**56.3**	7.4	23.1	1,695
052.alvinn	**48.3**	27.6	5.1	18.9	3,506
056.ear	35.3	26.4	6.2	32.1	17,768
077.mdljsp2	21.4	52.8	2.4	23.5	3,017
078.swm256	43.2	46.6	*0.6*	*9.5*	10,810
089.su2cor	33.3	31.8	4.3	30.6	4,915
090.hydro2d	28.5	38.0	3.9	29.6	7,891
093.nasa7	38.9	28.0	5.0	28.1	5,917
094.fpppp	47.2	34.7	3.0	15.1	5,076
TOTAL	33.2	27.2	6.5	33.1	93,741

Memory = loads + stores; Fpop = floating-point operations;

http://jimgray.azurewebsites.net/benchmarkhandbook/chapter9.pdf

034	Molecular dynamics (DP, Fortran)	4500 SLOC
048	Optical ray tracing (DP, Fortran)	500 SLOC
052	Neural network (SP, C)	300 SLOC
077	Molecular dynamics (SP, Fortran)	3900 SLOC
078	Shallow water equations (SP, Fortran)	500 SLOC
090	Navier-Stokes equations (DP, Fortran)	4500 SLOC

INTERPRETATION OF MEMORY ADDRESSES

- Modern ISA are ***byte addressed*** → each byte in physical memory is addressed with *one* virtual memory address:
  - 32-bit ISA: can address 4 gigabyte of memory (*pointers are 4 bytes*)
  - 64-bit ISA: can address 16 exabyte of memory (*pointers are 8 bytes*)
- The size of general-purpose registers in modern 64-bit ISA is ***64-bit*** *(must fit the size of a pointer).*
- The ISA provides *multiple instructions for different operand sizes*.

Historically the following terminology is used: (in assembly)

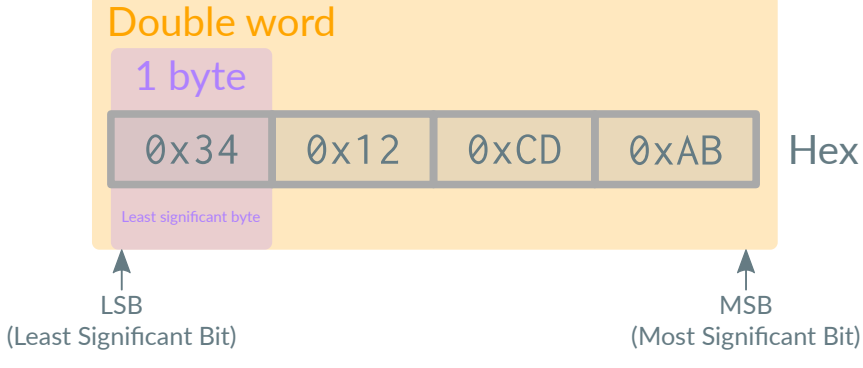
Bits	Name	Alternative
8	byte	—
16	half word	word
32	word	double word
64	double word	quad word

INTERPRETATION OF MEMORY ADDRESSES

- Two conventions for the *byte order* in words:
  - ***Little endian:*** use address of the *rightmost* byte (least significant byte)
  - ***Big endian:*** use address of the *leftmost* byte (most significant byte)
- In general you do not need to worry about endianness. We have talked about it when discussing MPI communication with heterogeneous machines.

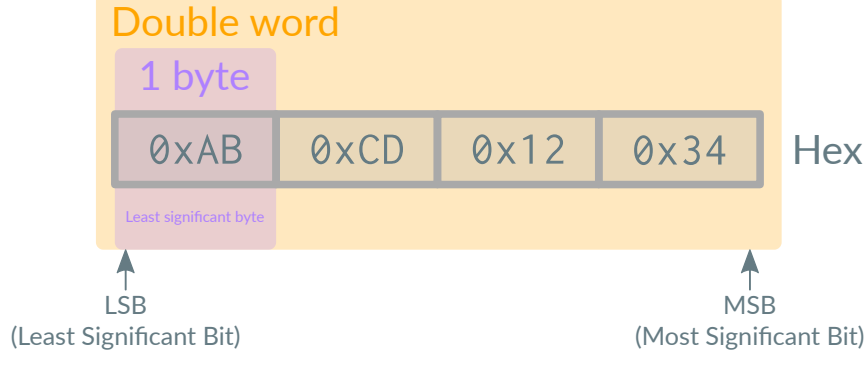
Consider the following 4-byte sequence in hex: `0xABCD1234`

Little endian:



Examples: Intel, AMD

Big endian:



Examples: IBM, SPARC

INTERPRETATION OF MEMORY ADDRESSES

- Endianness is one issue related to the interpretation of bytes in the ISA (you do not need to worry too much about it most of the time).
- Another issue is ***byte alignment***.
- Most ISA do support misaligned objects so alignment *is not* a strict requirement.
- Misaligned load/store instructions incur a *performance cost* (***recall:*** *cache lines*).
- An object is aligned at byte boundary `align_at` if the object's base address divides evenly by the desired byte boundary:

```cpp
#include <cstdio>
int main(void)
{
    int object;
    size_t base = reinterpret_cast<size_t>(&object);
    size_t align_at = 16; // 16-byte boundaries
    if (0 == base % align_at) {
        printf("Object at address %p is aligned at %lu byte boundaries\n",
               &base,
               align_at);
    } else {
        printf("Object at address %p is NOT aligned at %lu byte boundaries\n",
               &base,
               align_at);
    }
    return 0;
}
```

General rule:
Programs with aligned data structures run faster when you exploit data-level parallelism (SIMD). The byte boundary for the alignment is given by the SIMD vector width.

INTERPRETATION OF MEMORY ADDRESSES

	Value of 3 low-order bits of byte address							
Width of object	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
1 byte (byte)	Aligned	Aligned	Aligned	Aligned	Aligned	Aligned	Aligned	Aligned
2 bytes (half word)	Aligned		Aligned		Aligned		Aligned	
2 bytes (half word)		Misaligned		Misaligned		Misaligned		Misaligned
4 bytes (word)	Aligned				Aligned			
4 bytes (word)		Misaligned				Misaligned		
4 bytes (word)			Misaligned				Misaligned	
4 bytes (word)				Misaligned				Misaligned
8 bytes (double word)	Aligned							
8 bytes (double word)		Misaligned						
8 bytes (double word)			Misaligned					
8 bytes (double word)				Misaligned				
8 bytes (double word)					Misaligned			
8 bytes (double word)						Misaligned		
8 bytes (double word)							Misaligned	
8 bytes (double word)								Misaligned

Figure A.5 Aligned and misaligned addresses of byte, half-word, word, and double-word objects for byte-addressed computers. For each misaligned example some objects require two memory accesses to complete. Every aligned object can always complete in one memory access, as long as the memory is as wide as the object. The figure shows the memory organized as 8 bytes wide. The byte offsets that label the columns specify the low-order 3 bits of the address.

Hennessy and Patterson, 2011

Note: use the `alignof` opeartor in C++ to determine and the `alignas` operator to enforce the alignment requirement of a type `T`.

INTERPRETATION OF MEMORY ADDRESSES

Example: how do common allocators align memory

- Consider the objects shown on the left below.
- How are these objects aligned in memory by default? (See the `allocator_test` supplementary code for this lecture)

```
1 struct BadObject_t {
2     char buf[57];
3 };
```

- Object is 57 byte.
- p is an array of 2 such objects.

Allocator	Linux 64-bit, GCC 11.2.0	MacOS 64-bit, Clang 12.0.5
new	p[0] [57]: misaligned by 48 bytes p[1] [57]: misaligned by 41 bytes	p[0] [57]: misaligned by 32 bytes p[1] [57]: misaligned by 25 bytes
malloc	p[0] [57]: misaligned by 48 bytes p[1] [57]: misaligned by 41 bytes	p[0] [57]: misaligned by 32 bytes p[1] [57]: misaligned by 25 bytes
posix_memalign	p[0] [57]: misaligned by 0 bytes p[1] [57]: misaligned by 57 bytes	p[0] [57]: misaligned by 0 bytes p[1] [57]: misaligned by 57 bytes

```
1 struct NiceObject_t {
2     char buf[57];
3     char pad[7];
4 };
```

- Object padded to 64 byte.
- p is an array of 2 such objects.
- Instead of padding use `alignas`.

Allocator	Linux 64-bit, GCC 11.2.0	MacOS 64-bit, Clang 12.0.5
new	p[0] [64]: misaligned by 48 bytes p[1] [64]: misaligned by 48 bytes	p[0] [64]: misaligned by 0 bytes p[1] [64]: misaligned by 0 bytes
malloc	p[0] [64]: misaligned by 48 bytes p[1] [64]: misaligned by 48 bytes	p[0] [64]: misaligned by 0 bytes p[1] [64]: misaligned by 0 bytes
posix_memalign	p[0] [64]: misaligned by 0 bytes p[1] [64]: misaligned by 0 bytes	p[0] [64]: misaligned by 0 bytes p[1] [64]: misaligned by 0 bytes

Take-away: do not rely on the compiler for alignment. Enforce it when you need it.

GENERAL ISA ADDRESSING MODES

- Modern ISA support the following addressing modes. Suppose we perform addition with the add instruction on a load-store or register-memory architecture.
- When a memory location is used, the actual memory address specified by the addressing mode is called the *effective address*.
- Perform the addition with values in *registers* R4 and R3, store result in register R4.
- *Immediate* constants are allowed (depending on ISA limited to few bits only).
- Example for *dereferencing* a pointer (*indirection*).
- Stepping through array where d is the number of bytes per element.

Addressing mode	Example instruction	Meaning	When used
Register	`Add R4,R3`	`Regs[R4] ← Regs[R4] + Regs[R3]`	When a value is in a register.
Immediate	`Add R4,#3`	`Regs[R4] ← Regs[R4] + 3`	For constants.
Displacement	`Add R4,100(R1)`	`Regs[R4] ← Regs[R4] + Mem[100 + Regs[R1]]`	Accessing local variables (+ simulates register indirect, direct addressing modes).
Register indirect	`Add R4,(R1)`	`Regs[R4] ← Regs[R4] + Mem[Regs[R1]]`	Accessing using a pointer or a computed address.
Indexed	`Add R3,(R1 + R2)`	`Regs[R3] ← Regs[R3] + Mem[Regs[R1] + Regs[R2]]`	Sometimes useful in array addressing: R1 = base of array; R2 = index amount.
Direct or absolute	`Add R1,(1001)`	`Regs[R1] ← Regs[R1] + Mem[1001]`	Sometimes useful for accessing static data; address constant may need to be large.
Memory indirect	`Add R1,@(R3)`	`Regs[R1] ← Regs[R1] + Mem[Mem[Regs[R3]]]`	If R3 is the address of a pointer p, then mode yields $*p$.
Autoincrement	`Add R1,(R2)+`	`Regs[R1] ← Regs[R1] + Mem[Regs[R2]]` `Regs[R2] ← Regs[R2] + d`	Useful for stepping through arrays within a loop. R2 points to start of array; each reference increments R2 by size of an element, d.
Autodecrement	`Add R1, –(R2)`	`Regs[R2] ← Regs[R2] – d` `Regs[R1] ← Regs[R1] + Mem[Regs[R2]]`	Same use as autoincrement. Autodecrement/-increment can also act as push/pop to implement a stack.
Scaled	`Add R1,100(R2)[R3]`	`Regs[R1] ← Regs[R1] + Mem[100 + Regs[R2] + Regs[R3] * d]`	Used to index arrays. May be applied to any indexed addressing mode in some computers.

Figure A.6 Selection of addressing modes with examples, meaning, and usage. In autoincrement/-decrement and scaled addressing modes, the variable *d* designates the size of the data item being accessed (i.e., whether the instruction is accessing 1, 2, 4, or 8 bytes). These addressing modes are only useful when the elements being accessed are adjacent in memory. RISC computers use displacement addressing to simulate register indirect with 0 for the address and to simulate direct addressing using 0 in the base register. In our measurements, we use the first name shown for each mode. The extensions to C used as hardware descriptions are defined on page A-36.

Hennessy and Patterson, 2011

INSTRUCTION ENCODING

- The instruction encoding (instruction format) is an important decision in every ISA design.
- Here is where things become incompatible (see earlier slide).
- The *size* of the compiled program highly depends on the instruction encoding.
- The *performance* is dependent on the instruction encoding.
- The instruction encoding *depends* on the decisions made for supported addressing modes (see previous slide).
- The *cost* of the implementation is a factor too.

- The encoding should be compatible with previous releases and is usually *hard* to change without breaking a lot of software once the initial release is out (*binary compatibility*).
- The instruction encoding is *different* for different ISAs: RISC-V is not the same as x86 for example. There are basic encoding variations:
  - ***Variable:*** encoding size is different among instructions. Optimizes for *size* at the cost of performance (decoding). CISC architectures, e.g., x86 instructions may range from 1 to 15 byte.
  - ***Fixed:*** encoding size is *constant* among all instructions. Optimizes for performance (regularity) at the cost of larger code size. RISC architectures, e.g., RISC-V or PowerPC.
  - ***Hybrid:*** a *blend* of variable and fixed.

INSTRUCTION ENCODING

Example: integer addition with `add` instruction `a = a + b`. Assume that `add` is implemented as a 2-operand instruction where the result is stored back to `a`.

ISA A (CISC):

Both operands can be a *memory address* (not used today). *1 instruction* to complete the operation.

```
1 add a, b
```

ISA B (CISC):

One operand can be a *memory address* (common). *3 instructions* to complete the operation.

```
1 load R0, b // into register R0
2 add R0, a  // add a to R0
3 store a, R0
```

ISA C (RISC):

No operand can be a memory address (*only registers*). *4 instructions* to complete the operation.

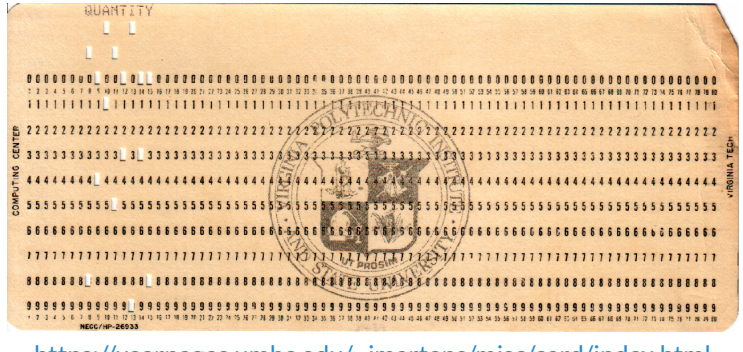
```
1 load R0, a // into register R0
2 load R1, b // into register R1
3 add R0, R1 // add R1 to R0
4 store a, R0
```

- ISA A requires the least number of instructions which will result in smaller binary sizes on average.
- ISA A does not need any registers to conduct the operation.
- ISA B and C require registers to store *temporaries* and the larger instruction count results in larger executable size. Both of these characteristics are more pronounced in RISC architectures.

INSTRUCTION ENCODING: STORED-PROGRAM COMPUTER

Encoded instructions are *simply data* that is stored in memory.

- Encoded instructions are binary data that must be *loaded from main memory* and pass through the *cache hierarchy* in the same way as other data not related to instructions does.
- Most modern architectures have a *separate L1 cache* for instructions and data because the accessing pattern between the two is different.
- The compiler may apply optimizations that allow instructions to be executed *out-of-order* (ILP, superscalar).
- The *program counter* (*instruction pointer*) is a special *register* on the hardware that holds the address of the next instruction to be executed.
- Computers that hold the program code in *memory* are called *stored-program computers*.



Punchcards have been used in early computers to store programs.

https://userpages.umbc.edu/~jmartens/misc/card/index.html

PIPELINING

Pipelining is an implementation technique on the *hardware* that takes advantage of the parallelism that exists among the actions required to execute an instruction.

- Pipelining is the *key* technique to make modern CPUs *fast*.
- All processors since about 1985 use pipelining to improve their performance by exploiting *instruction-level parallelism (ILP)*.
- It is based on the fact that the execution of *one instruction requires multiple operations (each takes at least one cycle)* which could be done in parallel if there are sufficient instructions.
- For that reason, pipelining reduces the *number of clock cycles per instruction (CPI)*. The inverse of CPI is the number of instructions per cycle *(IPC)*.
- ***Example:*** a pipeline is similar to an assembly line in an automobile factory where the conveyor belt feeds parts continuously. In case of a CPU pipeline we feed instructions.

SIMPLE ISA SPECIFICATION

We will study pipelining based on a simple load-store RISC architecture.

- All instructions work on data stored in *registers*.
- The only operations that affect memory are load and store operations (move data at a memory address to/from registers).
- The instruction format consists of a few elements with all instructions being the same size.

With these guidelines we can construct a simple instruction set, where we assume that *every instruction* can be implemented in *at most 5 clock cycles*:

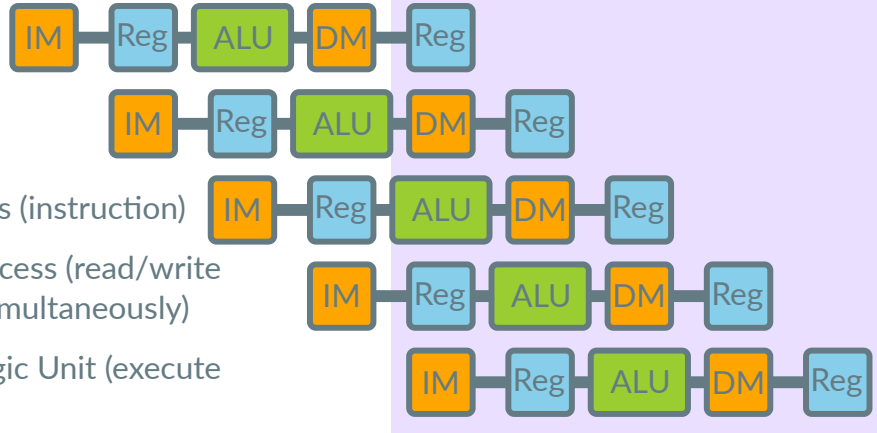
1. ***Instruction fetch cycle (IF):*** Fetch the current instruction from memory.
2. ***Instruction decode/register fetch cycle (ID):*** Decode and read register source specifiers. Performs a test to detect a possible branch instruction.
3. ***Execution/effective address cycle (EX):*** The ALU performs the operation depending on the fetched instruction.
4. ***Memory access (MEM):*** In case of a *load*, the memory is read using the *effective address*. In case of a *store*, the register with value obtained from cycle 2 is written using the *effective address*.
5. ***Write-back cycle (WB):*** Write the result of cycle 3 or cycle 4 (if it was a *load*) back to the *register* file.

5-STAGE PIPELINE

Using our simple instruction set defined on the previous slide, we can construct a *5-stage* pipeline:

Instruction number	*Clock cycle* 1	2	3	4	5	6	7	8	9
Instruction `i`	IF	ID	EX	MEM	WB				
Instruction `i+1`		IF	ID	EX	MEM	WB			
Instruction `i+2`			IF	ID	EX	MEM	WB		
Instruction `i+3`				IF	ID	EX	MEM	WB	
Instruction `i+4`					IF	ID	EX	MEM	WB

...



IM: Memory access (instruction)

Reg: Register file access (read/write can be done simultaneously)

ALU: Arithmetic-Logic Unit (execute instruction)

DM: Memory access (data)

- The *fastest* instruction in our ISA takes *2 cycles* (*branch instruction*).
- A *store* instruction finishes after *4 cycles*.
- Any other instruction takes *5 cycles* to complete.
- The pipeline *throughput* is bounded by the instruction with the highest latency which is 5 cycles in our ISA.
- The pipeline is *full* in the 5th cycle and has full throughput for the following instructions.

A full pipeline has a throughput of *one instruction every clock cycle*. A *5-fold* speedup relative to instructions with 5-cycle latencies. A pipeline is an example of instruction-level parallelism (ILP).

PIPELINE EXAMPLE

We use our pipeline implementation from the previous slides. Assume that a clock cycle takes $t_c = 1$ *ns* and we have the following *instruction mix*: 35% branch instructions, 20% store instructions and 45% other instructions. Furthermore, due to clock skew and setup time, the pipelined processor has a $t_o = 0.2$ *ns* overhead compared to a non-pipelined processor. *What is the speedup of the pipelined processor?*

1. Compute the average clock cycles per instruction (CPI) given the instruction mix:

$$\text{CPI} = 2 \times 0.35 + 4 \times 0.20 + 5 \times 0.45 = 3.75 \text{ cycles/instruction}$$

2. Average execution time of a non-pipelined processor:

$$T_s = t_c \times \text{CPI} = 1 \text{ ns} \times 3.75 = 3.75 \text{ ns}$$

3. Average execution time of a pipelined processor (assuming a full pipeline with $\text{CPI} = 1$):

$$T_p = t_c + t_o = 1 \text{ ns} + 0.2 \text{ ns} = 1.2 \text{ ns}$$

4. The speedup is given by Amdahls Law:

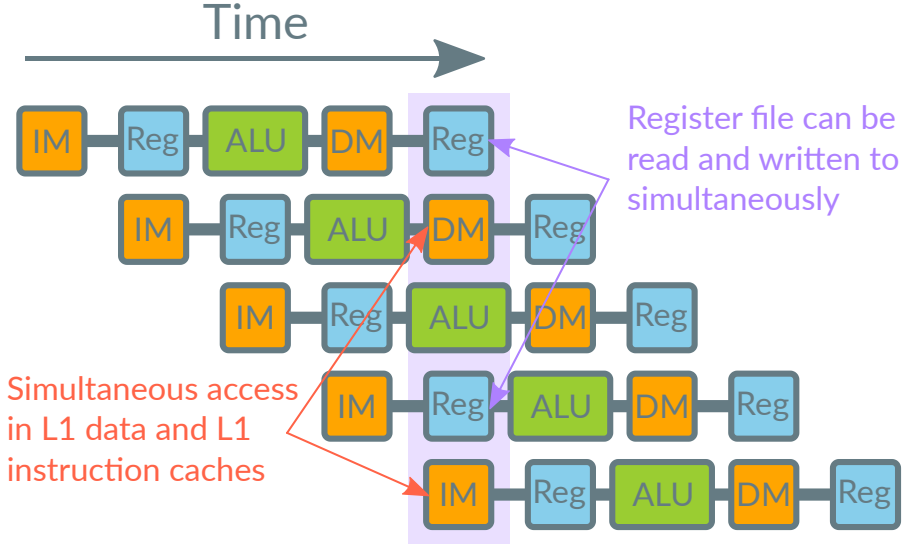
$$S_p = \frac{T_s}{T_p} = \frac{3.75 \text{ ns}}{1.2 \text{ ns}} = 3.125$$

The ideal speedup for this pipeline is 5, resulting in an efficiency of 62.5% for the given instruction mix.

PIPELINE HAZARDS

Pipeline hazards prevent the execution of the next instruction in the instruction stream. There are *three* classes of hazards:

1. ***Structural hazards:*** arise from resource conflicts when the hardware cannot support all possible combinations simultaneously.
2. ***Data hazards:*** the next instruction may depend on the result of the previous instruction.
3. ***Control hazards:*** arise from *branching* and other instructions that change the program counter.



- In each cycle it must be ensured that *all data paths are different* to avoid *structural hazards*.
- Register files allow *simultaneous read and write* accesses in the same clock cycle. A fundamental reason that pipelining works in load-store and register-memory architectures.
- The ALU can only execute *one microinstruction* per cycle for our example ISA and hardware (*scalar*).
- Here it should become clear why there are two L1 caches for data and instructions. The full pipeline requires simultaneous access to encoded instructions and data. Two separate caches avoid structural hazards.

PIPELINES IN REALITY

In a realistic scenario, pipeline stalls must also be counted in the CPI:

Pipeline CPI = Ideal pipeline CPI + Structural stalls + Data stalls + Control stalls

Processor classifications:

CPI > 1: subscalar

Processors without a pipeline implementation.

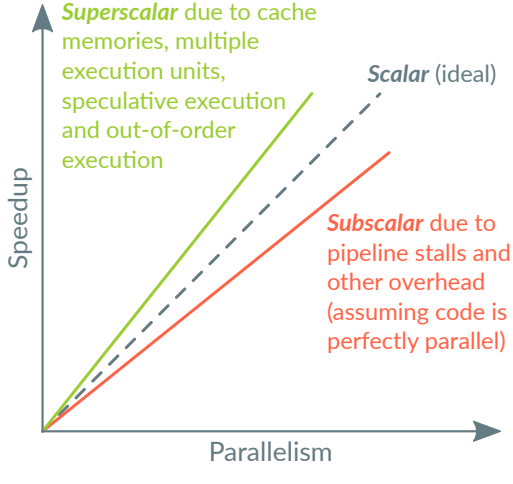
CPI = 1: scalar

Pipelined processors.

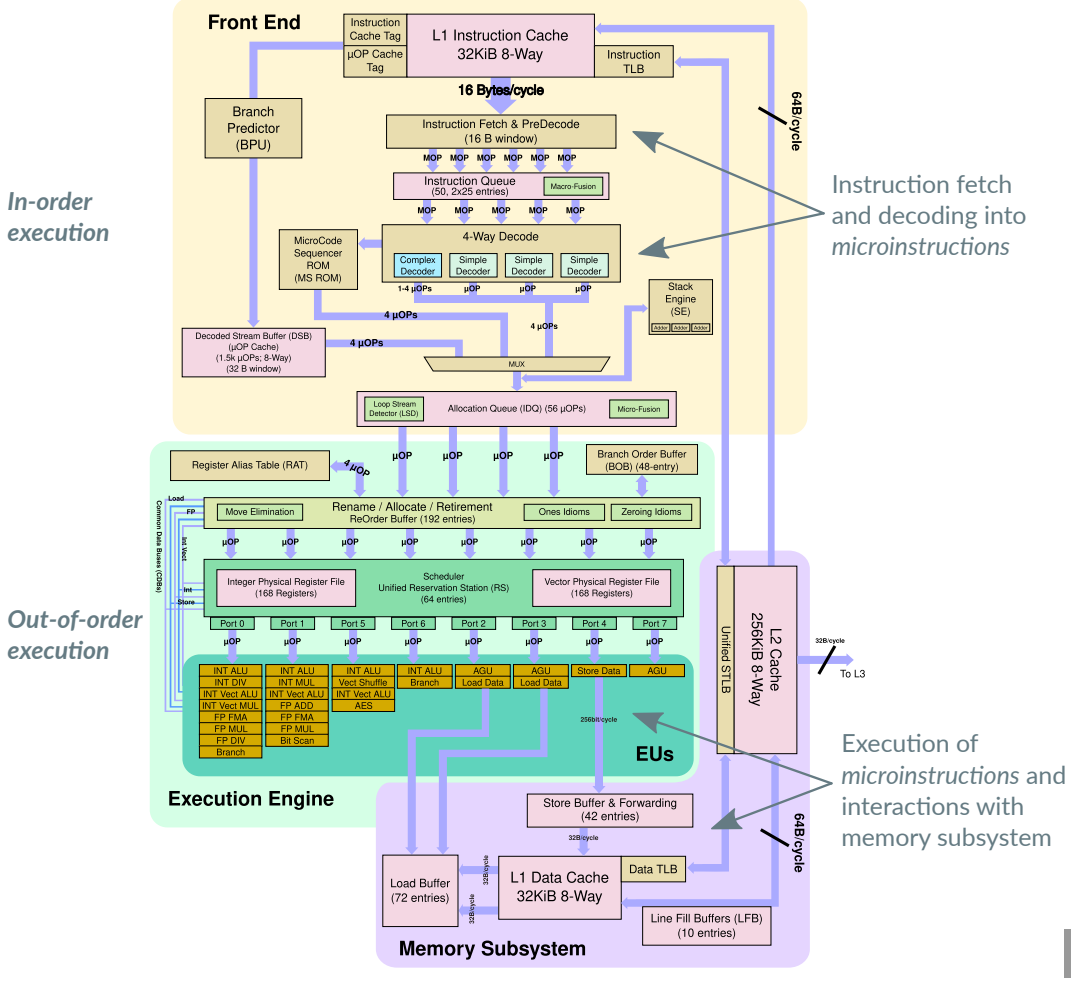
CPI < 1: superscalar

Pipelined processors with multiple execution units, speculative execution and out-of-order execution.

Amdahl's Law:



Intel Broadwell architecture:



RECAP

- The instruction set architecture is the basic language of the computer.
- Understanding its basic principles is important to understand how a computer works.
- The concept of a stored-program computer enables the general-purpose computer as we use it today.
- On average, about 22% of the instructions executed in a program are load instructions and 12% are store instructions.
- Pipelining is an important optimization technique used since 1985 to improve the performance of a computer.
- The architectures of modern CPUs use complex and deep pipelines together with techniques such as out-of-order execution, multiple execution units and speculative execution to further leverage instruction-level parallelism.

Further reading:

- Chapter 2 and 4 in *"Computer Organization and Design"*, D. Patterson and J. Hennessy, Morgan Kaufmann 2018 (RISC-V edition)
- Chapter 3 and 7 and Appendix A and C in *"Computer Architecture"*, J. Hennessy and D. Patterson, Morgan Kaufmann 2019
- Section 1.2 in *"Introduction to High Performance Scientific Computing"*, V. Eijkhout, free pdf 3rd edition 2020
- M. V. Wilkes and J. B. Stringer, *"Micro-Programming and the Design of the Control Circuits in an Electronic Digital Computer"*, Cambridge, 1952