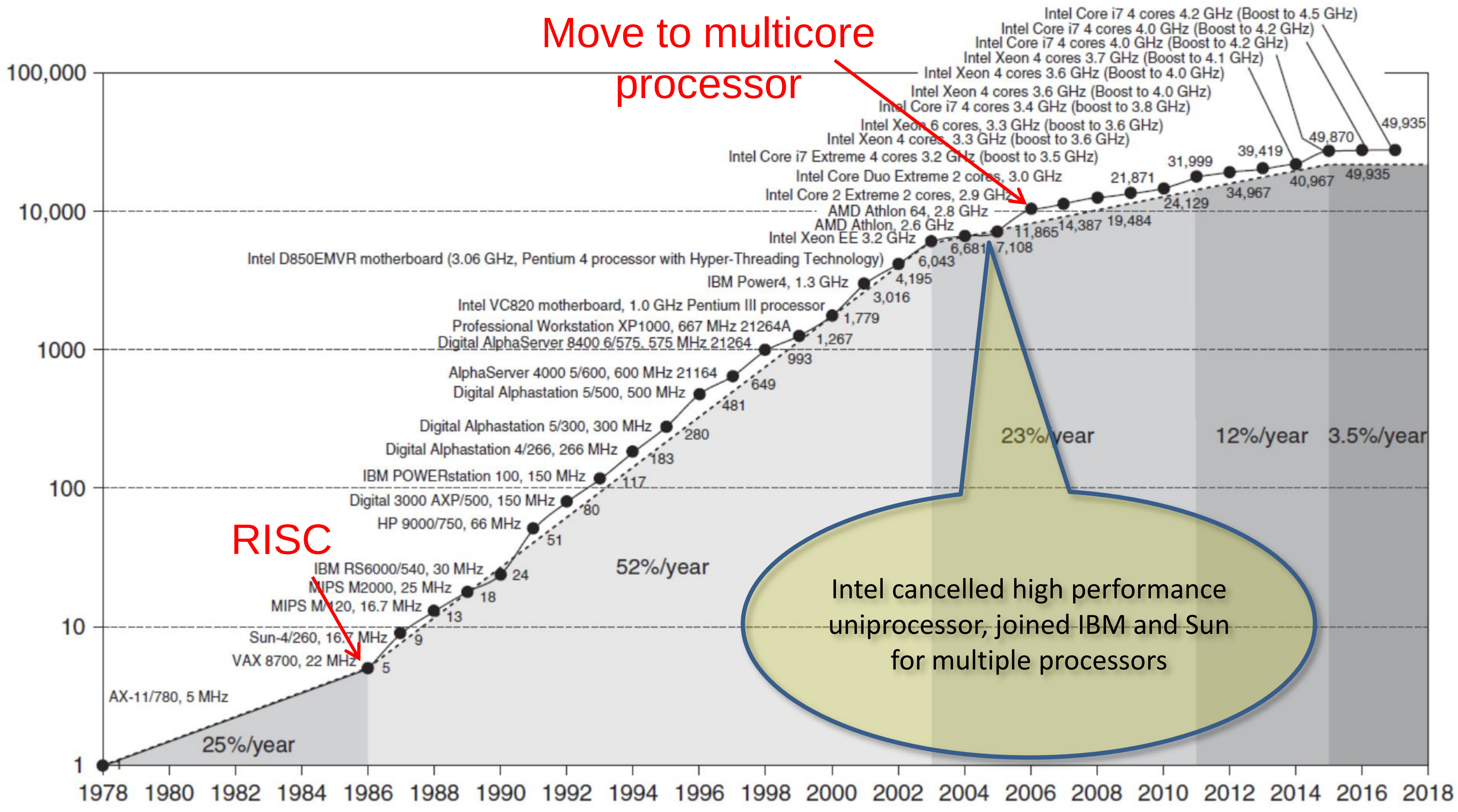
Computer Architecture
Lecture 1: Fundamentals of Quantitative Design and Analysis (Chapter 1)

Chih-Wei Liu 劉志尉

National Yang Ming Chiao Tung University

cwliu@twins.ee.nctu.edu.tw

Single Processor Performance



Computer Technology

- Performance improvements:
  - Improvements in semiconductor technology
    - Feature size, clock speed, power consumption, …
  - Improvements in computer architectures
    - The virtual elimination of assembly language programming
      - Enabled by high-level language (HLL) compilers
    - The standardized, vendor-independent operating systems
      - UNIX
    - Lead to RISC architectures
  - Together have enabled:
    - Lightweight, embedded computers
    - Productivity-based managed/interpreted programming languages

Current Trends in Architecture

- Cannot continue to leverage Instruction-Level parallelism (ILP)
  - Single processor performance improvement ended in 2003
  - Multiple processors or multiple cores is the current trend
- New models (explicitly parallel models) for performance:
  - Data-level parallelism (DLP)
  - Thread-level parallelism (TLP)
  - Request-level parallelism (RLP)
- These require explicit restructuring of the application

Classes of Computers

- Clusters / Warehouse Scale Computers
  - Used for “Software as a Service (SaaS).” Emphasis on availability and price-performance.
  - Supercomputers. Emphasis on floating-point performance and fast internal networks
- Servers
  - Emphasis on availability, scalability, and efficient throughput
- Desktop Computing
  - Emphasis on price-performance index
- Personal Mobile Device (PMD)
  - Emphasis on energy efficiency and real-time constraints
- Internet of Things (IT)/Embedded Computers
  - Have the widest spread of processing power and cost.
  - Emphasis on cost-price index (meets the performance need at a minimum price, rather than achieving more performance at a higher price.)

Parallelism?

- Classes of parallelism in applications:
  - Data-Level Parallelism (DLP)
  - Task-Level Parallelism (TLP)

- Classes of parallelism using in computer hardware:
  - OOO, speculative execution @ Instruction-Level Parallelism (ILP)
  - SIMD, vector architectures/GPUs @ Data-Level Parallelism
  - Tightly coupled multiple cores @ Thread-Level Parallelism
  - Loosely coupled multiple cores @ Request-Level Parallelism

Flynn’s Taxonomy for Parallelism

@1966

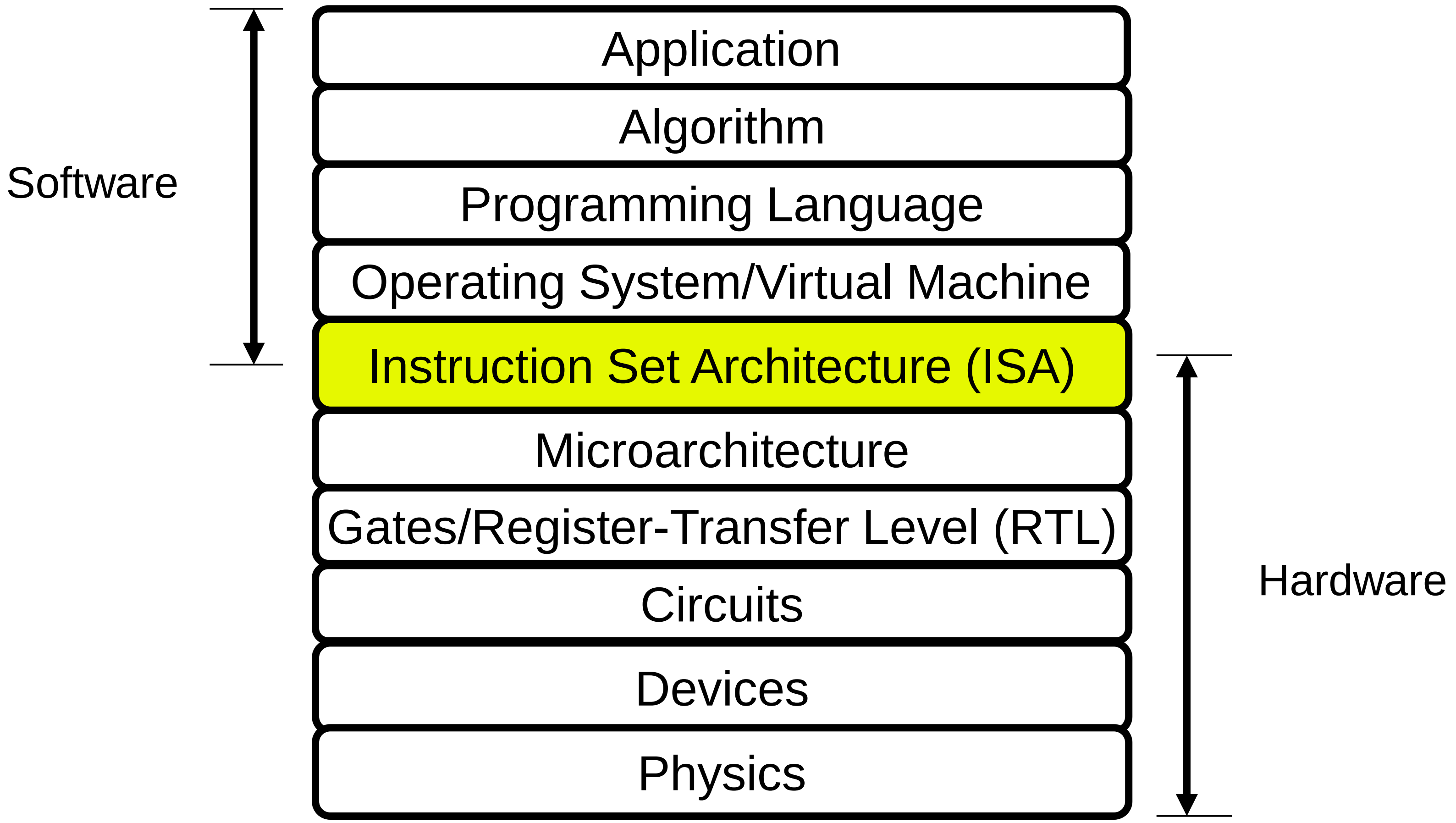
- Single instruction stream, single data stream (SISD)
- Single instruction stream, multiple data streams (SIMD)
  - Vector architectures
  - Multimedia extensions
  - Graphics processor units
- Multiple instruction streams, single data stream (MISD)
  - No commercial implementation
- Multiple instruction streams, multiple data streams (MIMD)
  - Tightly-coupled MIMD
  - Loosely-coupled MIMD

Defining Computer Architecture

- "Old" view of computer architecture:
  - Instruction Set Architecture (ISA) design, i.e. decisions regarding: registers, memory addressing, addressing modes, instruction operands, available operations, control flow instructions, instruction encoding

- "Real" computer architecture:
  - Specific requirements of the target machine, designed to maximize performance within constraints (cost, power, and availability)
  - Not only ISA, but also incudes microarchitecture, hardware, software, …

Computer System vs. ISA

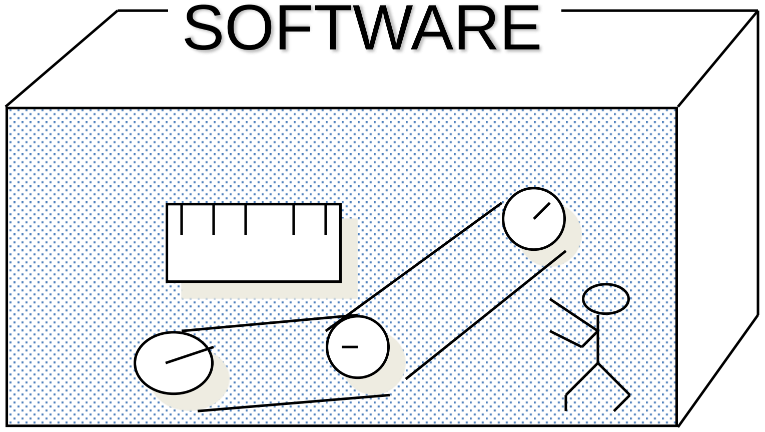

Software
Application
Algorithm
Programming Language
Operating System/Virtual Machine
Instruction Set Architecture (ISA)
Microarchitecture
Gates/Register-Transfer Level (RTL)
Circuits
Devices
Physics
Hardware

DEPT. OF ELECTRONICS ENGINEERING & INST. OF ELECTRONICS NCTU

Instruction Set Architecture, ISA

"... the attributes of a [computing] system as seen by the programmer, *i.e.* the conceptual structure and functional behavior, as distinct from the organization of the data flows and controls the logic design, and the physical implementation."
– Amdahl, Blaauw, and Brooks, 1964

- Organization of Programmable Storage
- Data Types & Data Structures:
  Encodings & Representations
- Instruction Formats
- Instruction (or Operation Code) Set
- Modes of Addressing and Accessing Data Items and Instructions
- Exceptional Conditions



ISA Design Issue

- Where are operands stored?
- How many explicit operands are there?
- How is the operand location specified?
- What type & size of operands are supported?
- What operations are supported?

Before answering these questions, let's consider more about

- Memory addressing
- Data operand
- Operations

Class of ISA

- RISC vs. CISC (code density issue)
- Almost all ISAs today are classified as general-purpose register (GPR) architectures
  - The operands are either registers or memory locations
  - Register-memory ISAs
    - Part of instructions can access memory, i.e. one of the operands can be memory location
    - E.g., 80x86
  - Load-store ISA
    - Only load and store instructions can access memory
    - All ISAs announced since 1985 are load-store. E.g., MIPS, ARM, RISC-V

RISC-V Registers

- 32 general-purpose registers and **32 floating-point registers**
  - Name, usage, and calling conventions

Register	Name	Use	Saver
x0	zero	constant 0	n/a
x1	ra	return addr	caller
x2	sp	stack ptr	callee
x3	gp	gbl ptr	
x4	tp	thread ptr	
x5-x7	t0-t2	temporaries	caller
x8	s0/fp	saved/ frame ptr	callee

Register	Name	Use	Saver
x9	s1	saved	callee
x10-x17	a0-a7	arguments	caller
x18-x27	s2-s11	saved	callee
x28-x31	t3-t6	temporaries	caller
f0-f7	**ft0-ft7**	**FP temps**	**caller**
f8-f9	**fs0-fs1**	**FP saved**	**callee**
f10-f17	**fa0-fa7**	**FP arguments**	**callee**
f18-f27	**fs2-fs21**	**FP saved**	**callee**
f28-f31	**ft8-ft11**	**FP temps**	**caller**

The registers that are preserved across a procedure call are labeled "Callee" saved.

Memory Addressing

- Most CPUs are byte-addressable and provide access for
  - Byte (8-bit)
  - Half word (16-bit)
  - Word (32-bit)
  - Double words (64-bit)

- How memory addresses are interpreted and how they are specified?
  - Little Endian or Big Endian
    - for ordering the bytes within a larger object within memory
  - Alignment or misaligned memory access
    - for accessing to an abject larger than a byte from memory
  - Addressing modes
    - for specifying constants, registers, and locations in memory

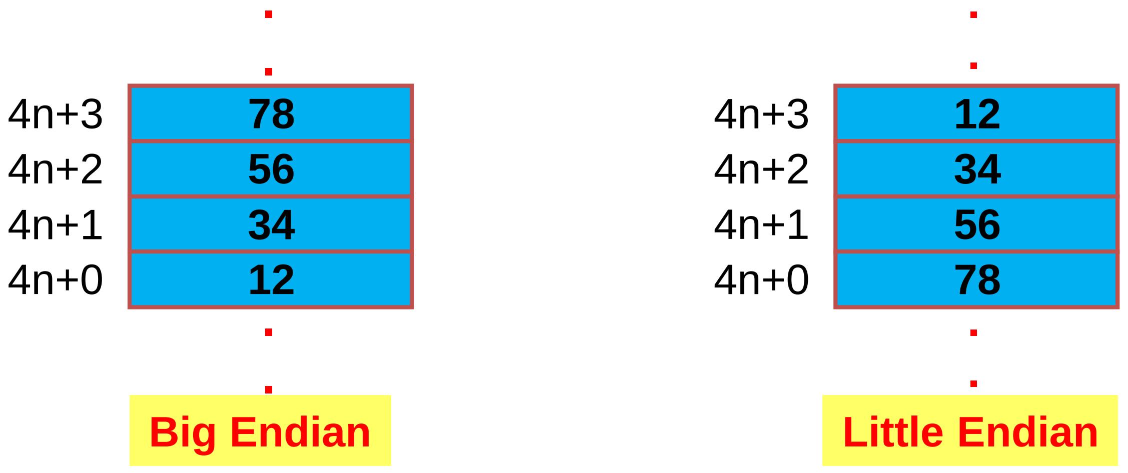
DEPT. OF ELECTRONICS ENGINEERING & INST. OF ELECTRONICS NCTU

Little or Big Endian ?

- No absolute advantage for one over the other, but

  Byte order is a problem when exchanging data among computers

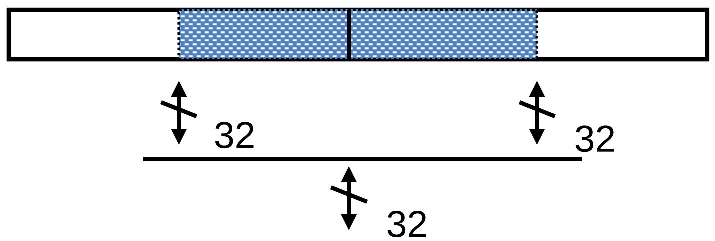
- Example
  - In C, **`int num = 0x12345678; // a 32-bit word`**,
  - how is `num` stored in memory?



Alignment Data Access

- The memory is typically aligned on a word or double-word boundary.
- An access to object of size S bytes at byte address A is called aligned if $A \bmod S = 0$.
- Access to an unaligned operand may require more memory accesses !!

Mis-aligned word reference



To Processor

Remarks

- ARM requires alignment access, while 80x86 and RISC-V do not require alignment.
- Unrestricted Alignment
  - Software is simple
  - Hardware must detect misalignment and make more memory accesses
  - Expensive logic to perform detection
  - Can slow down all references
  - Sometimes required for backwards compatibility
- Restricted Alignment
  - Software must guarantee alignment
  - Hardware detects misalignment access and traps
  - No extra time is spent when data is aligned

- Since we want to ***make the common case fast***, having restricted alignment is often a better choice, unless compatibility is an issue.

Addressing Mode ?

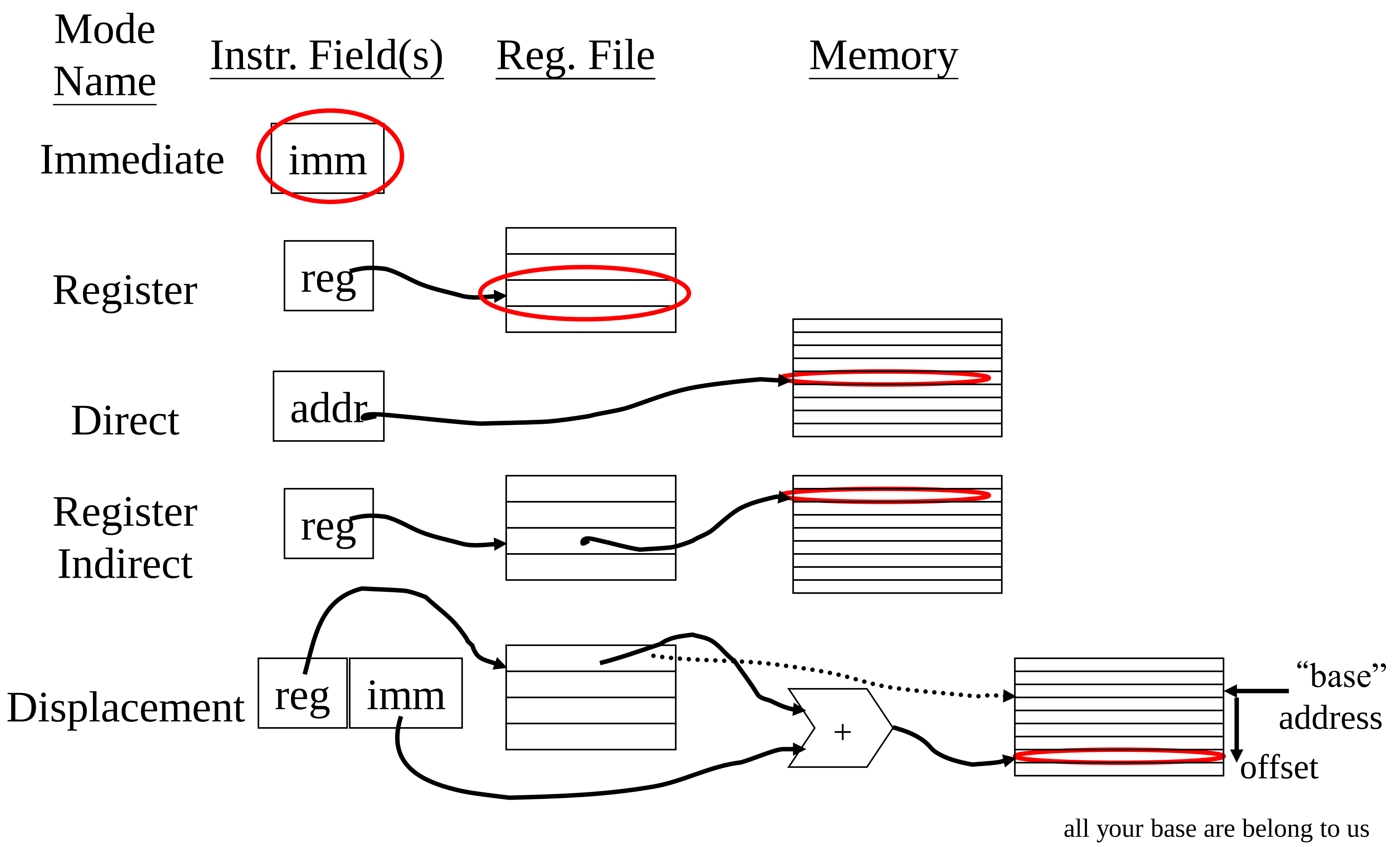
- It answers the question:
  - Where can operands/results be located?
- Address modes are used to specify registers and the address of a memory object
- Recall that we have two types of storage in computer : registers and memory
  - A single operand can come from either a register or a memory location
  - Addressing modes offer various ways of specifying the specific location

Addressing Mode Example

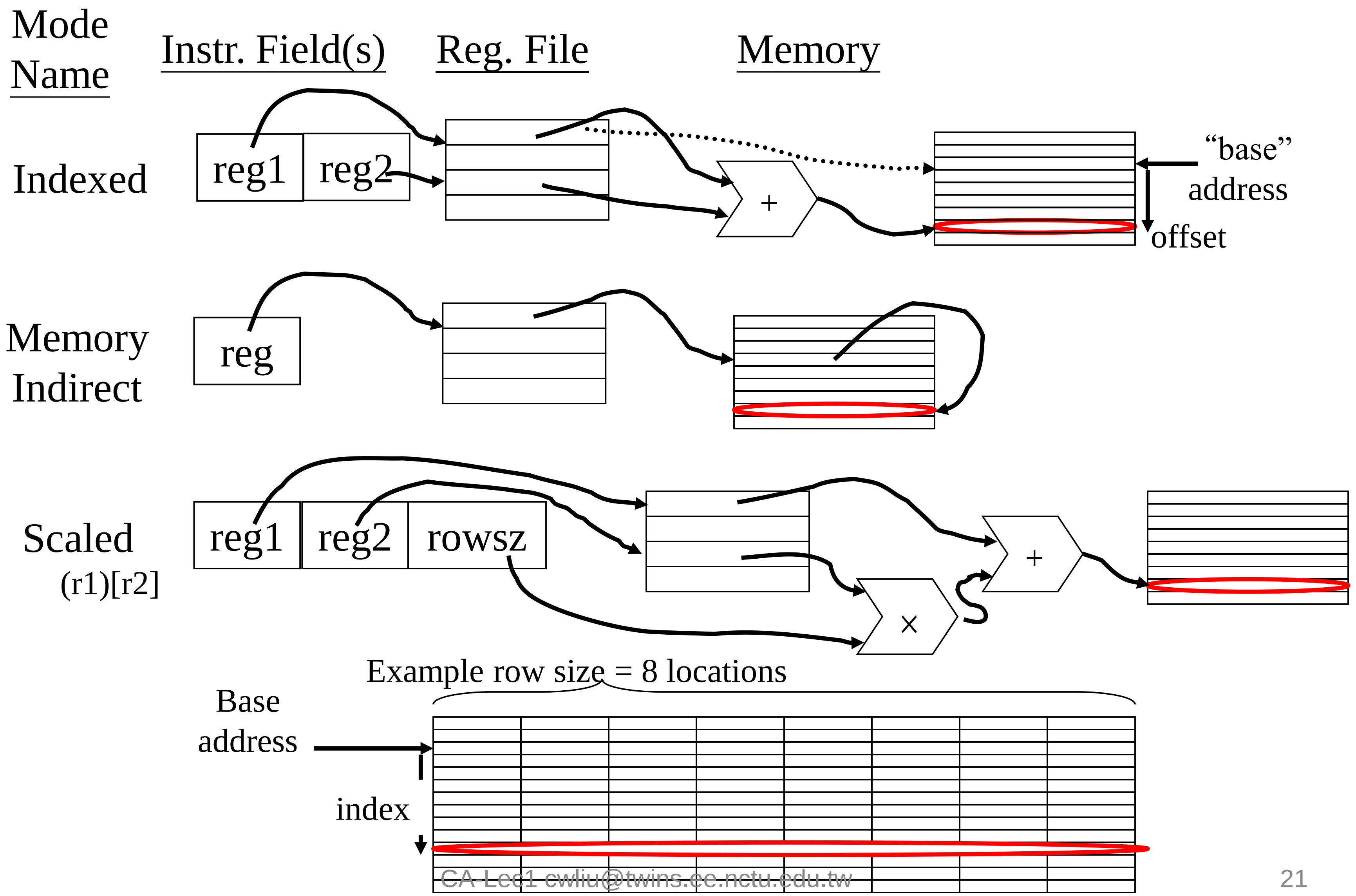
Addressing Mode	Example	Action
1. Register direct	`Add R1, R2, R3`	`R1 <- R2 + R3`
2. Immediate	`Add R1, R2, #3`	`R1 <- R2 + 3`
3. Register indirect	`Add R1, R2,(R3)`	`R1 <- R2 + M[R3]`
4. Displacement	`LD R1, 100(R2)`	`R1 <- M[100 + R2]`
5. Indexed	`LD R1, (R2 + R3)`	`R1 <- M[R2 + R3]`
6. Direct	`LD R1, (1000)`	`R1 <- M[1000]`
7. Memory Indirect	`Add R1, R2, @(R3)`	`R1 <- R2 + M[M[R3]]`
8. Auto-increment	`LD R1, (R2)+`	`R1 <- M[R2]`
		`R2 <- R2 + d`
9. Auto-decrement	`LD R1, (R2)-`	`R1 <- M[R2]`
		`R2 <- R2 – d`
10. Scaled	`LD R1, 100(R2)[R3]`	`R1 <- M[100+R2+R3*d]`

R: Register, M: Memory

Addressing Modes Visualization (1)



Addressing Modes Visualization (2)



How Many Addressing Mode ?

- A Tradeoff: **complexity vs. instruction count**
  - Should we add more modes?
    - Depends on the application class
    - Special addressing modes for DSP/GPU processors
      - Modulo or circular addressing
      - Bit reverse addressing
      - Stride, gather/scatter addressing
      - DSPs sometimes rely on hand-coded libraries to exercise novel addressing modes
- Need to support at least three types of addressing mode
  - **Displacement, immediate, and register**
    - A typical 12-bit displacement field
    - A typical of 12-bit immediate field

Type and Sizes of Operands

- 8-bit operand
  - Byte data, ASCII character
- 16-bit operand
  - Half-word data, Unicode character
- 32-bit operand
  - Integer or Word, IEEE 754 floating-point (FP) single precision
- 64-bit operand
  - Long integer or Double word, IEEE 754 FP double precision

Operations/Instructions in RISC

- RISC is a simple and easy-to-pipeline ISA
- Data transfers
  - Move data between registers and memory, or between integer and FP/special registers
- Arithmetic/logical
  - Operations on integer or logical data in GPRs
- Program flow control
  - Conditional branches
  - Unconditional jumps
  - Procedure call and return
    - MIPS, ARM, and RISC-V place the return address in a register (RA), while 80x86 place the return address in a stack in memory
- Predicated instructions
  - Whether the result of an arithmetic/logic operation is saved or ignored is depending on the condition code bits (e.g., ARM)

Instruction type/opcode	Instruction meaning
Data transfers	*Move data between registers and memory, or between the integer and FP or special registers; only memory address mode is 12-bit displacement + contents of a GPR*
`lb`, `lbu`, `sb`	Load byte, load byte unsigned, store byte (to/from integer registers)
`lh`, `lhu`, `sh`	Load half word, load half word unsigned, store half word (to/from integer registers)
`lw`, `lwu`, `sw`	Load word, load word unsigned, store word (to/from integer registers)
`ld`, `sd`	Load double word, store double word (to/from integer registers)
`flw`, `fld`, `fsw`, `fsd`	Load SP float, load DP float, store SP float, store DP float
`fmv._.x`, `fmv.x._`	Copy from/to integer register to/from floating-point register; "_" = S for single-precision, D for double-precision
`csrrw`, `csrrwi`, `csrrs`, `csrrsi`, `csrrc`, `csrrci`	Read counters and write status registers, which include counters: clock cycles, time, instructions retired
Arithmetic/logical	*Operations on integer or logical data in GPRs*
`add`, `addi`, `addw`, `addiw`	Add, add immediate (all immediates are 12 bits), add 32-bits only & sign-extend to 64 bits, add immediate 32-bits only
`sub`, `subw`	Subtract, subtract 32-bits only
`mul`, `mulw`, `mulh`, `mulhsu`, `mulhu`	Multiply, multiply 32-bits only, multiply upper half, multiply upper half signed-unsigned, multiply upper half unsigned
`div`, `divu`, `rem`, `remu`	Divide, divide unsigned, remainder, remainder unsigned
`divw`, `divuw`, `remw`, `remuw`	Divide and remainder: as previously, but divide only lower 32-bits, producing 32-bit sign-extended result
`and`, `andi`	And, and immediate
`or`, `ori`, `xor`, `xori`	Or, or immediate, exclusive or, exclusive or immediate
`lui`	Load upper immediate; loads bits 31-12 of register with immediate, then sign-extends
`auipc`	Adds immediate in bits 31–12 with zeros in lower bits to PC; used with `JALR` to transfer control to any 32-bit address
`sll`, `slli`, `srl`, `srli`, `sra`, `srai`	Shifts: shift left logical, right logical, right arithmetic; both variable and immediate forms
`sllw`, `slliw`, `srlw`, `srliw`, `sraw`, `sraiw`	Shifts: as previously, but shift lower 32-bits, producing 32-bit sign-extended result
`slt`, `slti`, `sltu`, `sltiu`	Set less than, set less than immediate, signed and unsigned
Control	*Conditional branches and jumps; PC-relative or through register*
`beq`, `bne`, `blt`, `bge`, `bltu`, `bgeu`	Branch GPR equal/not equal; less than; greater than or equal, signed and unsigned
`jal`, `jalr`	Jump and link: save PC + 4, target is PC-relative (`JAL`) or a register (`JALR`); if specify `x0` as destination register, then acts as a simple jump
`ecall`	Make a request to the supporting execution environment, which is usually an OS
`ebreak`	Debuggers used to cause control to be transferred back to a debugging environment
`fence`, `fence.i`	Synchronize threads to guarantee ordering of memory accesses; synchronize instructions and data for stores to instruction memory

Example: RV32I Instruction

```
lb  rd, imm[11:0](rs1)
lbu rd, imm[11:0](rs1)
lh  rd, imm[11:0](rs1)
lhu rd, imm[11:0](rs1)
lw  rd, imm[11:0](rs1)
sb  rs2, imm[11:0](rs1)
sh  rs2, imm[11:0](rs1)
sw  rs2, imm[11:0](rs1)
fence pred, succ
fence.i

add   rd, rs1, rs2
sub   rd, rs1, rs2
sll   rd, rs1, rs2
srl   rd, rs1, rs2
sra   rd, rs1, rs2
and   rd, rs1, rs2
or    rd, rs1, rs2
xor   rd, rs1, rs2
slt   rd, rs1, rs2
sltu  rd, rs1, rs2
addi  rd, rs1, imm[11:0]
slli  rd, rs1, shamt[4:0]
srli  rd, rs1, shamt[4:0]
srai  rd, rs1, shamt[4:0]
andi  rd, rs1, imm[11:0]
ori   rd, rs1, imm[11:0]
xori  rd, rs1, imm[11:0]
slti  rd, rs1, imm[11:0]
sltiu rd, rs1, imm[11:0]
lui   rd, imm[31:12]
auipc rd, imm[31:12]

beq  rs1, rs2, imm[12:1]
bne  rs1, rs2, imm[12:1]
blt  rs1, rs2, imm[12:1]
bltu rs1, rs2, imm[12:1]
bge  rs1, rs2, imm[12:1]
bgeu rs1, rs2, imm[12:1]
jal  rd, imm[20:1]
jalr rd, rs1, imm[11:0]
```

Appendices A and K

Procedure Calling

$$\begin{tikzcd} \textbf{Caller} \arrow[r, bend left] & \textbf{Callee} \end{tikzcd}$$

- Steps required
  1. Place parameters in registers
  2. Transfer control to procedure
  3. Acquire storage for procedure
  4. Perform procedure’s operations
  5. Place result in register for caller
  6. Return to place of call

Note that you have only one set of registers !!

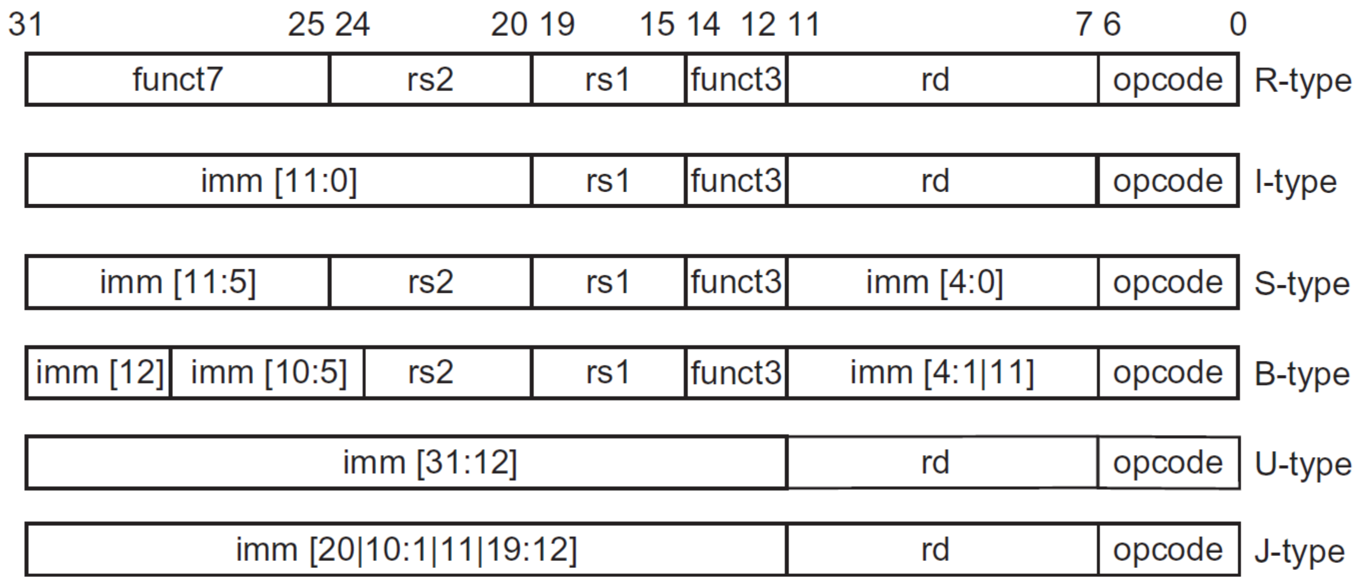
Predicated/Conditional Instructions

- ARM uses (N, Z, C, O) condition codes for result of an arithmetic/logical instruction
  - Top 4 bits of instruction word stores 4 condition value : Negative, Zero, Carry, Overflow
  - Compare instructions are used to set condition codes without keeping the result
- Almost all instructions can be conditional
  - Can avoid branches over single instructions

Encoding an ISA

- Fixed-length vs. Variable-length encoding
  - Code size (code density) issue
  - MIPS, ARMv8, and RISC-V are 32-bit fixed-length encoding
  - 80x86 is a variable-length (1~18 bytes) encoding
- Compressed encoding
  - For embedded system with a small on-chip memory
  - ARMv8/RISC-V 16-bit Thumb encoding
    - Thumb encoding ? (Appendix K)

Base RISC-V Instruction Format



31 25 24 20 19 15 14 12 11 7 6 0
funct7 | rs2 | rs1 | funct3 | rd | opcode | R-type
imm [11:0] | rs1 | funct3 | rd | opcode | I-type
imm [11:5] | rs2 | rs1 | funct3 | imm [4:0] | opcode | S-type
imm [12] | imm [10:5] | rs2 | rs1 | funct3 | imm [4:1|11] | opcode | B-type
imm [31:12] | rd | opcode | U-type
imm [20|10:1|11|19:12] | rd | opcode | J-type

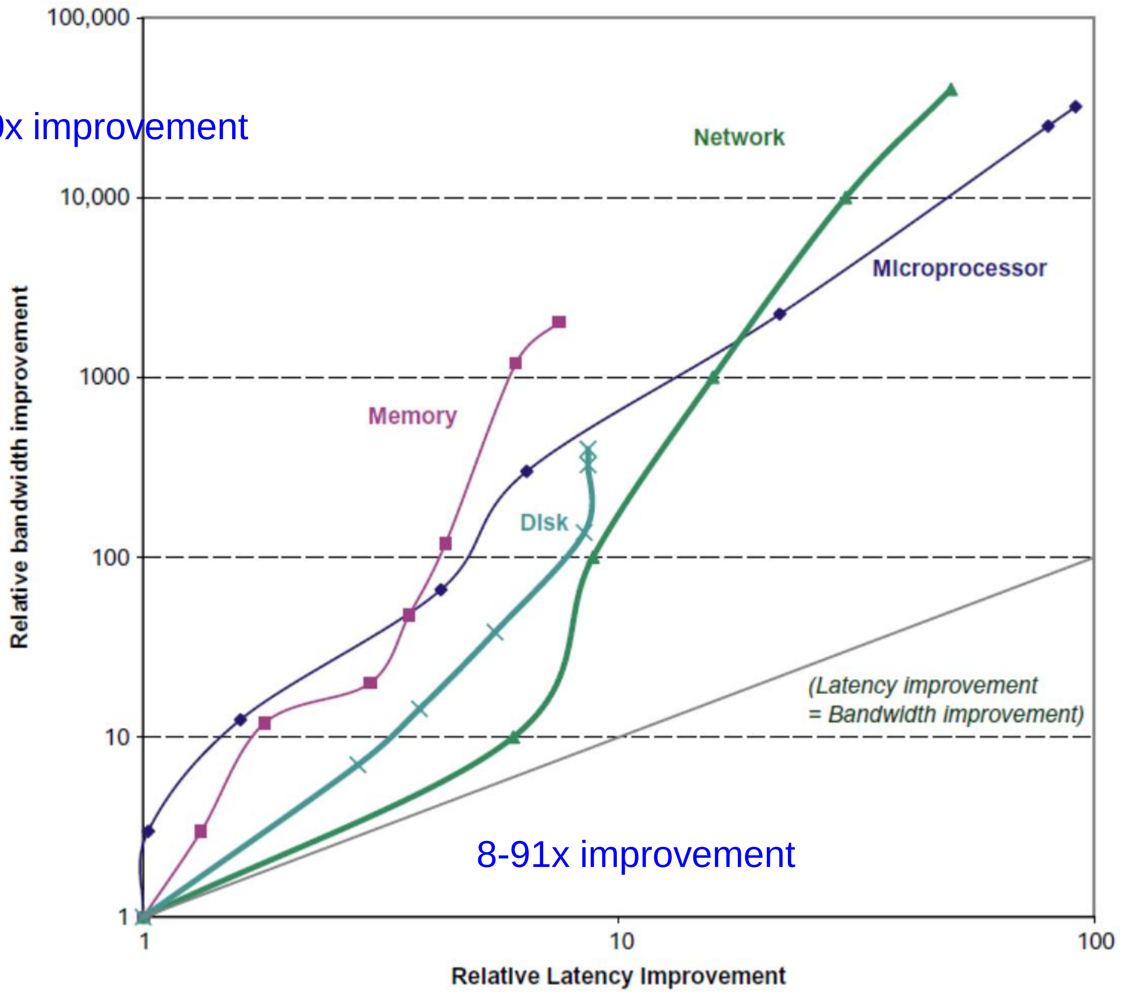
Figure 1.7 The base RISC-V instruction set architecture formats. All instructions are 32 bits long. The R format is for integer register-to-register operations, such as ADD, SUB, and so on. The I format is for loads and immediate operations, such as LD and ADDI. The B format is for branches and the J format is for jumps and link. The S format is for stores. Having a separate format for stores allows the three register specifiers (rd, rs1, rs2) to always be in the same location in all formats. The U format is for the wide immediate instructions (LUI, AUIPC).

Trends in Technology

- Integrated circuit technology: ~doubling every 18-24 months
  - Transistor density: 35%/year
  - Die size: 10-20%/year
  - Integration overall: 40-55%/year
- DRAM capacity: 25-40%/year (slowing)
  - 8 Gb (2014), 16 Gb (2019), possibly no 32 Gb
- Flash capacity: 50-60%/year
  - 8-10X cheaper/bit than DRAM
- Magnetic disk technology: recently slowed to 5%/year
  - Density increases may no longer be possible, maybe increase from 7 to 9 platters
  - 8-10X cheaper/bit then Flash
  - 200-300X cheaper/bit than DRAM
- Network technology: Appendix F

Bandwidth and Latency

400-32000x improvement



8-91x improvement

Log-log plot of bandwidth and latency milestones

Bandwidth and Latency

- Bandwidth or throughput
  - Total work done in a given time
  - 32,000-40,000X improvement for processors
  - 300-1200X improvement for memory and disks

- Latency or response time
  - Time between start and completion of an event
  - 50-90X improvement for processors
  - 6-8X improvement for memory and disks

Transistors and Wires

- Feature size
  - Minimum size of transistor or wire in x or y dimension
  - 10 microns in 1971 to .011 microns in 2017
  - Transistor performance scales linearly
    - Wire delay does not improve with feature size! Because the resistance and capacitance per unit length get worse.
  - Integration density scales quadratically
- As feature sizes shrink, devices shrink quadratically both in the horizontal dimension and in the vertical dimension

DEPT. OF ELECTRONICS ENGINEERING & INST. OF ELECTRONICS NCTU

Why Bandwidth Over Latency?

- Moore’s law helps BW more than latency
  - Faster transistors, more transistors, and more pin count help bandwidth
  - Smaller transistors communicate over (relatively) longer wires
- Latency helps BW, but not vice versa
  - Lower DRAM latency, i.e. more accesses per second, implies higher bandwidth
  - Higher density of disk helps BW and capacity, but not disk latency
- BW hurts latency
  - Queues or buffers help bandwidth, but hurt latency
  - Memory bank help bandwidth with widen the memory module, but not latency due to higher fan-out on address lines
- OS overhead hurts latency more than BW
  - Packet header overhead: bigger part of short message

Summary of Technology Trends

- A simple rule of thumb: bandwidth improves by at least the square of the improvement in latency.
  - In the time that bandwidth doubles, latency improves by no more than 1.2X to 1.4X
- Lag probably even larger in real systems, as bandwidth gains multiplied by replicated components
  - Multiple processors in a cluster or even in a chip
  - Multiple disks in a disk array
  - Multiple memory modules in a large memory
  - Simultaneous communication in switched LAN
- HW and SW developers should innovate assuming Latency Lags Bandwidth
  - If everything improves at the same rate, then nothing really changes
  - When rates vary, require real innovation

Define and Quantity Power and Energy

- Problem: Get power in, get power out
  1. (Power supply) What is the maximum power a processor ever requires?
  2. (Cooling system) What is the sustained power consumption?
  3. Low power consumption design or High energy efficiency design?

- **Thermal Design Power (TDP)**
  - Characterizes sustained power consumption
  - Used as target for power supply and cooling system
  - Lower than peak power, higher than average power consumption

- Power vs. Energy?
  - Clock rate can be reduced dynamically to limit power consumption
  - Energy per task is often a better metric
  - Designers and users need to consider is energy and energy efficiency

Power and Energy (1/2)

- For CMOS chips, traditional dominant power consumption has been in switching transistors, called dynamic power

$$P_{dynamic} \propto \frac{1}{2} \times C_{load} \times V^2 \times F$$

- For mobile devices, dynamic energy, instead of power, is the proper metric
  - Transistor switch from 0 -> 1 or 1 -> 0

$$E_{dynamic} \propto C \times V^2$$

- For a fixed task, slowing clock rate (frequency switched) reduces power, but not energy
- **Dropping voltage helps both**, so went from 5V to 1V
- As moved from one process to the next, the increase in the number of transistors switching, the frequency, dominates the decrease in load capacitance and voltage

➔ an overall growth in power consumption and energy

- To save energy & dynamic power,
  - most CPUs now turn off clock of inactive modules (e.g. Fl. Pt. Unit)
  - some CPUs are designed to have adjustable voltage

DVFS Power/Energy Saving Example

- Some microprocessors today are designed to have adjustable voltage. Assume a 15% reduction in voltage will result in a 15% reduction in frequency, what would be the impact on dynamic energy and dynamic power?

- Because the capacitance is unchanged, the answer for energy is

$$\frac{\text{Energy}_{\text{new}}}{\text{Energy}_{\text{old}}} = \frac{(Voltage \times 0.85)^2}{Voltage^2} = 0.72$$

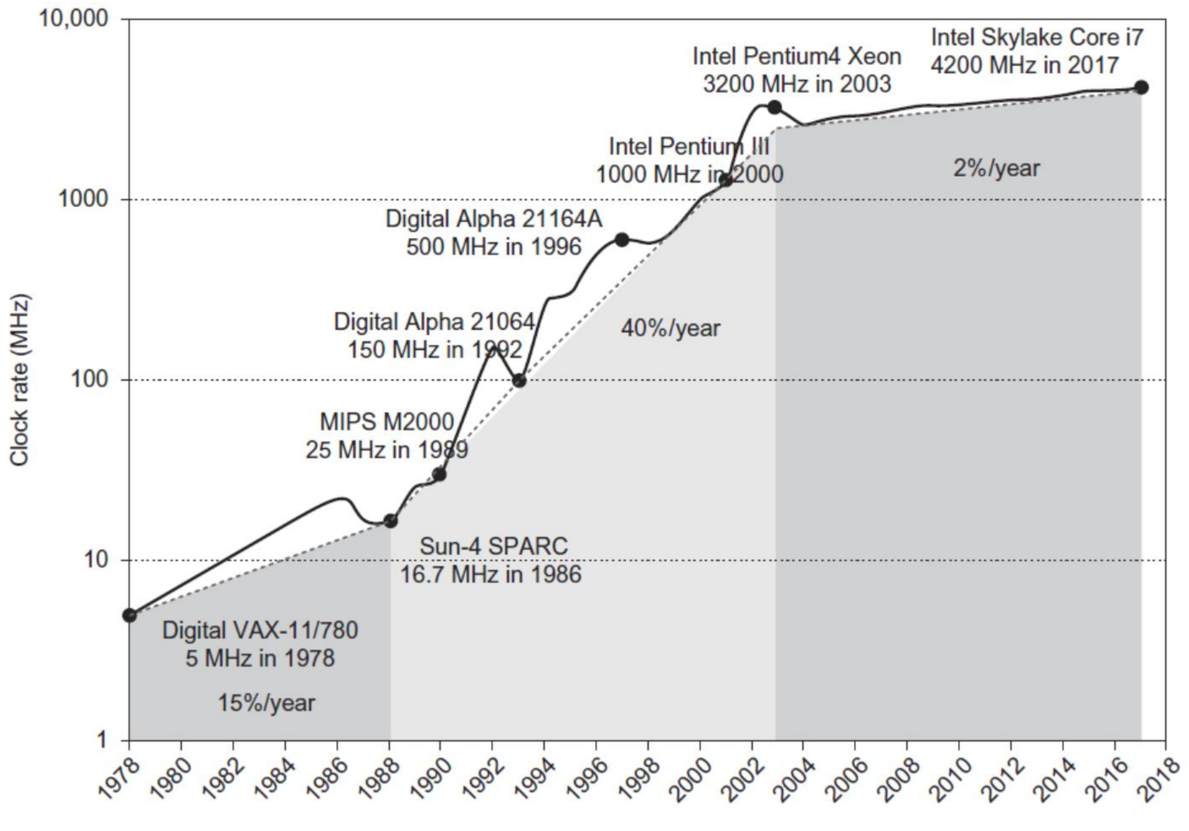
- For power, the answer is

$$\frac{\text{Power}_{\text{new}}}{\text{Power}_{\text{old}}} = \frac{(Voltage \times 0.85)^2 \times (Frequency \times 0.85)}{Voltage^2 \times Frequency} = 0.61$$

Lower than energy !!

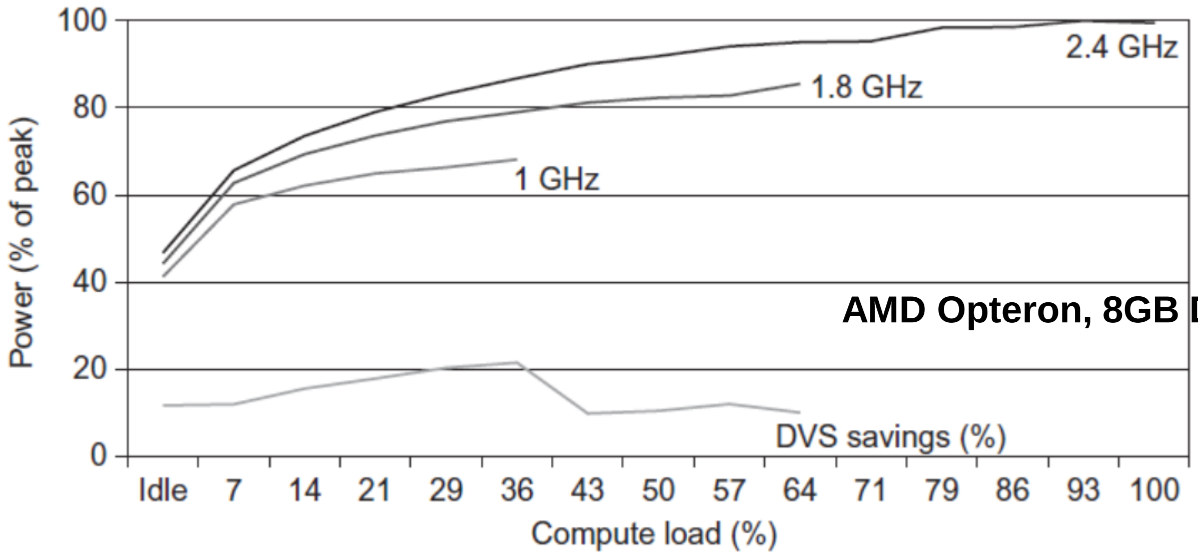
Clock Rate and Power

- Intel 80386 consumed ~ 2 W
- 4.0 GHz Intel Core i7 consumes 95W
- Heat must be dissipated from 1.5 x 1.5 cm^2 chip
- This is the limit of what can be cooled by air



Techniques for Reducing Power

1. *Clock gating*: Just turn off the clock of inactive modules (clock gating)
2. Dynamic Voltage-Frequency Scaling (*DVFS*)



AMD Opteron, 8GB DRAM, one ATA disk

3. *System power control*: sleep, Idle, Low power, and normal modes.
4. Overclocking one while turning off the other cores (Intel *Turbo mode*)

Power and Energy (2/2)

- Because leakage current flows even when a transistor is off, now *static power* is important too

$$P_{static} \propto I_{static} \times V$$

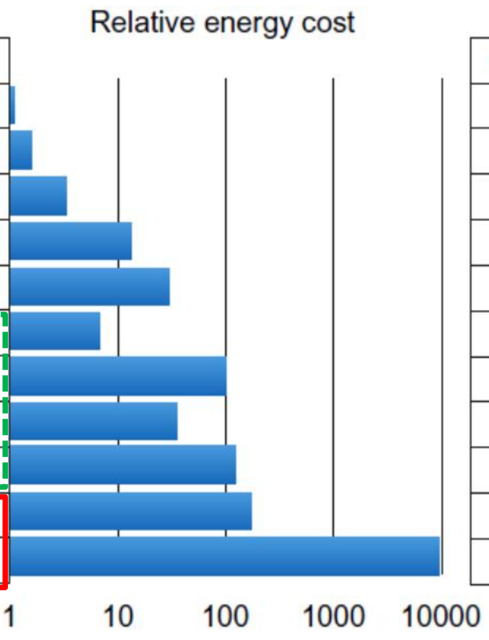
- Increasing the number of transistors increases static power even if they are turned off
- Leakage current increases in processors with smaller transistor sizes
- Static power is becoming an important issue
  - 25%-50% (or more) of total power is static
- Very low power systems even ***gate voltage*** to inactive modules (i.e. power gating) to control loss due to leakage

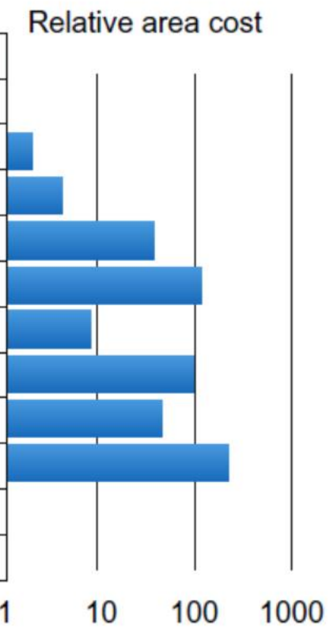
Energy Cost and Die Area Cost Example

[ISSCC 2014]

- @ TSMC 45nm CMOS technology
- FP units are used by DesignWare Library
- **New design directions?** By reducing wide FP operations and deploying special-purpose memories to reduce access to DRAM.

Operation:	Energy (pJ)	Area (μm^2)
8b Add	0.03	36
16b Add	0.05	67
32b Add	0.1	137
16b FB Add	0.4	1360
32b FB Add	0.9	4184
8b Mult	0.2	282
32b Mult	3.1	3495
16b FB Mult	1.1	1640
32b FB Mult	3.7	7700
32b SRAM Read (8KB)	5	N/A
32b DRAM Read	640	N/A





Energy numbers are from Mark Horowitz *Computing's Energy problem (and what we can do about it)*. ISSCC 2014
Area numbers are from synthesized result using Design compiler under TSMC 45nm tech node. FP units used DesignWare Library.

Trends in Cost

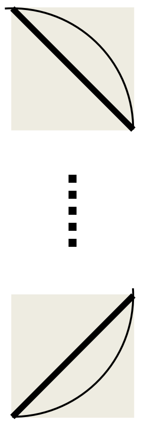
- Price: what you sell a finished good for
- Cost: amount spent to produce it, including overhead
  - Price and cost of DRAM track closely.
- The impact of time, volume, and commodification
  - Learning curve: manufacturing costs decrease over time (max. measured by yield)
  - Volume decreases the cost
    - Costs decrease about 10% for each doubling of volumn
  - Commodities: sell on the grocery stores, multiple suppliers.

Cost of an IC

- A wafer is tested and chopped into dies

$$C_{\text{die}} = \frac{C_{\text{wafer}}}{\text{Die per wafer} \times \text{Die yield}}$$

$$\text{Die per wafer} \approx \frac{\pi \times (\text{Wafer diameter}/2)^2}{\text{Die area}} - \frac{\pi \times \text{Wafer diameter}}{\sqrt{2} \times \sqrt{\text{Die area}}}$$



- The die is still tested and packaged into IC

$$C_{\text{IC}} = \frac{C_{\text{die}} + C_{\text{testing die}} + C_{\text{packaging and final test}}}{\text{Final test yield}}$$

IC Yield

- Bose-Einstein formula:

$$\text{Die yield} = \text{Wafer yield} \times 1 \big/ \left(1 + \text{Defects per unit area} \times \text{Die area}\right)^{N}$$

  - Wafer yield is almost 100%
  - Defects per unit area = 0.012—0.016 defects per cm^2 (28nm, 2017)
  - Defects per unit area = 0.016—0.047 defects per cm^2 (16nm, 2017)
  - N = process-complexity factor = 7.5—9.5.5 (28nm, 2017) and 10—14 (16nm, 2017)
- Example: Find the die yield for dies that are 1.5 cm on a side. Assuming a 300mm wafer and a defect density of 0.047/cm^2 and *N* is 12.

$$\text{Dies per wafer} = \frac{\pi \times (30/2)^2}{2.25} - \frac{\pi \times 30}{\sqrt{2 \times 2.25}} = 270$$

$$\text{Die yield} = 1 \big/ \left(1 + 0.047 \times 2.25\right)^{12} \times 270 = 120$$

Dependability?

- Old CW: ICs are reliable components
- New CW: On the transistor feature size down to 65nm or smaller, both transient faults and permanent faults will become more common place.
- Define and quantity service level agreement (SLA) or dependability
  - Service accomplishment vs. Service interruption
  - 2 states of finite state machine: Failure and Restoration
- How to quantify module/system reliability?
- How to quantify module/system availability?

Module Reliability and Availability

- 2-state Markov chain for service accomplishment and service interruption
  - Module reliability = measure of continuous service accomplishment.
  - Module availability = measure of the service accomplishment with respect to the alternation between the 2 states.
- ***Mean Time To Failure*** (MTTF) measures reliability
  - Failures In Time (FIT), failures per billion hours of operations, i.e. the rate of failures.
  - FIT = 1/MTTF
- ***Mean Time To Repair*** (MTTR) measures service interruption
- Module availability = MTTF / (MTTF + MTTR)
  - Mean Time Between Failures (MTBF) = MTTF+MTTR

Dependability Example

- Assume a disk subsystem with the following components and MTTF:
  - 10 disks, each rated at 1000000-hour MTTF
  - 1 ATA controller, 500000-hour MTTF
  - 1 power supply, 200000-hour MTTF
  - 1 fan, 200000-hour MTTF
  - 1 ATA cable, 1000000-hour MTTF
- Suppose the *lifetimes are exponentially distributed* and *failures are independent*
- Answer:

$$\text{Failure rate} = \frac{10}{1000000} + \frac{1}{500000} + \frac{1}{200000} + \frac{1}{200000} + \frac{1}{1000000}$$

$$MTTF = \frac{1}{FIT} = \frac{1000000 \text{ hours}}{23} \approx 43500 \text{ hours}$$

Measuring Performance

- Typical performance metrics:
  - Response time
  - Throughput
- Execution time
  - Wall clock time: includes all system overheads
  - CPU time: only computation time
- Benchmarks
  - Kernels (e.g. matrix multiply)
  - Toy programs (e.g. sorting)
  - Synthetic benchmarks (e.g. Dhrystone)
  - Benchmark suites (e.g. SPEC CPU 2017, TPC-C)

- Speedup of X relative to Y = Execution time$_Y$ / Execution time$_X$

CPU Time Equation

$$\text{CPU time} = \text{Instruction count} \times \text{Cycles per instruction} \times \text{Cycle time}$$

$$= \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Clock cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Clock cycle}}$$

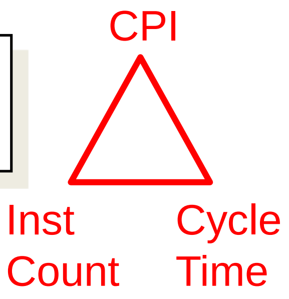
$$= \frac{\text{Seconds}}{\text{Program}}$$

It is difficult to change one parameter in complete isolation from others!!

- Instruction set architecture and compiler technology
- Organization and instruction set architecture
- Hardware technology and organization

Aspects of CPU Performance (CPU Law)

$$\text{CPU time} = \frac{\text{Seconds}}{\text{Program}} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Cycle}}$$



	IC	CPI	Clock Rate
Program	X		
Compiler	X	(X)	
Inst. Set.	X	X	
Organization		X	X
Technology			X

Reporting Performance Results

- Two different machines X and Y.

  X is *n* times faster than Y

$$\frac{\text{Execution time}_Y}{\text{Execution time}_X} = n$$

Since execution time is the reciprocal of performance

$$\frac{\text{Execution time}_Y}{\text{Execution time}_X} = n = \frac{\text{Performance}_X}{\text{Performance}_Y}$$

- Says n -1 = m/100

  This concludes that X is m% faster than Y

CPI and CPU Time

Different instruction types having different CPIs

$$\text{CPU clock cycles} = \sum_{i=1}^{n} \text{CPI}_i \times \text{IC}_i$$

Throughput $\searrow$

CPI_i = average number of clock cycles for instruction-i

IC_i = number of time the instruction-i is executed in a program

$$\text{CPU time} = \left(\sum_{i=1}^{n} \text{CPI}_i \times \text{IC}_i\right) \times \text{clock cycle time}$$

$$\text{CPI} = \frac{\sum_{i=1}^{n} \text{CPI}_i \times \text{IC}_i}{\text{Instruction count}} = \sum_{i=1}^{n} \text{CPI}_i \times \frac{\text{IC}_i}{\text{Instruction count}}$$

Example

We have the following measurements:

- Freq. of FP operation (other than FPSQR) = 25%
- Average CPI of FP operation = 4
- Average CPI of other instructions = 1.33
- Freq. of FPSQR = 2%
- CPI of FPSQR = 20

Assume we have 2 design alternatives

1. CPI of FPSQR: 20 $\rightarrow$ 2 , 10 times improve
2. CPI of FP operations: 4 $\rightarrow$ 2.5, 1.6 times improve

Answer: (Only CPI changes, clock rate, instruction count remains identical)

$$CPI_{original} = \sum_{i=1}^{n} CPI_i \times (IC_i / \text{Instruction count}) = 4 \times 0.25 + 1.33 \times 0.75 = 2.0$$

$$\begin{aligned} CPI_{new\,FP\,SQR} &= CPI_{original} - 2\%\,(CPI_{old\,FP\,SQR} - CPI_{new\,FP\,SQR\,only}) \\ &= 2.0 - 2\%\,(20 - 2) = 1.64 \end{aligned}$$

$$CPI_{new\,FP} = \sum_{i=1}^{n} CPI_i \times (IC_i / \text{Instruction count}) = 2.5 \times 0.25 + 1.33 \times 0.75 = 1.625$$

Better !!

Summarizing Performance Results

- Which benchmark(s)?
- Performance results should be reproducibility
  - Describe exactly the software system being measured and whether any special modifications have been made.
- ***Arithmetic average*** of execution time of all programs?
  - Could add a weight per program?
  - How pick weight? Different company wants different weights for their product
- Normalized execution time with respect to a reference computer
  - Time on reference computer / time on computer being rated
  - SPEC uses SPECRatio to compare performance
- ***Geometric mean*** of the SPECRatio of all programs?

Arithmetic Mean vs. Geometric Mean

- Arithmetic mean of time $\frac{1}{n}\sum_{i=1}^{n}\mathrm{Time}_i$
  - Time_i is the execution time for the ith program in the workload
- Weighted arithmetic mean $\sum_{i=1}^{n}\mathrm{Weight}_i \times \mathrm{Time}_i$
  - Weight_i factors add up to 1
- Geometric mean of ratio $\sqrt[n]{\prod_{i=1}^{n}\text{Execution time ratio}_i}$
  - Execution time ratio_i is the execution time normalized to the reference machine, for the ith program

Ratio of SPECRatio

$$\text{e.g. } 1.25 = \frac{SPECRatio_A}{SPECRatio_B} = \frac{\dfrac{ExecutionTime_{reference}}{ExecutionTime_A}}{\dfrac{ExecutionTime_{reference}}{ExecutionTime_B}}$$

$$= \frac{ExecutionTime_B}{ExecutionTime_A} = \frac{Performance_A}{Performance_B}$$

- SPECRatio is just a ratio rather than an absolute execution time
- Note that when comparing 2 computers as a ratio, execution times on the reference computer drop out, so choice of reference computer is irrelevant

Geometric mean of the ratios is the ratio of geometric means

- Choice of reference computer is irrelevant.

$$\frac{Geometric\ Mean_A}{Geometric\ Mean_B} = \frac{\sqrt[n]{\prod_{i=1}^{n} SPECRatioA_i}}{\sqrt[n]{\prod_{i=1}^{n} SPECRatioB_i}} = \sqrt[n]{\prod_{i=1}^{n} \frac{SPECRatioA_i}{SPECRatioB_i}}$$

$$= \sqrt[n]{\prod_{i=1}^{n} \frac{ExecutionTimeB_i}{ExecutionTimeA_i}} = \sqrt[n]{\prod_{i=1}^{n} \frac{PerformanceA_i}{PerformanceB_i}}$$

- Geometric mean does not predict execution time

Example

Benchmarks	Sun Ultra Enterprise 2 time (seconds)	AMD A10-6800K time (seconds)	SPEC 2006Cint ratio	Intel Xeon E5-2690 time (seconds)	SPEC 2006Cint ratio	AMD/Intel times (seconds)	Intel/AMD SPEC ratios
perlbench	9770	401	24.36	261	37.43	1.54	1.54
bzip2	9650	505	19.11	422	22.87	1.20	1.20
gcc	8050	490	16.43	227	35.46	2.16	2.16
mcf	9120	249	36.63	153	59.61	1.63	1.63
gobmk	10,490	418	25.10	382	27.46	1.09	1.09
hmmer	9330	182	51.26	120	77.75	1.52	1.52
sjeng	12,100	517	23.40	383	31.59	1.35	1.35
libquantum	20,720	84	246.08	3	7295.77	29.65	29.65
h264ref	22,130	611	36.22	425	52.07	1.44	1.44
omnetpp	6250	313	19.97	153	40.85	2.05	2.05
astar	7020	303	23.17	209	33.59	1.45	1.45
xalancbmk	6900	215	32.09	98	70.41	2.19	2.19
Geometric mean			31.91		63.72	2.00	2.00

Principles of Computer Design

1. Take Advantage of Parallelism
   - e.g. multiple processors, disks, memory banks, pipelining, multiple functional units
2. Principle of Locality
   - Reuse of data and instructions
3. Focus on the Common Case
   - Amdahl's Law

$$\text{Execution time}_{\text{new}} = \text{Execution time}_{\text{old}} \times \left((1 - \text{Fraction}_{\text{enhanced}}) + \frac{\text{Fraction}_{\text{enhanced}}}{\text{Speedup}_{\text{enhanced}}} \right)$$

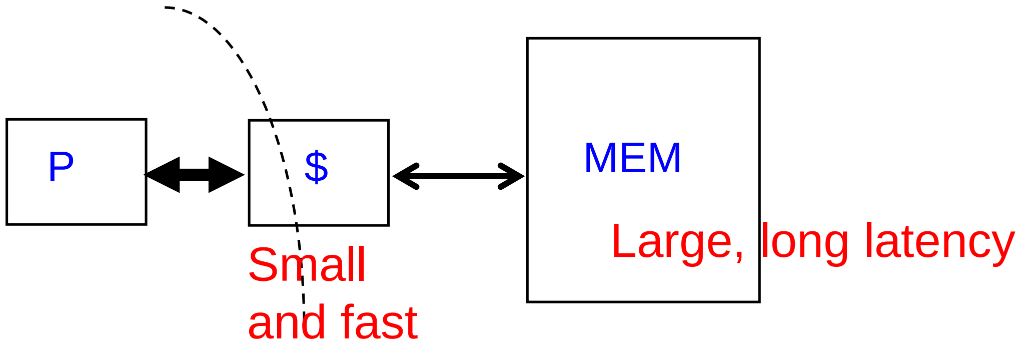
(1) Taking Advantage of Parallelism

- Using parallelism to improve throughput
  - Carry lookahead adders uses parallelism to speed up computing sums from linear to logarithmic in number of bits per operand
  - Set-associative caches searches in parallel by using multiple memory banks searched
  - Pipelining technique overlaps instruction execution to reduce the total time to complete an instruction sequence.
  - Server computer increase throughput of via multiple processors or multiple tasks/disks

- Different levels of parallelism: ILP, DLP, TLP, …

(2) The Principle of Locality

- Program access a relatively small portion of the address space at any instant of time.
- Two Different Types of Locality:
  - Temporal Locality (Locality in Time): If an item is referenced, it will tend to be referenced again soon (e.g., loops, reuse)
  - Spatial Locality (Locality in Space): If an item is referenced, items whose addresses are close by tend to be referenced soon (e.g., straight-line code, array access)
- Cache in memory hierarchy for performance.



(3) Focus on the Common Case

- Common sense guides computer design
  - Since its engineering, common sense is valuable
- In making a design trade-off, favor the frequent case over the infrequent case
  - E.g., Instruction fetch and decode unit used more frequently than multiplier, so optimize it 1st
  - E.g., If database server has 50 disks / processor, storage dependability dominates system dependability, so optimize it 1st
- Frequent case is often simpler and can be done faster than the infrequent case
  - E.g., overflow is rare when adding 2 numbers, so improve performance by optimizing more common case of no overflow
  - May slow down overflow, but overall performance improved by optimizing for the normal case
- What is frequent case and how much performance improved by making case faster => Amdahl's Law

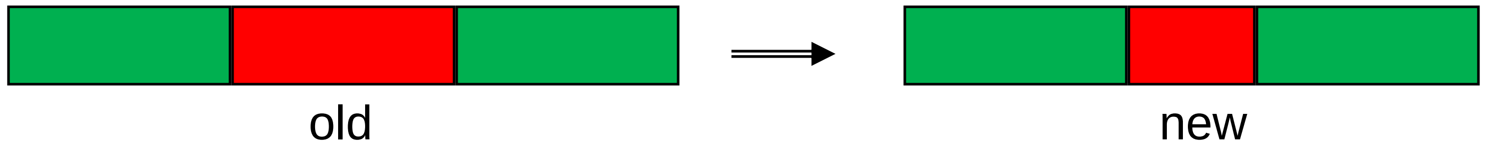
Amdahl's Law

$$\text{ExTime}_{new} = \text{ExTime}_{old} \times \left[(1 - \text{Fraction}_{enhanced}) + \frac{\text{Fraction}_{enhanced}}{\text{Speedup}_{enhanced}} \right]$$

$$\text{Speedup}_{overall} = \frac{\text{ExTime}_{old}}{\text{ExTime}_{new}} = \frac{1}{(1 - \text{Fraction}_{enhanced}) + \frac{\text{Fraction}_{enhanced}}{\text{Speedup}_{enhanced}}}$$

Best you could ever hope to do:

$$\text{Speedup}_{maximum} = \frac{1}{(1 - \text{Fraction}_{enhanced})}$$



Example

Two design alternative for FP square root

1. Add FPSQR hardware

   20% of the execution time in benchmark

   ➔ Speedup factor 10

2. Make all FP instructions faster

   50% of the execution time for all FP instructions

   ➔ 1.6 times faster

Answer

$$\text{Speedup}_{\text{FP SQR}} = \frac{1}{(1-0.2)+\frac{0.2}{10}} = 1.22$$

$$\text{Speedup}_{\text{FP}} = \frac{1}{(1-0.5)+\frac{0.5}{1.6}} = 1.23$$

➔ Improving the performance of the FP operations overall is slightly better because of the higher frequency

SPEC Power Benchmark

- *SPECpower* uses a software stack written in Java which represents the server side of business applications.
- SPECpower measures the power consumption of server at different workload levels
  - Performance: ssj_ops/sec
  - Power: Watts (Joules/sec)

$$\text{Overall ssj\_ops per Watt} = \left(\sum_{i=0}^{N-1} \text{ssj\_ops}_i\right) \Big/ \left(\sum_{i=0}^{N-1} \text{power}_i\right)$$

server side Java operations per second per watt

SPECpower_ssj2008 for Xeon X5650

Target Load %	Performance (ssj_ops)	Average Power (Watts)
100%	865,618	258
90%	786,688	242
80%	698,051	224
70%	607,826	204
60%	521,391	185
50%	436,757	170
40%	345,919	157
30%	262,071	146
20%	176,061	135
10%	86,784	121
0%	0	80
Overall Sum	4,787,166	1,922
Σssj_ops/Σpower =		2,490

Fallacies and Pitfalls

Counterexamples Mistakes

- *Pitfall*: All exponential laws must come to an end
  - Dennard scaling (power density was constant as transistors got smaller, 1974)
    - If the transistor gets smaller then both the current and the voltage are also reduced. ➔ Chip could be designed to operate faster and still use less power.
    - **Counterexample: (1) Threshold voltage. (2) Static power is significant fraction of total power**
  - Hard disk capacity
    - Increasing density per drive by adding more platters
    - **30-100% per year to 5% per year**
  - Moore's Law
    - Most visible with DRAM capacity
    - **ITRS disbanded**
    - Only four foundries left producing state-of-the-art logic chips (2017) and 11 nm, 3 nm might be the limit
- *Fallacy*: Microprocessors are a silver bullet
  - Performance is now a programmer's burden