Introduction to Computer Architecture: Review

1. Introduction to Processors
   - Basic computer operation:
     - 1. Load the instruction
     - 2. Figure out what operation to do (control)
     - 3. Figure out what data to use (data)
     - 4. Do the computation
     - 5. Figure out what instruction to load next

2. ISA: Instruction Formats and Execution
   - MIPS instruction formats (R, I, J)
   - Immediate sizes and how they are used
     - Sign-extended
     - How to load 32-bit constants
   - Control
     - Jumps (unconditional – how far can they jump?)
     - Branches (conditional – how far?)
   - RISC vs. CISC

1. Binary Numbers
   - Binary Numbers
     - Adders have 3 inputs and 2 outputs
     - Overflow limits the maximum we can represent
   - Two’s complement
     - Basic idea: biggest digit is negative
     - \(1000 = -8 \times 2^3 + 0\times 2^2 + 0\times 2^1 + 0 = -8\)
     - Numbers range from \(-2^{n-1}\) to \(2^{n-1}-1\)
   - Subtraction (invert and add 1)
   - Addition (same as unsigned)
   - Comparisons (a bit tricky)

Course Outline

1. Introduction to Processors and Binary Numbers
2. ISA: Instruction Formats and Execution
3. ISA: Addressing Modes and Procedure Calls
4. Arithmetic and Integer Numbers
5. Digital Logic
6. Performance
7. Datapath: Single-cycle
8. Datapath: Multi-cycle
9. Datapath: Exceptions and Pipelining
10. Datapath: Pipelining Implementation
11. I/O
12. Memories
13. Caches
14. Virtual Memory 1: Address Translation
15. Virtual Memory 2: TLBs and Caches
16. Review
3. ISA: Addressing Modes

3.1. Base addressing (I-format)

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>Register</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Memory</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Word</td>
</tr>
</tbody>
</table>

Example:

lw R1, 100(R2)

Don’t forget R- and J-addressing modes!

3.2. Add 16 (signed) bits to the address. We can access +/-32,000 (whats?)

4. Arithmetic

<table>
<thead>
<tr>
<th>Binary Addition</th>
<th>Binary Multiplication</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 + 0 = 0</td>
<td>0 x 0 = 0</td>
</tr>
<tr>
<td>0 + 1 = 1</td>
<td>0 x 1 = 0</td>
</tr>
<tr>
<td>1 + 0 = 1</td>
<td>1 x 0 = 0</td>
</tr>
<tr>
<td>1 + 1 = 10</td>
<td>1 x 1 = 1</td>
</tr>
</tbody>
</table>

Addition of two POSITIVE integers:

B = [110111]₂ → [31]₁₀

C₀ = [11111]₁₀ → [11111]₂

Overflow

Serial multiplication

3. ISA: Procedure Calls

3.1. Caller

CALL

add $t1,$a2,$a3

Input arguments are stored in registers:

 callee saves $t0, $s0, $s1

  after the call, the caller restores $t0, $s0, $s1

return

  callee finds its arguments in $s0, $t0, $s1

3.2. Callee

CALL

lw $a0, leaf_example($sp)

sw $t0, leaf_example($sp)

addi $t1, $s1, $t0

lw $s0, 0($sp)

sw $t0, 4($sp)

lw $s1, 8($sp)

sw $t0, -4($sp)

lw $t0, 12($sp)

sw $t0, -8($sp)

add $t1,$a2,$a3

return

  results are placed in $t0

  callee must restore $s0

Caller uses $t0, $s0, $s1.

S0 is not saved, so the caller has to save it. What about $s0, $t0?

After the call, the caller restores $t0, $s0, $s1.

Result is in $s0. Why did we not save $s0?

Summary: ISA

- Architecture is what’s visible to the program about the machine
  - Not everything in the deep implementation is “visible”
  - The name for this invisible stuff is microarchitecture or “implementation”
    (and it’s really messy...but fun.)

- A big piece of the ISA = assembly language structure
  - Primitive instructions, execute sequentially, atomically
  - Issues are formats, computations, addressing modes, etc.
  - Two broad flavors:
    - RISC: lots of complicated instructions
    - CISC: a few essential instructions
  - Basically all recent machines are RISC, but the dominant machine of today, Intel x86, is RISC-like (though they do RISC-like tricks in the guts...)

- We did one example in some detail: MIPS (from PA&Chap 3)
  - A RISC machine, its virtue is that it is pretty simple
  - Can “get” the assembly language without too much memorization

4. Integer Numbers

- Signed Magnitude
  - 1 bit for sign
  - Have two zeros
  - Operations are a pain

- Two’s complement
  - To negate: invert and add 1 (easy to do with C₀)
  - Complement of 0001 is 1011 = 10000 – 0001 = 1111
  - Overflow is different

- Non-integers
  - Fixed point 0010.1100
  - Floating point (mantissa, exponent, and sign)
    - Multiplication easier than addition
5. Digital Logic – Basic Gates

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>Out</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

**AND**

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>Out</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

**OR**

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>Out</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

**XOR**

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>Out</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

**NAND**

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>Out</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

**NOR**

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>Out</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

**XNOR**

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>Out</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

5. Karnaugh Maps

- Order variables such that only 1 changes in each row/column (Grey coding)
- Groups may overlap

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>Out</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>01</td>
<td>11</td>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>00</td>
<td>01</td>
<td>11</td>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>00</td>
<td>01</td>
<td>11</td>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>00</td>
<td>01</td>
<td>11</td>
<td>10</td>
<td>0</td>
</tr>
</tbody>
</table>

5. Logic Blocks — Memory

- 1-hot row enables
- Array of SRAM Cells (Each box stores 1 bit)
- Order variables such that only 1 changes in each row/column (Grey coding)
- Groups may overlap

5. State Storage — Building a Counter

- When the clock edge rises (C→1)
- The next value is stored as the current value
- The new current value then goes through the adder to make a new next value
- Everything takes some time

5. Finite State Machines

<table>
<thead>
<tr>
<th>State</th>
<th>Button A</th>
<th>Button B</th>
</tr>
</thead>
<tbody>
<tr>
<td>IDLE</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>LAUNCH</td>
<td>false</td>
<td>false</td>
</tr>
</tbody>
</table>

5. Logic: Summary

- **Combinational Logic**
  - Inputs “immediately” produce output
  - Use truth tables to determine logic equations
  - Common functions such as MUX, DEMUX, decode, encode
- **Sequential Logic**
  - Current state is updated to next state by the clock
  - Store current state in a latch/memory/flip-flop
  - Combinational logic used to determine the next state, and update the current state on the clock
- **What do you need for the labs?**
  - Combinational logic for the ALU and adder
  - Combinational logic to decode instructions and connect up the ALU
- **What do you need for life?**
  - Understand that state is stored in memories
  - ...that state is updated by combinational logic
  - ...that clock speed depends on how long it takes to calculate the next state
6. Performance

- Comparing Machines Using Microarchitecture
  - Latency (instruction execution time from start to finish)
  - Throughput (number of instructions per unit of time)
  - Processor cycle time (GHz)
  - CPI - cycles per instruction
  - MIPS - millions of instructions per second
  - FLOPs – floating point instructions per second (also GFLOP/TFLOP/PFLOP/EFLOP)

- Comparing Machines Using Benchmark Programs
  - Which programs?
  - Benchmark suites/microbenchmarks
  - Different Means: Arithmetic, Harmonic, and Geometric

6. Metrics

- CPI (Cycles Per Instruction):
  - 25 instructions are loads/stores (each takes 2 cycles)
  - 50 instructions are adds (each takes 1 cycle)
  - 25 instructions are square root (each takes 100 cycles)
  - CPI = \( \frac{(25 \times 2) + (50 \times 1) + (25 \times 100)}{100} = 2600 / 100 = 26.0 \)

- MIPS (Millions of Instructions Per Second):
  - Machine A has a special instruction for performing square root calculations
    - It takes 100 cycles to execute
  - Machine B doesn’t have the special instruction: it must perform square root calculations in software using simple instructions (e.g., Add, Mul, Shift) that each take 1 cycle to execute
  - Machine A: 1/100 MIPS = 0.01 MIPS
  - Machine B: 1 MIPS
  - Square root takes 100 cycles: hurts average “instructions per second” but may improve performance dramatically!

6. Comparisons

- Averages (Arithmetic, Harmonic, Geometric)
- Normalizing (Weights, Runtimes)
- What you really care about is time… for your application (benchmarks)
- Amdahl’s Law

6. Performance Summary

- Performance is important to measure
  - For architects comparing different deep mechanisms
  - For developers of software trying to optimize code, applications
  - For users, trying to decide which machine to use, or to buy

- Performance metric are subtle
  - Easy to mess up the “machine A is XXX times faster than machine B” numerical performance comparison
  - You need to know exactly what you are measuring: time, rate, throughput, CPI, cycles, etc.
  - You need to know how combining these to give aggregate numbers does different kinds of “distortions” to the individual numbers (P&H is good in Chapter 2 on this stuff)
  - No metric is perfect, so lots of emphasis on standard benchmarks today

7. Datapath: Single-cycle

- Control Path
  - What to do
  - Logic for decoding signals (ALU, register file, muxes)

- Data Path
  - Process the data
  - What resources do we need? (ALU, register file, memory, PC)

7. Complete Single-cycle Datapath
7. Cost of the Single Cycle Architecture

- Instr Class 1
- Instr Class 2
- Instr Class 3

Our Cycle Time (longest Instruction)

Most of the time is wasted!

8. Multi-cycle Solution

Idea: Let the FASTEST instruction determine clock period

- Instr Class 1
- Instr Class 2
- Instr Class 3

Takes 4 cycles

Takes 2 cycles

Less Wasted Time

8. Performance of Multicycle Implementation

- Each type of instruction can take a variable # of cycles
- Example
  - Assume the following instruction distributions:
    - loads: 5 cycles 22%
    - stores: 4 cycles 11%
    - R-type: 4 cycles 49%
    - branches: 3 cycles 16%
    - jump: 3 cycles 2%
  - What’s the average Cycles Per Instruction (CPI)
    - CPI = (CPU clock cycles/Instruction Count)
    - CPI = (5 cycles * 0.22) + (4 cycles * 0.11) + (4 cycles * 0.49) + (3 cycles * 0.16) + (3 cycles * 0.02)
    - CPI = 4.04 cycles per instruction
  - What was the CPI for the single-cycle machine?
    - Single cycle implies 1 clock cycle per instruction --> CPI = 1.0
    - So isn’t the single-cycle machine about 4 times faster?

8. Summary

- Single cycle implementations have to consider the worst case delay through the datapath to come-up with the cycle time.
- Multicycle implementations have the advantage of using a different number of cycles for executing each instruction.
- In general, the multicycle machine is better than the single cycle machine, but the actual execution time strongly depends on the workload.
- The most widely used machine implementation is neither single cycle, nor multicycle – it’s the pipelined implementation.
  (Next lecture)
9. Pipelined Datapath

This computation is "too long"

Pipelined version, 5 pipe stages

100 ns

~20 ns

Latches, called 'Pipeline registers', break up computation into stages. They store the intermediate results.

8. Performance of Pipelined Systems

- Unpipelined
  - Throughput: 1 per 5 cycles
  - Time

- Pipelined
  - Throughput: 1 per 1 cycle
  - Time

- Ideally, Speedup\textsubscript{pipeline} = Time\textsubscript{sequential} / Pipeline Depth

8. Complete 5 Stage Pipeline

Flow of Instructions Through Pipeline

10. Data Hazards

- In this particular case...
  - R10 value is not computed or returned to register file when later instruction wants to use it as an input

Double pumping reg file doesn't help here; later instruction needs R10 2 clock cycles before it's been computed & stored back.
Oops...


- What prevents us from just doing a zillion pipe stages?
  - Those latches are \textbf{NOT} free, they take up area, and there is a real delay to go THRU the latch itself
  - In modern, deep pipeline (15-20 stages), this is a real effect
  - Typically see logic "depths" in one pipe stage of 15-20 "gates"

At these speeds, and with this few levels of logic, latch delay is important

10 stage pipe

~20 ns

~20 ns

5 cycles

16

32

Current PC

ADDER

<< 2

Program Execution

Time

Clock Cycle 1

Clock Cycle 2

Clock Cycle 3

Clock Cycle 4

Clock Cycle 5

Clock Cycle 6

Clock Cycle 7

LW R1, 100(R0)

LW R2,200(R0)

LW R3, 300(R0)

In cycle 4 we have 3 instructions "in-flight":
Inst 1 is accessing the memory (DM)
Inst 2 is using the ALU (EX)
Inst 3 is access the register file (ID)
10. Forwarding

10. Hazards
- **Data hazards**
  - Instruction depends on result of prior computation which is not ready yet
  - Stall, double pump, and forward, to fix
- **Structural hazards**
  - HW cannot support a combination of instructions
  - OK, maybe add extra hardware resources; may still have to stall
- **Control hazards**
  - Pipelining of branches and other instructions which change the PC
  - Branch predictors, branch delay slots, early branch computation

10. Pipelining Summary
- **Need to keep the pipeline full for performance**
  - Hazards make this hard
  - Dependencies and resource conflicts
  - Time to access memory (cache)
- **Performance**
  - Can’t use infinitely many stages
  - Code and pipeline stages (branch delay slot)
- **Exceptions/Interrupts**
  - Can happen at different places in the pipeline
  - Can happen out-of-order (early in the pipeline for a later instruction vs. later in the pipeline for an earlier one)
  - Need to restart instructions
  - Jump to the OS to handle them

11. Input/Output
- **Busses**
  - Clocking, width, arbitration
- **Performance**
  - Latency
  - Throughput
- **Talking to devices**
  - Method: memory-mapped/IO instructions
  - Means: polling, interrupt, DMA

12. Memory
- **Random Access Memories**
  - **DRAM**: Dynamic Random Access Memory
    - High density, low power, cheap, but
    - Dynamic needs to be “refreshed” regularly
  - **SRAM**: Static Random Access Memory
    - Low density, high power, expensive, fast
    - Static content will last “forever” (until lose power)
- **What gets used where?**
  - Main memory is DRAM: you need it big, so you need it cheap
  - CPU cache memory is SRAM: you need it fast, so if it’s more expensive, so it’s smaller than you would usually want due to resource limitations
- **Relative performance**
  - Size: DRAM/SRAM: 4-by bigger for DRAM
  - Cost/Cycle time: SRAM/DRAM: 8-16x faster, more $$$ for SRAM

12. Performance Impact (4 cycles)
- **With 1-cycle memory system**
  - Program took 8 cycles
  - CPI = 8 cycles / 4 instructions = 2.0
- **With lots more instructions**
  - CPI would approach 1.0
- **With 4-cycle memory system**, program takes 18 cycles
  - CPI = 17 cycles / 4 instructions = 4.5
  - Doesn’t include instruction fetch penalty found in real memory system
12. Memory Hierarchy of a Modern Computer System

- By taking advantage of the principle of locality:
  - Present the user with as much memory as is available in the cheapest technology.
  - Provide access at the speed offered by the fastest technology.

12. Memory Hierarchy: How Does it Work?

- Temporal Locality (Locality in Time):
  - If an item is referenced, the same item will tend to be referenced again soon
  - Keep most recently accessed data items closer to the processor
- Spatial Locality (Locality in Space):
  - If an item is referenced, nearby items will tend to be referenced soon
  - Move recently accessed “blocks” (groups of contiguous words) closer to proc.

- “Block” (or “line”) - minimum unit of data between 2 levels

13. Basic Cache Design

- Cache only holds a portion of a program
  - Which part of the program does the cache contain?
  - Cache holds most recently accessed references
  - Cache is divided into units called cache blocks (also known as cache lines), each block holds a contiguous set of memory addresses
  - How does the CPU know which part of the program the cache is holding?
    - Each cache block has extra bits, called the cache tag, which holds the main memory address of the data in the block

13. The ABC’s (or 1-2-3-4’s) of Caches

- Caching is a general concept used in processors, operating systems, file systems, and applications.
  - Wherever it is used, there are four basic questions which arise. These include:
    - Q1: Where can a block be placed in a cache?
      - Direct mapped, associative, fully-associative
    - Q2: How is a block found if it is in a cache?
      - Indexing (direct mapped), limited search (associative), full search (fully-associative)
    - Q3: Which block should be replaced on a miss?
      - Random, least-recently used (LRU)
    - Q4: What happens on a write?
      - Write-through or write-back

13. Cache Block Placement

### Direct-Mapped Cache Indexing

- 4-entry direct-mapped cache

<table>
<thead>
<tr>
<th>Cache Line 0</th>
<th>Address mod 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cache Line 1</td>
<td>Address mod 1</td>
</tr>
<tr>
<td>Cache Line 2</td>
<td>Address mod 2</td>
</tr>
<tr>
<td>Cache Line 3</td>
<td>Address mod 3</td>
</tr>
</tbody>
</table>

Memory Space
Direct-Mapped Cache Indexing

• 4-entry direct-mapped cache

<table>
<thead>
<tr>
<th>Tag</th>
<th>Cache Line 0</th>
<th>Address</th>
<th>Index</th>
<th>mod 0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Memory addresses that map to the same cache line will conflict in the cache – we can only store one or the other.

13. Direct-Mapped Cache Indexing

• Direct map caches have a 1:1 mapping from memory addresses to cache entries
  
  – E.g., 4-entry cache, with 4 bytes per line
    
    • Entry 0: xxx0xx
    
    • Entry 1: xxx01x
    
    • Entry 2: xxx10x
    
    • Entry 3: xxx11x
  
  – So addresses 0100000 and 1100001 map to the same cache line.
  
  – To tell which is in the cache we look at the tag:
    
    – If the tag is 000 we have the first memory address
    
    – If the tag is 110 we have the second memory address
  
  – To get individual bytes, we look at the byte offset
    
    – 000000 is the first byte in the line in cache entry 00
    
    – 010000 is the second byte in the line in cache entry 00

13. Set-Associative Cache Indexing

• Set associative caches have a 1:1 mapping from memory addresses to sets

• Within a set a memory address can be put anywhere (multiple entries per set)
  
  – E.g., 2-way set-associative cache with 4 bytes per line
    
    • Set 0: xxx0xx
    
    • Set 1: xxx1xx
  
    – So addresses 1010000 and 0110001 map to the same set, but the set can have multiple entries.
    
    – If the set has 2 entries, we can store both values in the cache, even though they map to the same set.

Set-Associative Cache Indexing

• 4-entry, 2-way set-associative cache, 4 bytes/line

<table>
<thead>
<tr>
<th>Address</th>
<th>Index</th>
<th>Set 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>xxxxxx</td>
<td>xxxxx</td>
<td>x</td>
</tr>
<tr>
<td>xxxxxx</td>
<td>xxxxx</td>
<td>x</td>
</tr>
<tr>
<td>xxxxxx</td>
<td>xxxxx</td>
<td>x</td>
</tr>
<tr>
<td>xxxxxx</td>
<td>xxxxx</td>
<td>x</td>
</tr>
</tbody>
</table>

Need the tag to tell us which memory address xxxxx is in each of the lines for YZ2.
Set-Associative Cache Indexing

- 4-entry, 2-way set-associative cache, 4 bytes/line

<table>
<thead>
<tr>
<th>Address</th>
<th>Index</th>
<th>Tag</th>
</tr>
</thead>
<tbody>
<tr>
<td>xxxxxx000</td>
<td>xxxxxx011</td>
<td>xxxxxx101</td>
</tr>
<tr>
<td>xxxxxx000</td>
<td>xxxxxx011</td>
<td>xxxxxx101</td>
</tr>
<tr>
<td>xxxxxx000</td>
<td>xxxxxx011</td>
<td>xxxxxx111</td>
</tr>
</tbody>
</table>

Memory Space

But now we can have any two of set 0 and any two of set 1 at the same time.

Q1: Block Placement

Direct Mapped
block 12 can go only into block 4 (12 mod 8)

Each memory location maps to one cache line. Mapping is done by a modulo operator, which is accomplished by ignoring some of the MSBs. (e.g., address 001100 + 12 and 000100 + 4 both get mapped to 100 + 4)

Q1: Block Placement

Fully-associative
block 12 can go anywhere

Any memory location can go to any cache line. This is infinitely flexible, but... It requires very complex (and slow) hardware.

Q1: Block Placement

2-way Set-Associative
block 12 can go anywhere in set

Every memory location maps to exactly one set of cache lines. Within a set it can go in any location. Tradeoff between simplicity (direct mapped to sets) and flexibility (fully associative within sets).

13. Cache Performance

- What’s the impact on performance (CPU time) when the following cache behavior is included:
  - 50 cycle miss penalty
  - All instructions normally take 2.0 cycles (excluding memory stalls)
  - Miss rate is 2.0%
  - Average of 1.33 memory references per instruction

IC = T (CPU) / (Memory stall cycles/instruction) / 4 (Clock cycle time)

C = T (CPU) / (Memory stall cycles/instruction) / A (Clock cycle time)

- Two important results to keep in mind:
  - The lower the IC, the higher the relative impact of a cache miss penalty (more sensitive to memory latency)
  - Comparing two machines with identical memory systems, the machine with the higher clock rate will have the larger number of clock cycles per miss and hence the memory portion of its CPI will be higher.

14. Virtual Memory

- What is virtual memory?
  - Technique that allows execution of a program that
    - can reside in non-contiguous memory locations
    - does not have to completely reside in memory
  - Allows the computer to “fake” a program into believing that its
    - memory is contiguous
    - memory space is larger than physical memory

- Why is VM important?
  - Cheap, no longer have to buy lots of RAM
  - Removes burden of memory resource management from the programmer
  - Enables multiprocessing, time-sharing, protection
14. Basic VM Algorithm
- Program uses virtual addresses (load, store, instruction fetch)
- Computer translates virtual address (VA) to physical address (PA)
- Computer reads RAM using PA, returning the data to program

![Virtual Address to Physical Address Translation](image)

14. Page Tables and Entries
- Page Tables
  - Size of entries/size of table
  - Multi-level page tables (why?)
  - How to look up entries
  - TLB
  - Thrashing of pages (LRU example)

14. Making Address Translation Fast
- A cache for address translations: translation lookaside buffer

![Address Translation/Cache Lookup](image)

14. Page Table Protection
- Separate page tables (per process) provide page-level protection
- OS creates and manages page tables so that no user-level process can alter any process’s page table
- Page tables are mapped into kernel memory where only the OS can read or write

![Page Table Protection](image)

15. Multi-level Page Tables to Save Space
- Level 1 Table (Page Directory)
- Level 2 Table (Page Table)

![Multi-level Page Tables](image)
Overlapped Cache & TLB Access

- Simple for small caches
  - \( IDX + BO \leq PO \)
- Must satisfy
  - Cache size/Assoc \( \leq \) Page size
- Assume 4K pages & 2-way set-associative caches
- What is the max size allowed for parallel address translation to work?

Virtual Address Cache

- Lookup using VA
- TLB access on miss
- Use PA to access next level (L2)

Or, Only Use Virtual Bits to Index Cache

- Don’t need to wait for TLB
- Parallel TLB access (e.g., for larger caches)
- Physically-indexed but Virtually-indexed Cache
- Can distinguish addresses from different processes
- But, what if multiple processes share memory?

Summary

- Memory access is hard and complicated:
  - Speed of CPU core demands very fast memory access. We do cache hierarchy to solve this one. 
    - cache misses are most of the time. Occasionally a hit.
  - Size of programs demands large RAM. We do VM hierarchy to solve this one. 
    - VM misses are most of the time. Occasionally a hit.
- VM hierarchy:
  - Another form of cache, but now between RAM and disk.
  - Atomic units of memory are pages, typically 4kB to 2MB.
  - Page table serves as translation mechanism from virtual to physical address
    - Page table lives in physical memory, managed by OS
    - For this addresses, multi-level tables used, some of the table is in VM
  - TLB is yet another cache—caches translated addresses, page table entries.
    - Saves from having to go to physical memory to do lookup on each access
    - Usually very small, managed by OS
  - VM, TLB, cache form “interesting” interactions.
    - Big impacts on speed, pipelining. Big impacts on exactly where the virtual to physical mapping takes place.

And Now For Something Completely Different...

- Course Evaluations! (15 minutes)

  Followed by...
  ...what to expect on the exam (without ruining too much of the surprise)
Final Exam

- Format
  - 3 short-answer questions
  - 6 true/false (-1/0/1 points each)
  - 4 or 5 longer questions
  - Hopefully no more than 2.5 hours to finish
  - (This is a bit harder than the exam from last year)
  - You are allowed one double-sided, hand-written, A4 sheet of notes during the exam and a calculator

- Likely topics:
  - Caches, virtual memory, performance, pipelines, assembly, arithmetic, (simple) logic, input/output, etc.
  - Very likely topics: anything that we spent time going through multiple animations/examples in class

Questions?