CS 61C: Great Ideas in Computer Architecture (Machine Structures)

*Instruction Level Parallelism: Multiple Instruction Issue*

Guest Lecturer: Justin Hsia
You Are Here!

- **Software**
  - **Parallel Requests**
    - Assigned to computer
    - e.g., Search “Katz”
  - **Parallel Threads**
    - Assigned to core
    - e.g., Lookup, Ads
  - **Parallel Instructions**
    - >1 instruction @ one time
    - e.g., 5 pipelined instructions
  - **Parallel Data**
    - >1 data item @ one time
    - e.g., Add of 4 pairs of words
  - **Hardware descriptions**
    - All gates functioning in parallel at same time

- **Hardware**
  - **Warehouse Scale Computer**
  - **Harness Parallelism & Achieve High Performance**

- **Today’s Lecture**
  - Instruction Unit(s)
    - A₀+B₀, A₁+B₁, A₂+B₂, A₃+B₃
  - Functional Unit(s)
  - Cache
  - Memory
  - Input/Output
  - Core

- **Smart Phone**

7/28/2011 Summer 2011 -- Lecture #23
Agenda

• Control Hazards
• Higher Level ILP
• Administrivia
• Dynamic Scheduling
• Technology Break
• Example AMD Barcelona
• Big Picture: Types of Parallelism
• Summary
Review: Hazards

Situations that prevent starting the next instruction in the next clock cycle

1. **Structural hazards**
   - A required resource is busy (e.g., needed in multiple stages)

2. **Data hazard**
   - Data dependency between instructions.
   - Need to wait for previous instruction to complete its data read/write

3. **Control hazard**
   - Flow of execution depends on previous instruction
Review: Load / Branch Delay Slots

- Stall is equivalent to nop

\texttt{lw \$t0, 0(\$t1)}

\texttt{nop}

\texttt{sub \$t3,\$t0,\$t2}

\texttt{and \$t5,\$t0,\$t4}

\texttt{or \$t7,\$t0,\$t6}
Agenda

• Control Hazards
• Higher Level ILP
• Administrivia
• Dynamic Scheduling
• Technology Break
• Example AMD Barcelona
• Big Picture: Types of Parallelism
• Summary
Greater Instruction-Level Parallelism (ILP)

• Deeper pipeline (5 => 10 => 15 stages)
  – Less work per stage ⇒ shorter clock cycle
• Multiple issue is *superscalar*
  – Replicate pipeline stages ⇒ multiple pipelines
  – Start multiple instructions per clock cycle
  – CPI < 1, so use Instructions Per Cycle (IPC)
  – E.g., 4 GHz 4-way multiple-issue
    • 16 BIPS, peak CPI = 0.25, peak IPC = 4
  – But dependencies reduce this in practice
Multiple Issue

• **Static multiple issue**
  – Compiler groups instructions to be issued together
  – Packages them into “issue slots”
  – Compiler detects and avoids hazards

• **Dynamic multiple issue**
  – CPU examines instruction stream and chooses instructions to issue each cycle
  – Compiler can help by reordering instructions
  – CPU resolves hazards using advanced techniques at runtime
Superscalar Laundry: Parallel per stage

- More resources, HW to match mix of parallel tasks?
Pipeline Depth and Issue Width

- Intel Processors over Time

<table>
<thead>
<tr>
<th>Microprocessor</th>
<th>Year</th>
<th>Clock Rate</th>
<th>Pipeline Stages</th>
<th>Issue width</th>
<th>Cores</th>
<th>Power</th>
</tr>
</thead>
<tbody>
<tr>
<td>i486</td>
<td>1989</td>
<td>25 MHz</td>
<td>5</td>
<td>1</td>
<td>1</td>
<td>5W</td>
</tr>
<tr>
<td>Pentium</td>
<td>1993</td>
<td>66 MHz</td>
<td>5</td>
<td>2</td>
<td>1</td>
<td>10W</td>
</tr>
<tr>
<td>Pentium Pro</td>
<td>1997</td>
<td>200 MHz</td>
<td>10</td>
<td>3</td>
<td>1</td>
<td>29W</td>
</tr>
<tr>
<td>P4 Willamette</td>
<td>2001</td>
<td>2000 MHz</td>
<td>22</td>
<td>3</td>
<td>1</td>
<td>75W</td>
</tr>
<tr>
<td>P4 Prescott</td>
<td>2004</td>
<td>3600 MHz</td>
<td>31</td>
<td>3</td>
<td>1</td>
<td>103W</td>
</tr>
<tr>
<td>Core 2 Conroe</td>
<td>2006</td>
<td>2930 MHz</td>
<td>14</td>
<td>4</td>
<td>2</td>
<td>75W</td>
</tr>
<tr>
<td>Core 2 Yorkfield</td>
<td>2008</td>
<td>2930 MHz</td>
<td>16</td>
<td>4</td>
<td>4</td>
<td>95W</td>
</tr>
<tr>
<td>Core i7 Gulftown</td>
<td>2010</td>
<td>3460 MHz</td>
<td>16</td>
<td>4</td>
<td>6</td>
<td>130W</td>
</tr>
</tbody>
</table>
Pipeline Depth and Issue Width

- Clock
- Power
- Pipeline Stages
- Issue width
- Cores


1 10 100 1000 10000
Static Multiple Issue

• Compiler groups instructions into *issue packets*
  – Group of instructions that can be issued on a single cycle
  – Determined by pipeline resources required

• Think of an issue packet as a very long instruction (VLIW)
  – Specifies multiple concurrent operations
Scheduling Static Multiple Issue

• Compiler must remove some/all hazards
  – Reorder instructions into issue packets
  – No dependencies with a packet
  – Possibly some dependencies between packets
    • Varies between ISAs; compiler must know!
  – Pad with nop if necessary
MIPS with Static Dual Issue

• Dual-issue packets
  – One ALU/branch instruction
  – One load/store instruction
  – 64-bit aligned
    • ALU/branch, then load/store
    • Pad an unused instruction with nop

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction type</th>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>MEM</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>ALU/branch</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
<tr>
<td>n + 4</td>
<td>Load/store</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
<tr>
<td>n + 8</td>
<td>ALU/branch</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
<tr>
<td>n + 12</td>
<td>Load/store</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
<tr>
<td>n + 16</td>
<td>ALU/branch</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
<tr>
<td>n + 20</td>
<td>Load/store</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
</tbody>
</table>
Hazards in the Dual-Issue MIPS

• More instructions executing in parallel
• EX data hazard
  • Forwarding avoided stalls with single-issue
  • Now can’t use ALU result in load/store in same packet
    • \texttt{add $t0, $s0, $s1}
    • \texttt{load $s2, 0($t0)}
      • Split into two packets, effectively a stall
• Load-use hazard
  • Still one cycle use latency, but now two instructions
• More aggressive scheduling required
Scheduling Example

• Schedule this for dual-issue MIPS

```
Loop: lw $t0, 0($s1)       # $t0=array element
     addu $t0, $t0, $s2  # add scalar in $s2
     sw $t0, 0($s1)     # store result
     addi $s1, $s1,-4   # decrement pointer
     bne $s1, $zero, Loop # branch $s1!=0
```

<table>
<thead>
<tr>
<th>ALU/branch</th>
<th>Load/store</th>
<th>cycle</th>
</tr>
</thead>
<tbody>
<tr>
<td>Loop:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>nop</td>
<td>lw $t0, 0($s1)</td>
<td>1</td>
</tr>
<tr>
<td>addi $s1, $s1,-4</td>
<td>nop</td>
<td>2</td>
</tr>
<tr>
<td>addu $t0, $t0, $s2</td>
<td>nop</td>
<td>3</td>
</tr>
<tr>
<td>bne $s1, $zero, Loop</td>
<td>sw $t0, 4($s1)</td>
<td>4</td>
</tr>
</tbody>
</table>

- IPC = 5/4 = 1.25 (c.f. peak IPC = 2)
Loop Unrolling

• Replicate loop body to expose more parallelism

• Use different registers per replication
  – Called register renaming
  – Avoid loop-carried anti-dependencies
    • Store followed by a load of the same register
    • Aka “name dependence”
      – Reuse of a register name
## Loop Unrolling Example

<table>
<thead>
<tr>
<th>ALU/branch</th>
<th>Load/store</th>
<th>cycle</th>
</tr>
</thead>
<tbody>
<tr>
<td>Loop:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>addi $s1, $s1, -16</td>
<td>lw $t0, 0($s1)</td>
<td>1</td>
</tr>
<tr>
<td>nop</td>
<td>lw $t1, 12($s1)</td>
<td>2</td>
</tr>
<tr>
<td>addu $t0, $t0, $s2</td>
<td>lw $t2, 8($s1)</td>
<td>3</td>
</tr>
<tr>
<td>addu $t1, $t1, $s2</td>
<td>lw $t3, 4($s1)</td>
<td>4</td>
</tr>
<tr>
<td>addu $t2, $t2, $s2</td>
<td>sw $t0, 16($s1)</td>
<td>5</td>
</tr>
<tr>
<td>addu $t3, $t4, $s2</td>
<td>sw $t1, 12($s1)</td>
<td>6</td>
</tr>
<tr>
<td>nop</td>
<td>sw $t2, 8($s1)</td>
<td>7</td>
</tr>
<tr>
<td>bne $s1, $zero, Loop</td>
<td>sw $t3, 4($s1)</td>
<td>8</td>
</tr>
</tbody>
</table>

- **IPC = 14/8 = 1.75**
  - Closer to 2, but at cost of registers and code size
Agenda

• Control Hazards
• Higher Level ILP
• **Administrivia**
  • Dynamic Scheduling
  • Technology Break
  • Example AMD Barcelona
• Big Picture: Types of Parallelism
• Summary
Administrivia

• Project 2 Part 2 due Sunday.
  – Slides at end of July 12 lecture contain useful info.

• Lab 12 cancelled!
  – Replaced with free study session where you can catch up on labs / work on project 2.
  – The TAs will still be there.

• Project 3 will be posted late Sunday (7/31)
  – Two-stage pipelined CPU in Logisim
Agenda

• Control Hazards
• Administrivia
• Higher Level ILP
• Dynamic Scheduling
• Technology Break
• Example AMD Barcelona
• Big Picture: Types of Parallelism
• Summary
Dynamic Multiple Issue

- "Superscalar" processors
- CPU decides whether to issue 0, 1, 2, ... instructions each cycle
  - Avoiding structural and data hazards
- Avoids need for compiler scheduling
  - Though it may still help
  - Code semantics ensured by the CPU
Dynamic Pipeline Scheduling

• Allow the CPU to execute instructions *out of order* to avoid stalls
  – But commit result to registers in order

• Example

```assembly
lw $t0, 20($s2)
addu $t1, $t0, $t2
subu $s4, $s4, $t3
slti $t5, $s4, 20
```

– Can start `subu` while `addu` is waiting for `lw`
Why Do Dynamic Scheduling?

• Why not just let the compiler schedule code?
• Not all stalls are prediciable
  – e.g., cache misses
• Can’t always schedule around branches
  – Branch outcome is dynamically determined
• Different implementations of an ISA have different latencies and hazards
Speculation

• “Guess” what to do with an instruction
  – Start operation as soon as possible
  – Check whether guess was right
    • If so, complete the operation
    • If not, roll-back and do the right thing

• Common to both static and dynamic multiple issue

• Examples
  – Speculate on branch outcome (Branch Prediction)
    • Roll back if path taken is different
  – Speculate on load
    • Roll back if location is updated
Pipeline Hazard: Matching socks in later load

A depends on D; stall since folder tied up;

7/28/2011  Summer 2011 -- Lecture #23  26
Out-of-Order Laundry: Don’t Wait

- A depends on D; rest continue; need more resources to allow out-of-order
Out-of-Order Execution (1/3)

Basically, **unroll loops in hardware**

1. Fetch instructions in program order (≤4/clock)

2. Predict branches as taken/untaken

3. To avoid hazards on registers, *rename registers* using a set of internal registers (~80 registers)
Out-of-Order Execution (2/3)

Basically, **unroll loops in hardware**

4. Collection of renamed instructions might execute in a *window* (~60 instructions)

5. Execute instructions with ready operands in 1 of multiple *functional units* (ALUs, FPUs, Ld/St)

6. Buffer results of executed instructions until predicted branches are resolved in *reorder buffer*
Out-of-Order Execution (3/3)

Basically, **unroll loops in hardware**

7. If predicted branch correctly, $commit$ results in program order

8. If predicted branch incorrectly, discard all dependent results and start with correct PC
Dynamically Scheduled CPU

Branch prediction, Register renaming

Instruction fetch and decode unit

Reservation station

Reservation station

Reservation station

Reservation station

Functional units

Integer

Integer

Floating point

Load-store

Execute...

... and Hold

Reorder buffer for register and memory writes

In-order issue

Preserves dependencies

Wait here until all operands available

Out-of-order execute

Results also sent to any waiting reservation stations

In-order commit

Can supply operands for issued instructions
Out-Of-Order Intel

• All use O-O-O since 2001

<table>
<thead>
<tr>
<th>Microprocessor</th>
<th>Year</th>
<th>Clock Rate</th>
<th>Pipeline Stages</th>
<th>Issue width</th>
<th>Out-of-order/Speculation</th>
<th>Cores</th>
<th>Power</th>
</tr>
</thead>
<tbody>
<tr>
<td>i486</td>
<td>1989</td>
<td>25MHz</td>
<td>5</td>
<td>1</td>
<td>No</td>
<td>1</td>
<td>5W</td>
</tr>
<tr>
<td>Pentium</td>
<td>1993</td>
<td>66MHz</td>
<td>5</td>
<td>2</td>
<td>No</td>
<td>1</td>
<td>10W</td>
</tr>
<tr>
<td>Pentium Pro</td>
<td>1997</td>
<td>200MHz</td>
<td>10</td>
<td>3</td>
<td>Yes</td>
<td>1</td>
<td>29W</td>
</tr>
<tr>
<td>P4 Willamette</td>
<td>2001</td>
<td>2000MHz</td>
<td>22</td>
<td>3</td>
<td>Yes</td>
<td>1</td>
<td>75W</td>
</tr>
<tr>
<td>P4 Prescott</td>
<td>2004</td>
<td>3600MHz</td>
<td>31</td>
<td>3</td>
<td>Yes</td>
<td>1</td>
<td>103W</td>
</tr>
<tr>
<td>Core</td>
<td>2006</td>
<td>2930MHz</td>
<td>14</td>
<td>4</td>
<td>Yes</td>
<td>2</td>
<td>75W</td>
</tr>
<tr>
<td>Core 2 Yorkfield</td>
<td>2008</td>
<td>2930 MHz</td>
<td>16</td>
<td>4</td>
<td>Yes</td>
<td>4</td>
<td>95W</td>
</tr>
<tr>
<td>Core i7 Gulftown</td>
<td>2010</td>
<td>3460 MHz</td>
<td>16</td>
<td>4</td>
<td>Yes</td>
<td>6</td>
<td>130W</td>
</tr>
</tbody>
</table>
Agenda

- Control Hazards
- Administrivia
- Higher Level ILP
- Dynamic Scheduling
- **Technology Break**
- Example AMD Barcelona
- Big Picture: Types of Parallelism
- Summary
Agenda

• Control Hazards
• Administrivia
• Higher Level ILP
• Dynamic Scheduling
• Technology Break
• Example AMD Barcelona
• Big Picture: Types of Parallelism
• Summary
AMD Opteron X4 Microarchitecture

Queues:
- 106 RISC ops
- 24 integer ops
- 36 FP/SSE ops
- 44 ld/st
AMD Opteron X4 Pipeline Flow

• For integer operations

  ▪ 12 stages (Floating Point is 17 stages)
  ▪ Up to 106 RISC-ops in progress

• Intel Nehalem is 16 stages for integer operations, details not revealed, but likely similar to above.
  ▪ Intel calls RISC operations “Micro operations” or “μops”
Does Multiple Issue Work?

The BIG Picture

- Yes, but not as much as we’d like
- Programs have real dependencies that limit ILP
- Some dependencies are hard to eliminate
  - e.g., pointer aliasing
- Some parallelism is hard to expose
  - Limited window size during instruction issue
- Memory delays and limited bandwidth
  - Hard to keep pipelines full
- Speculation can help if done well
Agenda

- Higher Level ILP
- Administrivia
- Dynamic Scheduling
- Example AMD Barcelona
- Technology Break
- Big Picture: Types of Parallelism
- Summary
New-School Machine Structures
(It’s a bit more complicated!)

- **Parallel Requests**
  Assigned to computer
  e.g., Search “Katz”

- **Parallel Threads**
  Assigned to core
  e.g., Lookup, Ads

- **Parallel Instructions**
  >1 instruction @ one time
  e.g., 5 pipelined instructions

- **Parallel Data**
  >1 data item @ one time
  e.g., Add of 4 pairs of words

- **Hardware descriptions**
  All gates functioning in parallel at same time

---

**Software**

**Hardware**

*Harness Parallelism & Achieve High Performance*

Project 1
- Core
- Memory
- Input/Output
- Instruction Unit(s)
- Main Memory
- (Cache)

Project 2
- Core
- Functional Unit(s)

Logic Gates Project 3

Smart Phone

Warehouse Scale Computer

Lab 14

7/28/2011
Big Picture on Parallelism

Two types of parallelism in *applications*

1. *Data-Level Parallelism (DLP)*: arises because there are many data items that can be operated on at the same time

2. *Task-Level Parallelism*: arises because tasks of work are created that can operate largely in parallel
Big Picture on Parallelism

Hardware can exploit app DLP and Task LP in four ways:

1. **Instruction-Level Parallelism**: Hardware exploits application DLP using ideas like pipelining and speculative execution.

2. **SIMD architectures**: exploit app DLP by applying a single instruction to a collection of data in parallel.

3. **Thread-Level Parallelism**: exploits either app DLP or TLP in a tightly-coupled hardware model that allows for interaction among parallel threads.

4. **Request-Level Parallelism**: exploits parallelism among largely decoupled tasks and is specified by the programmer of the operating system.
Peer Instruction

Instr LP, SIMD, Thread LP, Request LP are examples of

- Parallelism *above* the Instruction Set Architecture
- Parallelism explicitly *at* the level of the ISA
- Parallelism *below* the level of the ISA

<table>
<thead>
<tr>
<th>Inst. LP</th>
<th>SIMD</th>
<th>Thr. LP</th>
<th>Req. LP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Red</td>
<td>=</td>
<td>=</td>
<td>=</td>
</tr>
<tr>
<td>White</td>
<td>V</td>
<td>=</td>
<td>=</td>
</tr>
<tr>
<td>Green</td>
<td>=</td>
<td>=</td>
<td>∧</td>
</tr>
<tr>
<td>Yellow</td>
<td>V</td>
<td>=</td>
<td>∧</td>
</tr>
<tr>
<td>Purple</td>
<td>=</td>
<td>∧</td>
<td>∧</td>
</tr>
<tr>
<td>Blue</td>
<td>∧</td>
<td>∧</td>
<td>∧</td>
</tr>
</tbody>
</table>
Peer Answer

Instr LP, SIMD, Thread LP, Request LP are examples of:

- Parallelism *above* the Instruction Set Architecture
- Parallelism explicitly *at* the level of the ISA
- Parallelism *below* the level of the ISA

<table>
<thead>
<tr>
<th>Inst. LP</th>
<th>SIMD</th>
<th>Thr. LP</th>
<th>Req. LP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Red</td>
<td>=</td>
<td>=</td>
<td>=</td>
</tr>
<tr>
<td>White</td>
<td>∨</td>
<td>=</td>
<td>=</td>
</tr>
<tr>
<td>Green</td>
<td>=</td>
<td>=</td>
<td>∧</td>
</tr>
<tr>
<td>Yellow</td>
<td>∨</td>
<td>=</td>
<td>∧</td>
</tr>
<tr>
<td>Purple</td>
<td>=</td>
<td>∧</td>
<td>∧</td>
</tr>
<tr>
<td>Blue</td>
<td>∧</td>
<td>∧</td>
<td>∧</td>
</tr>
</tbody>
</table>
Peer Question

State if the following techniques are associated primarily with a software- or hardware-based approach to exploiting ILP (in some cases, the answer may be both): Superscalar, Out-of-Order execution, Speculation, Register Renaming

<table>
<thead>
<tr>
<th></th>
<th>Superscalar</th>
<th>Out of Order</th>
<th>Speculation</th>
<th>Register Renaming</th>
</tr>
</thead>
<tbody>
<tr>
<td>Red</td>
<td>HW</td>
<td>HW</td>
<td>HW</td>
<td>HW</td>
</tr>
<tr>
<td>White</td>
<td>SW</td>
<td>SW</td>
<td>SW</td>
<td>SW</td>
</tr>
<tr>
<td>Green</td>
<td>Both</td>
<td>Both</td>
<td>Both</td>
<td>Both</td>
</tr>
<tr>
<td>Yellow</td>
<td>HW</td>
<td>HW</td>
<td>Both</td>
<td>Both</td>
</tr>
<tr>
<td>Purple</td>
<td>HW</td>
<td>HW</td>
<td>HW</td>
<td>Both</td>
</tr>
<tr>
<td>Blue</td>
<td>HW</td>
<td>HW</td>
<td>HW</td>
<td>SW</td>
</tr>
</tbody>
</table>
Peer Answer

State if the following techniques are associated primarily with a software- or hardware-based approach to exploiting ILP (in some cases, the answer may be both): Superscalar, Out-of-Order execution, Speculation, Register Renaming

<table>
<thead>
<tr>
<th></th>
<th>Superscalar</th>
<th>Out of Order</th>
<th>Speculation</th>
<th>Register Renaming</th>
</tr>
</thead>
<tbody>
<tr>
<td>Red</td>
<td>HW</td>
<td>HW</td>
<td>HW</td>
<td>HW</td>
</tr>
<tr>
<td>Orange</td>
<td>SW</td>
<td>SW</td>
<td>SW</td>
<td>SW</td>
</tr>
<tr>
<td>Green</td>
<td>Both</td>
<td>Both</td>
<td>Both</td>
<td>Both</td>
</tr>
<tr>
<td><strong>Yellow</strong></td>
<td><strong>HW</strong></td>
<td><strong>HW</strong></td>
<td><strong>Both</strong></td>
<td><strong>Both</strong></td>
</tr>
<tr>
<td>Pink</td>
<td>HW</td>
<td>HW</td>
<td>HW</td>
<td>Both</td>
</tr>
<tr>
<td>Blue</td>
<td>HW</td>
<td>HW</td>
<td>HW</td>
<td>SW</td>
</tr>
</tbody>
</table>
“And in Conclusion, ...”

• Big Ideas of Instruction Level Parallelism
• Pipelining, Hazards, and Stalls
• Forwarding, Speculation to overcome Hazards
• Multiple issue to increase performance
  – IPC instead of CPI
• Dynamic Execution: Superscalar in-order issue, branch prediction, register renaming, out-of-order execution, in-order commit
  – “unroll loops in HW”, hide cache misses
But wait... there's more???

If we have time...
Review: C Memory Management

• C has three pools of data memory (+ code memory)
  – Static storage: global variable storage, basically permanent, entire program run
  – The Stack: local variable storage, parameters, return address
  – *The Heap (dynamic storage):*
    
    ```c
    malloc() grabs space from here, free() returns it
    ```

• Common (Dynamic) Memory Problems
  – Using uninitialized values
  – Accessing memory beyond your allocated region
  – Improper use of free/realloc by messing with the pointer handle returned by malloc
  – Memory leaks: mismatched malloc/free pairs

OS prevents accesses between stack and heap (via virtual memory)
Simplest Model

• Only one program running on the computer
  – Addresses in the program are exactly the physical memory addresses

• Extensions to the simple model:
  – What if less physical memory than full address space?
  – What if we want to run multiple programs at the same time?
Problem #1: Physical Memory Less Than the Full Address Space

- One architecture, many implementations, with possibly different amounts of memory
- Memory used to very expensive and physically bulky
- Where does the stack grow from then?
Idea: Level of Indirection to Create Illusion of Large Physical Memory

- Hi Order Bits of Virtual Address
- Address Map or Table
- Hi Order Bits of Physical Address

- Stack
- Heap
- Static Data
- Code

- "Logical" "Virtual"
- Virtual "Page" Address
- Physical "Page" Address
- Real
Problem #2: Multiple Programs
Sharing the Machine’s Address Space

• How can we run multiple programs without accidentally stepping on same addresses?

• How can we protect programs from clobbering each other?
Idea: Level of Indirection to Create Illusion of Separate Address Spaces

One table per running application OR swap table contents when switching
Extension to the Simple Model

• Multiple programs sharing the same address space
  – E.g., Operating system uses low end of address range
    shared with application
  – Multiple programs in shared (virtual) address space
    • Static management: fixed partitioning/allocation of space
    • Dynamic management: programs come and go, take different
      amount of time to execute, use different amounts of memory
• How can we protect programs from clobbering each other?
• How can we allocate memory to applications on demand?
Static Division of Shared Address Space

- E.g., how to manage the carving up of the address space among OS and applications?
- Where does the OS end and the application begin?
- Dynamic management, with protection, would be better!
First Idea: Base + Bounds Registers for Location Independence

Location-independent programs
Programming and storage management ease:
need for a base register

Protection
Independent programs should not affect each other inadvertently: need for a bound register

Historically, base + bounds registers were a very early idea in computer architecture
Simple Base and Bound Translation

Base and bounds registers are visible/accessible to programmer. 

*Trap* to OS if bounds violation detected ("seg fault"/"core dumped")
Why do we want to run multiple programs?

*Run others while waiting for I/O*

What prevents programs from accessing each other’s data?
Restriction on Base + Bounds Regs

Want only the Operating System to be able to change Base and Bound Registers

Processors need different execution modes

1. **User mode**: can use Base and Bound Registers, but cannot change them

2. **Supervisor mode**: can use and change Base and Bound Registers
   - Also need Mode Bit (0=User, 1=Supervisor) to determine processor mode
   - Also need way for program in User Mode to invoke operating system in Supervisor Mode, and vice versa
As programs come and go, the storage is “fragmented”. Therefore, at some stage programs have to be moved around to compact the storage. Easy way to do this?