EEL 5764 Computer Architecture

Sandip Ray
Department of Electrical and Computer Engineering
University of Florida

Lecture 14-15:

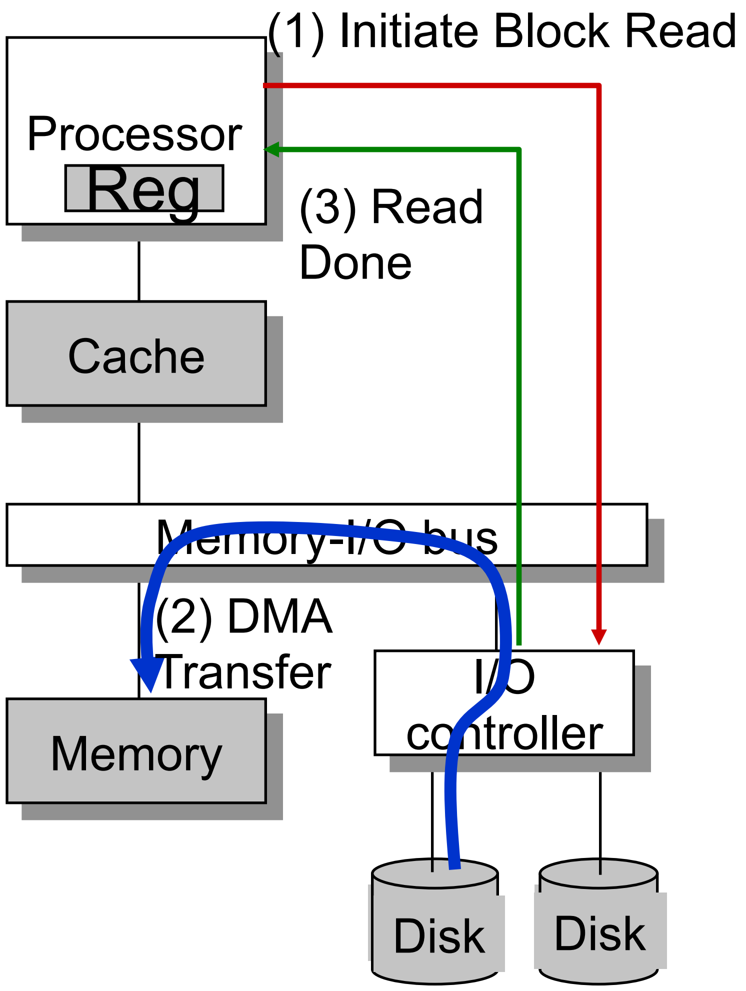
- Virtual Memories
- Instruction Set Architecture

Announcement

- Please use Canvas and preferably the discussion section for your course-related questions
- For any message you send through Canvas please copy the **entire teaching staff (instructor and TA)**
- For HW-related questions and clarifications, please contact TA first

Servicing Page Faults

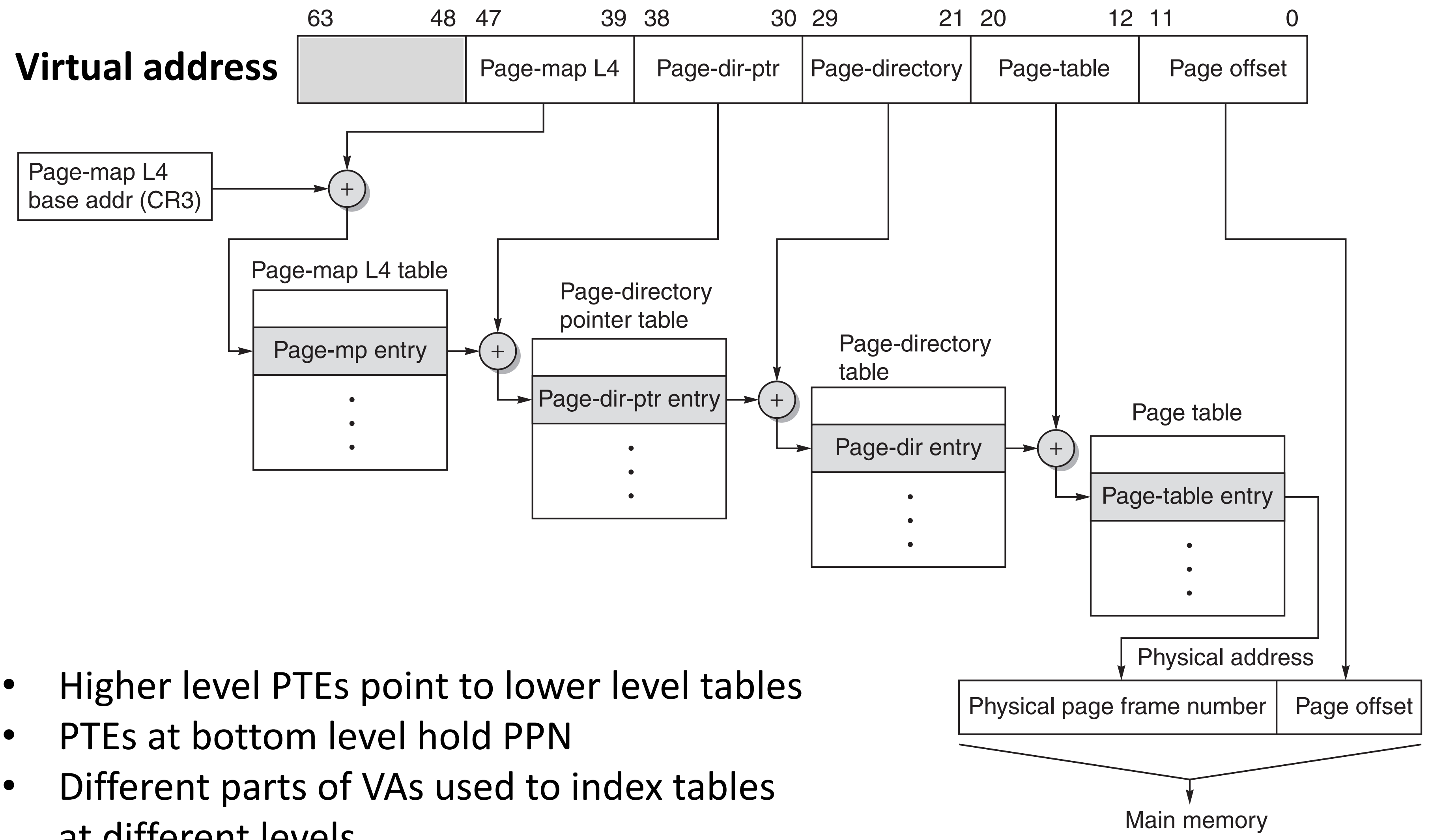
- Processor signals controller
  - → Read block of length P starting at disk address X and store starting at memory address Y
- Read occurs
  - → Direct Memory Access (DMA)
  - → Under control of I/O controller
- I/O controller signals completion
  - → Interrupt processor
  - → OS resumes suspended process



Page Replacement

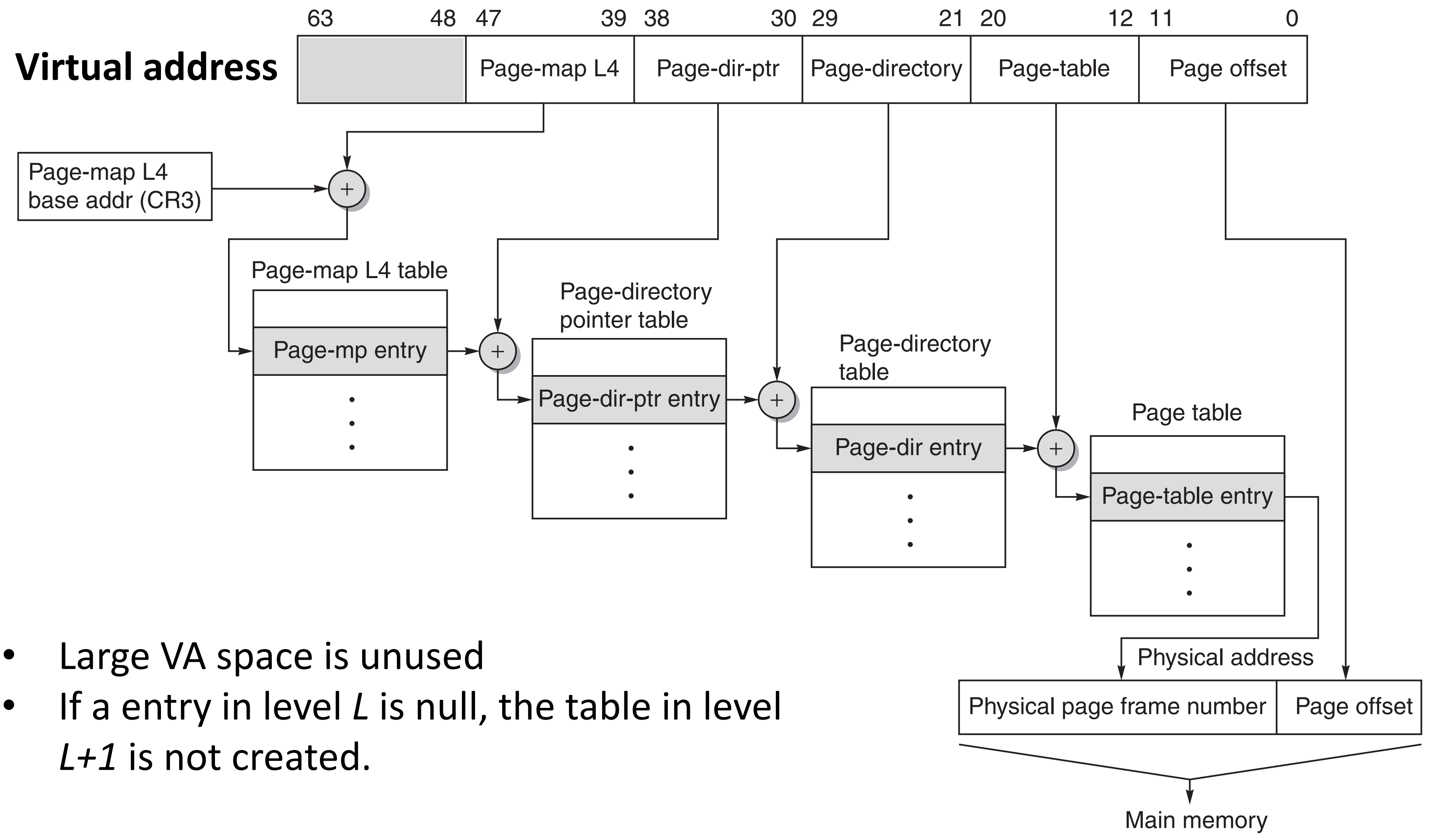
- When there are no available free pages to handle a fault we must find a page to replace.
- This is determined by the *page replacement algorithm*.
- The goal of the replacement algorithm is to reduce the fault rate by selecting the best victim page to remove.
- LRU is the most popular replacement policy.
  - → Each page is associated with a *use/reference* bit.
  - → It is set whenever a page is accessed.
  - → OS periodically clears these bits.

Multi-Level Page Table



- Higher level PTEs point to lower level tables
- PTEs at bottom level hold PPN
- Different parts of VAs used to index tables at different levels

Multi-Level Page Table



- Large VA space is unused
- If a entry in level *L* is null, the table in level *L+1* is not created.

Selecting Page Size

- The larger the page size, the smaller page table
  - →A page table can occupy a lot of space
- Larger page size -> larger L1 cache
- Larger page size -> less TLB misses, faster xlation
- More efficient to xfer larger pages from 2nd storage
- Large page size
  - →Wasted storage
  - →Wasted IO bandwidth
  - →Slower process start up

Virtual Memory – Protection

- Virtual memory and multiprogramming
  - → Multiple processes sharing processor and physical memory
- Protection via virtual memory
  - → Keeps processes in their own memory space
- Role of architecture:
  - → Provide user mode and supervisor mode
  - → Protect certain aspects of CPU state: PC, register, etc
  - → Provide mechanisms for switching between user mode and supervisor mode
  - → Provide mechanisms to limit memory accesses
  - → Provide TLB to translate addresses

Page Table Entries

- Address – physical page number
- Valid/present bit
- Modified/dirty bit
- Reference bit
  - → For LRU
- Protection bits – Access right field defining allowable accesses
  - → E.g., read only, read-write, execute only
  - → Typically support multiple protection modes (kernel vs user)

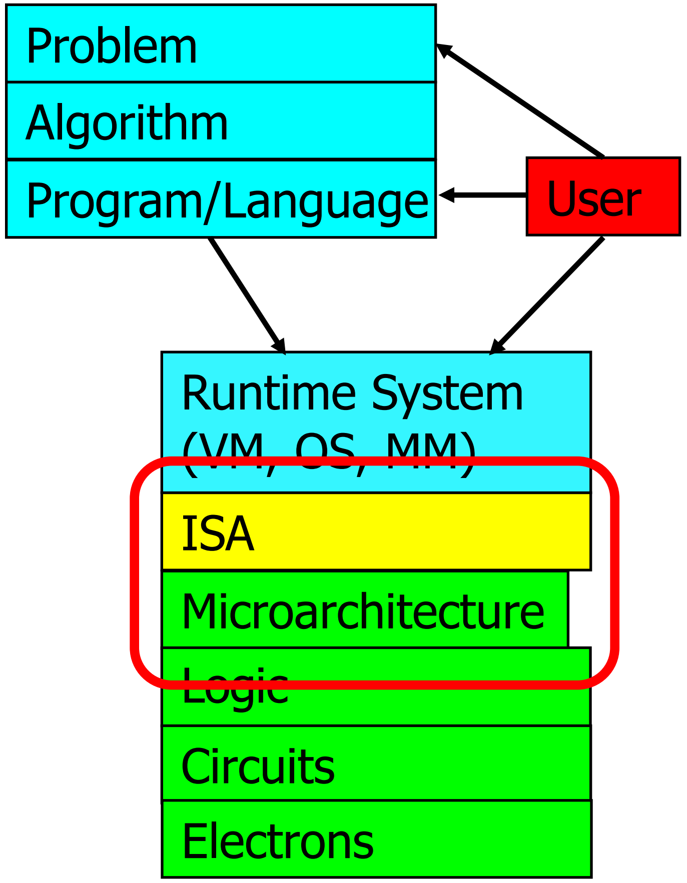
Summary

- OS virtualizes memory and IO devices
  - → Each process has an illusion of private CPU and memory
- Virtual memory
  - → Arbitrarily large memory, isolation/protection, inter-process communication
  - → Reduce page table size
  - → Translation buffers – cache for page table
  - → Manage TLB misses and page faults

Instruction Set Architecture

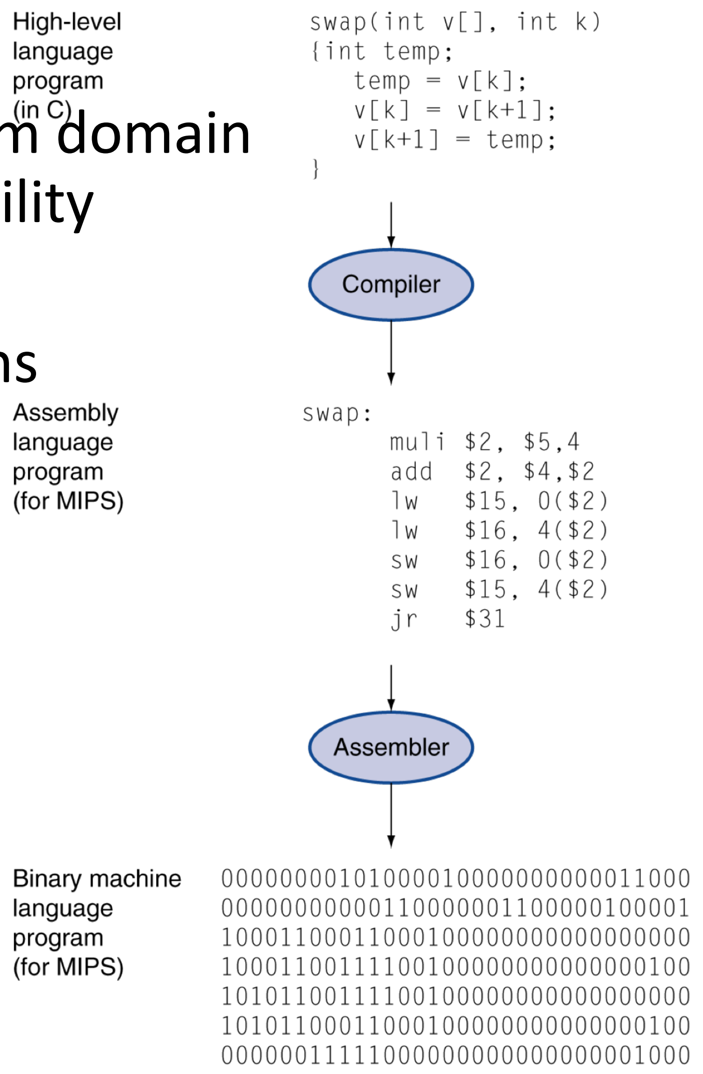
Abstractions

- Abstraction helps us deal with complexity
  - → Hide lower-level detail
- Instruction set architecture (ISA)
  - → The hardware/software interface
  - → Defines storage, operations, etc
- Implementation
  - → The details underlying the interface
  - → An ISA can have multiple implementations



Levels of Program Code

- High-level language
  - → Level of abstraction closer to problem domain
  - → Provides for productivity and portability
- Assembly language
  - → Textual representation of instructions
- Hardware representation
  - → Machine code - Binary digits (bits)
  - → Encoded instructions and data
- Compiler
  - → Translate HL prgm to assembly
- Assembler
  - → Translate assembly to machine code


High-level language program (in C)
Compiler
Assembly language program (for MIPS)
Assembler
Binary machine language program (for MIPS)

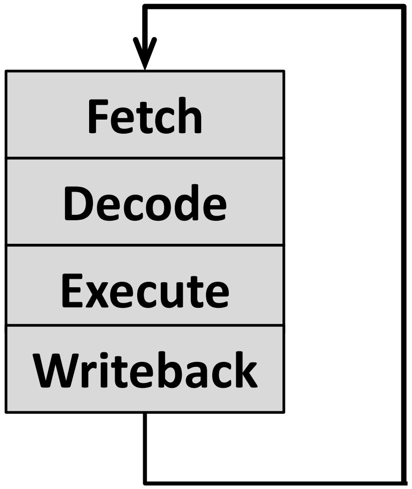
```
swap(int v[], int k)
{int temp;
   temp = v[k];
   v[k] = v[k+1];
   v[k+1] = temp;
}
```

```
swap:
      muli $2, $5,4
      add  $2, $4,$2
      lw   $15, 0($2)
      lw   $16, 4($2)
      sw   $16, 0($2)
      sw   $15, 4($2)
      jr   $31
```

```
00000000101000010000000000011000
00000000000110000001100000100001
10001100011000100000000000000000
10001100111100100000000000000100
10101100111100100000000000000000
10101100011000100000000000000100
00000011111000000000000000001000
```

Program Execution Model

- A computer is just a FSM
  - → States stored in registers, memory, PC, etc
  - → States changed by instruction execution
- An instruction is executed in
  - → **Fetch** an instruction into CPU from memory
  - → **Decode** it to generate control signals
  - → **Execute** it (add, mult, etc)
  - → **Write back** output to reg or memory
- *Programs* and *data* coexist in memory



What Makes a Good ISA?

- **Programmability**
  - → Who does assembly programming these days?
- **Performance/Implementability**
  - → Easy to design high-performance implementations?
  - → Easy to design low-power implementations?
  - → Easy to design low-cost implementations?
- **Compatibility**
  - → Easy to maintain as languages, programs evolve
  - → x86 (IA32) generations: 8086, 286, 386, 486, Pentium, Pentium-II, Pentium-III, Pentium4, Core2, Core i7, ...

Performance

- **Execution time = IC * CPI * cycle time**
- **IC:** instructions executed to finish program
  - → Determined by program, compiler, ISA
- **CPI:** number of cycles needed for each instruction
  - → Determined by program, compiler, ISA, u-architecture
- **Cycle time:** inverse of clock frequency
  - → Determined by u-architecture & technology
- Ideally optimize all three
  - → Their optimizations often against each other
  - → Compiler plays a significant role

Instruction Granularity

- **CISC** (Complex Instruction Set Computing) **ISAs**
  - → Big heavyweight instructions (lots of work per instruction)
  - + Low “insns/program”
  - – Higher “cycles/insn” and “seconds/cycle”
    - ➤ We have the technology to get around this problem

- **RISC** (Reduced Instruction Set Computer) **ISAs**
  - → Minimalist approach to an ISA: simple insns only
  - + Low “cycles/insn” and “seconds/cycle”
  - – Higher “insn/program”, but hopefully not as much
    - ➤ Rely on compiler optimizations

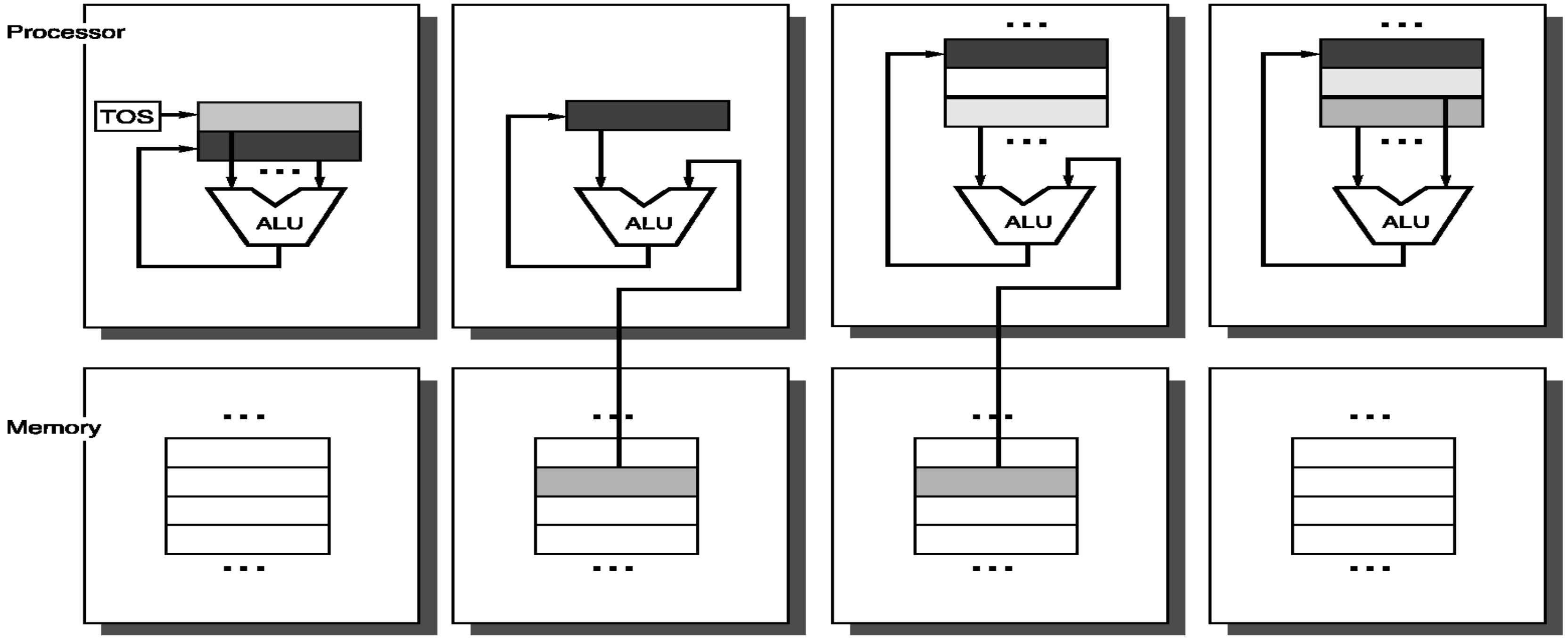
Classifying Architectures

- One important classification scheme is by the type of addressing modes supported.
- Stack architecture: Operands implicitly on top of a stack.
- Accumulator architecture: One operand is implicitly an accumulator (a special register).
- General-purpose register arch.: Operands may be any of a large (typically 10s-100s) # of registers.
  - → Register-memory architectures: One op may be memory.
  - → Load-store architectures: All ops are registers, except in special load and store instructions.

Illustrating Architecture Types

Assembly for `C:=A+B`:

Stack	Accumulator	Register (register-memory)	Register (load-store)
`Push A`	`Load A`	`Load R1,A`	`Load R1,A`
`Push B`	`Add B`	`Add R1,B`	`Load R2,B`
`Add`	`Store C`	`Store C,R1`	`Add R3,R1,R2`
`Pop C`			`Store C,R3`



Number of Registers

- Registers have advantages
  - → faster than memory, good for compiler optimization, hold variables
- Have as many as possible?
  - → **No**
- One reason registers are faster: there are
  - → **fewer of them** – small is fast (hardware truism)
- Another: they are **directly addressed**
  - → More registers, means more bits per register in instruction
  - → Thus, fewer registers per instruction or larger instructions
- More registers means **more saving/restoring**
  - → Across function calls, traps, and context switches
- Trend toward more registers:
  - → 8 (x86) → 16 (x86-64), 16 (ARM v7) → 32 (ARM v8)

Number of Operands

- A further classification is by the max. number of operands, and # that can be memory: e.g.,

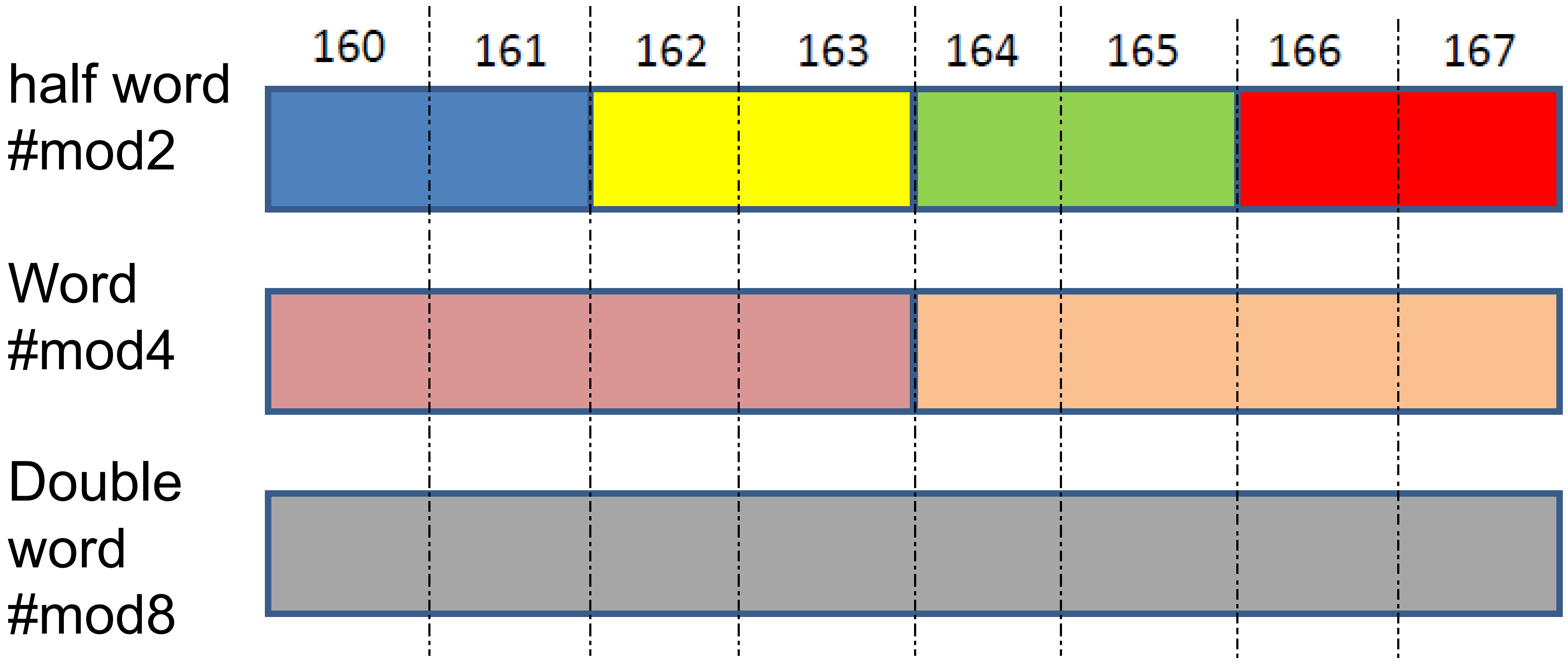
→ 2-operand (e.g. a += b)	
➤ src/dest(reg), src(reg)	
➤ src/dest(reg), src(mem)	IBM 360, x86, 68k
➤ src/dest(mem), src(mem)	VAX
→ 3-operand (e.g. a = b+c)	
➤ dest(reg), src1(reg), src2(reg)	MIPS, PPC, SPARC, &c.
➤ dest(reg), src1(reg), src2(mem)	
➤ dest(mem), src1(mem), src2(mem)	VAX

Memory Addressing

- Byte Addressing
  - → Each byte has a unique address
- Addressing units
  - → Half-word: 16-bit (or 2 bytes)
  - → Word: 32-bit (or 4 bytes)
  - → Double word : 64-bit (or 8 bytes)
  - → Quad word: 128-bit (or 16 bytes)
- Two issues
  - → **Alignment** specifies whether there are any boundaries for word addressing
  - → **Byte order** (Big Endian vs. Little Endian)
    - ➤ specifies how multiple bytes within a word are mapped to memory addresses

Memory Addressing

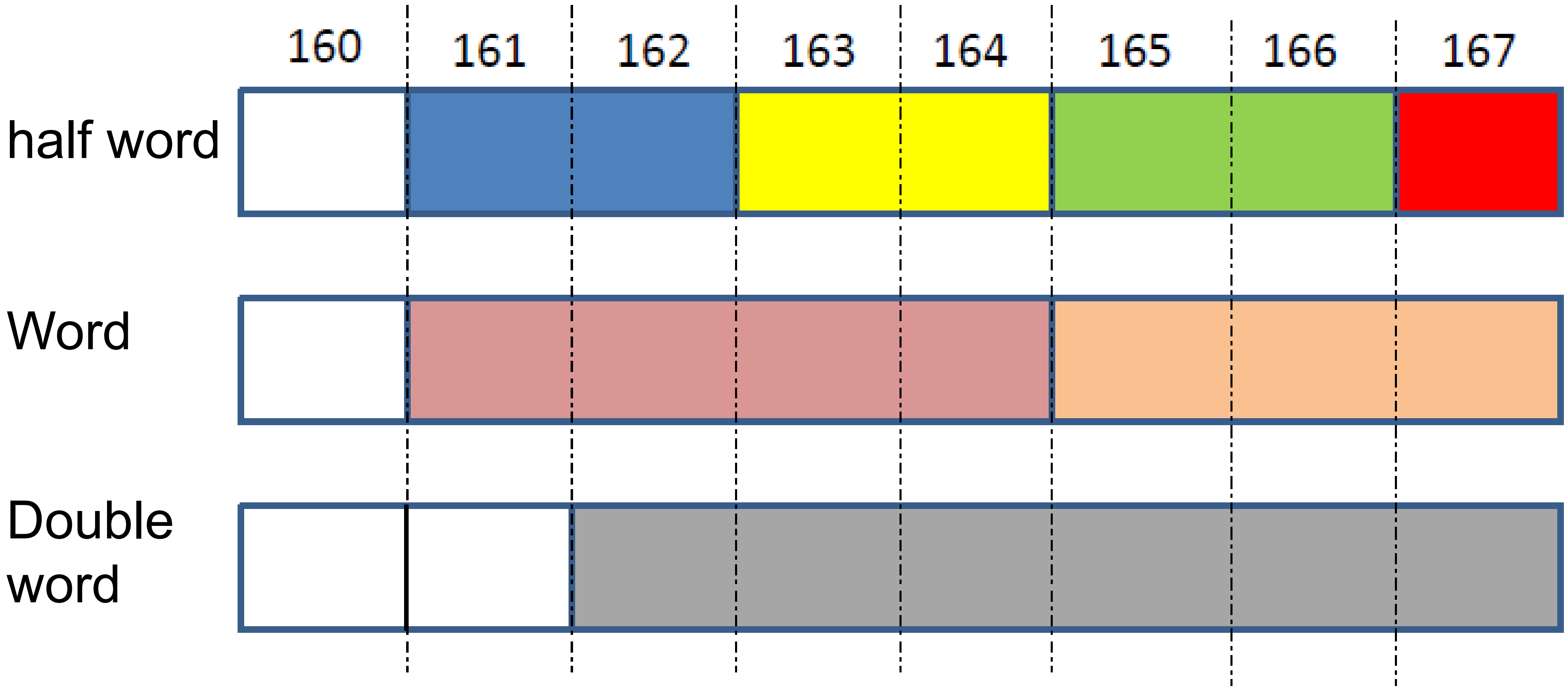
- Alignment
  → Must half word, words, double words begin mod 2, mod 4, mod 8 boundaries



Aligned if *A mod s = 0*

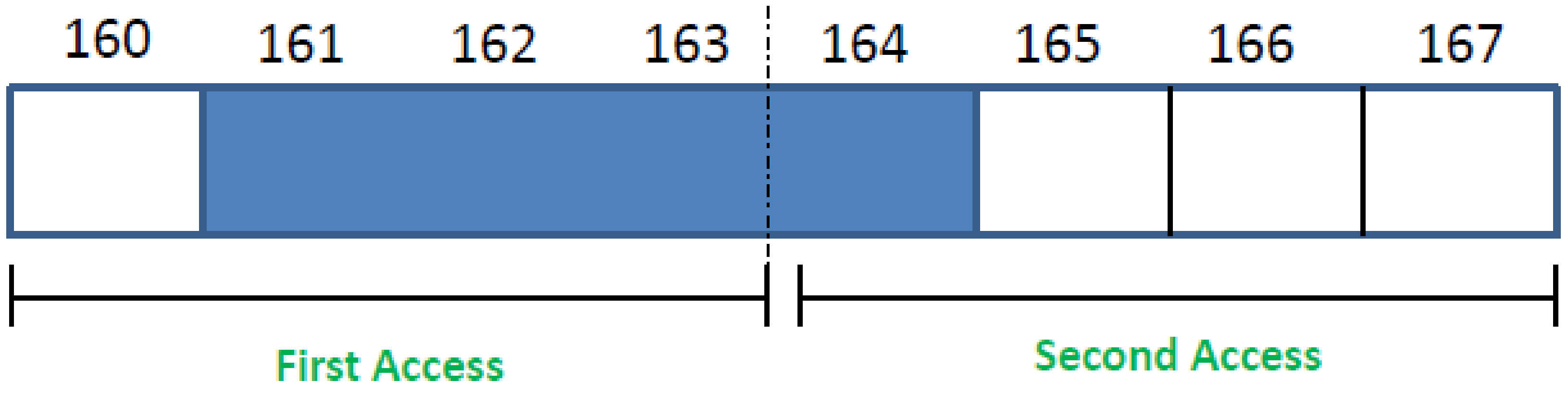
Memory Addressing

- Alignment
  - → Or there no alignment restrictions



Memory Addressing

- Non-aligned memory references may cause multiple memory accesses



- Consider a system in which memory reads return 4 bytes and a reference to a word spans a 4-byte boundary: two memory accesses are required
- Complicates memory and cache controller design
- Assemblers typically force alignment for efficiency