G22.2243-001
High Performance Computer Architecture

Lecture 2
Instruction Set Architecture
Pipelining
From Last Week: Amdahl's Law

- Speedup due to enhancement E:

\[
\text{Speedup (E)} = \frac{\text{Execution time without E}}{\text{Execution time with E}} = \frac{\text{Performance with E}}{\text{Performance without E}}
\]

- Suppose that enhancement E accelerates a fraction \( f \) of the task by a factor \( s \), and the remainder of the task is unaffected.

- New execution time and the overall speedup?

\[
\text{Exec. time}_{\text{new}} = \text{Exec. time}_{\text{old}} \times [ (1 - f) + f/s ]
\]

\[
\text{Speedup (E)} = \frac{\text{Exec. Time}_{\text{old}}}{\text{Exec. Time}_{\text{new}}} = \frac{1}{[1 - f + f/s]} \leq \frac{1}{(1 - f)}
\]
Example of Amdahl’s Law

- Floating point instructions improved to run 2x; but only 10% of the time was spent on these instructions
- How much improvement in performance should one expect?

\[
\text{Exec. time}_{\text{new}} = \text{Exec. time}_{\text{old}} \times [(1 - f) + f/s]
\]

\[
\text{Speedup (E)} = \frac{\text{Exec. Time}_{\text{old}}}{\text{Exec. Time}_{\text{new}}} = \frac{1}{1 - f + f/s} \leq \frac{1}{1 - f}
\]

\[
\text{Exec. time}_{\text{new}} = \text{Exec. time}_{\text{old}} \times [(1 - 0.1) + 0.1/2] = \text{Exec. time}_{\text{old}} \times 0.95
\]

\[
\text{Speedup (E)} = \frac{\text{Exec. Time}_{\text{old}}}{\text{Exec. Time}_{\text{new}}} = \frac{1}{0.95} = 1.053
\]

- The new machine is 5.3% faster for this mix of instructions
Recap

• The most accurate measure of performance is the execution time of representative real programs or collections of programs (benchmarks)
• Computer Architecture is an iterative process:
  – Identify bottlenecks and search the possible design space
  – Innovate and make selections
  – Implement/simulate selections made and evaluate the impact
• Make the common case fast (recall the Amdahl’s Law)
• RISC: Reduced Instruction Set Computers
  • Identify most frequently-used instructions
    – Implement them in hardware
  • Emulate other instructions (slowly) in software
    – Pretty much every technique used in current-day microprocessors
Outline

• Instruction set principles
  – What is an instruction set?
  – What is a good instruction set?
  – Instruction set aspects
  – RISC vs. CISC

Instruction set examples: Appendix B

• Pipelining
  – Why pipelining?
  – Basic stages of a pipeline
  – Expected improvement
  – Complications?

[Hennessy/Patterson CA:AQA (3rd Edition): Appendix B & Appendix A]
Instruction Set Principles

Instruction Set Architecture (ISA)

“Instruction set architecture is the structure of a computer that a machine language programmer must understand to write a correct (timing independent) program for that machine.”

Source: IBM in 1964 when introducing the IBM 360 architecture

• An instruction set is a functional description of the processor
  – What operations can it do
  – What storage mechanisms does it support

• ISA defines the hardware/software interface

A good interface:
  – Lasts through many implementations
  – Can be used in many different ways
  – Provides convenient functionality to higher levels
  – Permits an efficient implementation at lower levels
Instruction Set Design Issues

- What operations are supported?
  - add, sub, mul, move, compare . . .

- Where are operands stored?
  - Registers (how many of them are there), memory, stack, accumulator

- How many explicit operands are there?
  - 0, 1, 2, or 3

- How is the operand location specified?
  - register, immediate, indirect, . . .

- What type and size of operands are supported?
  - byte, int, float, double, string, vector, . . .
A "Typical" RISC

- 32-bit fixed format instruction (3 formats)
- Memory access only through load/store operations
- 32 32-bit general-purpose registers
  - R0 contains zero
  - Double precision operations take pair (floating point registers may be separate)
- 3-address (src1, src2, dst), register-register arithmetic instructions
- Single address mode for load/store: base + displacement
  - no indirection
- Simple branch conditions
- Delayed branch

- Examples:
  SUN SPARC, MIPS, HP PA-RISC, DEC Alpha, IBM PowerPC, CDC 6600, CDC 7600, Cray-1, Cray-2, Cray-3
**Example: MIPS**

### Register-Register

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21 20</th>
<th>16 15</th>
<th>11 10</th>
<th>6 5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Op</td>
<td>Rs1</td>
<td>Rs2</td>
<td>Rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Register-Immediate (e.g., load/store)

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21 20</th>
<th>16 15</th>
<th></th>
<th></th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Op</td>
<td>Rs1</td>
<td>Rd</td>
<td></td>
<td></td>
<td>immediate</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Branch

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21 20</th>
<th>16 15</th>
<th></th>
<th></th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Op</td>
<td>Rs1</td>
<td>Rs2</td>
<td>Opx</td>
<td></td>
<td>immediate</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Jump / Call

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Op</td>
<td></td>
<td></td>
<td>PC-region target address</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
ISA Metrics

- Orthogonality
  - No special registers, few special cases, all operand modes available with any data type or instruction type

- Completeness
  - Support for a wide range of operations and target applications

- Regularity
  - No overloading for the meanings of instruction fields

- Streamlined
  - Resource needs easily determined

- Ease of compilation (or assembly language programming)

- Ease of implementation
Closer look at ISA Aspects

• Operand location
• Addressing modes
• Types of instructions
ISA Aspect (1): Operand Location

Accumulator (before 1960):
1 address add A acc ← acc + mem[A]

Stack (1960s to 1970s):
0 address add tos ← tos + next

Memory-Memory (1970s to 1980s):
2 address add A, B mem[A] ← mem[A] + mem[B]
3 address add A, B, C mem[A] ← mem[B] + mem[C]

Register-Memory (1970s to present):
2 address add R1, A R1 ← R1 + mem[A]
load R1, A R1 ← mem[A]

Register-Register, also called Load/Store (1960s to present):
3 address add R1, R2, R3 R1 ← R2 + R3
load R1, R2 R1 ← mem[R2]
store R1, R2 mem[R1] ← R2
Choices for Operand Location

• Running example: C := A + B

• **Accumulator**

```c
load A    accum = M[A];
add B     accum += M[B];
store C   M[C] = accum;
```

  + Less hardware, code density
  – Memory bottleneck

• **Stack**

```c
push A    S[++tos] = M[A];
push B    S[++tos] = M[B]
add        t1= S[tos--]; t2= S[tos--]; S[++tos]= t1 + t2;
pop C     M[C] = S[tos--];
```

  + Less hardware, code density
  – Memory, pipelining bottlenecks
  – x86 uses stack model for floating point computations
Choices for Operand Location (cont’d)

• Running example: \( C := A + B \)

• **Memory-Memory**

  \[
  \text{add } C, A, B \quad M[C] = M[A] + M[B];
  \]

  + Code density (most compact)
  - Memory bottleneck
  - No current machines support memory-memory (VAX did)

• **Memory-Register**

  \[
  \text{load } R1, A \quad R1 = M[A];
  \text{add } R1, B \quad R1 += M[B];
  \text{store } C, R1 \quad M[C] = R1;
  \]

  + Like several explicit (extended) accumulators
  + Code density, easy to decode
  - Asymmetric operands, different amount of work per instruction
  - Examples: IBM 360/370, x86, Motorola 68K
Choices for Operand Location (cont’d)

• Running example: \( C := A + B \)
• Register-Register (Load-Store)

\[
\begin{align*}
\text{load } & R1, A & R1 &= M[A]; \\
\text{load } & R2, B & R2 &= M[B]; \\
\text{add } & R3, R1, R2 & R3 &= R1 + R2; \\
\text{store } & C, R3 & M[C] &= R3;
\end{align*}
\]

+ Easy decoding, operand symmetry
+ Deterministic cost for ALU operations (simple cost model)
+ Scheduling opportunities
− Code density
Operand Location: Registers vs. Memory

• Pros and cons of registers
  + Faster, direct access
  + Simple cost model (fixed latency, no misses)
  + Short identifier
    – Must save/restore on procedure calls, context switches
    – Fixed size (larger-sized structures must live in memory)

• Pros and cons of more registers
  + Possible to keep more operands for longer in faster memory
    • Shorter operand access time, lower memory traffic
    – Longer specifiers
    – Larger cost for saving CPU state
    – Trend towards more registers
      • 8 (x86) -> 32 (MIPS/Alpha/PPC) -> 128 (IA-64)
      • Driven by increasing compiler involvement in scheduling
Closer look at ISA Aspects

• Operand location
• Addressing modes
• Types of instructions
ISA Aspect (2): Addressing

- **Endian-ness**: Order of bytes in words
  - **Big**: byte at lowest address has the most significance (big end)
    - E.g., IBM, Sun SPARC
  - **Little**: bytes at lower address have lower significance (little end)
    - E.g., x86
  - Some processors allow mode to be selectable
    - E.g., PowerPC, MIPS (new implementations of the ISA)

- **Alignment**: Natural boundaries defined by architecture
  - Aligned address: (address % size) equals 0

- Different ISAs support different restrictions on alignment
  - **None**: (all alignments supported by hardware): expensive/exception handling
  - **Restricted**: misaligned access traps to software
  - **Middle ground**: misaligned data okay, but requires multiple instructions
    - E.g., MIPS: `lwl/lwr` (Load Word Left/Right: only modify part of register)
### Types of Addressing Modes (VAX)

<table>
<thead>
<tr>
<th>Mode</th>
<th>Example</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Register direct</td>
<td>add R4, R3</td>
<td>R4 ← R4 + R3</td>
</tr>
<tr>
<td>2. Immediate</td>
<td>add R4, #3</td>
<td>R4 ← R4 + 3</td>
</tr>
<tr>
<td>3. Displacement</td>
<td>add R4, 100(R1)</td>
<td>R4 ← R4 + M[100+R1]</td>
</tr>
<tr>
<td>4. Register indirect</td>
<td>add R4, (R1)</td>
<td>R4 ← R4 + M[R1]</td>
</tr>
<tr>
<td>5. Indexed</td>
<td>add R4, (R1 + R2)</td>
<td>R4 ← R4 + M[R1 + R2]</td>
</tr>
<tr>
<td>6. Direct</td>
<td>add R4, (1000)</td>
<td>R4 ← R4 + M[1000]</td>
</tr>
<tr>
<td>7. Memory Indirect</td>
<td>add R4, @(R3)</td>
<td>R4 ← R4 + M[M[R3]]</td>
</tr>
<tr>
<td>8. Autoincrement</td>
<td>add R4, (R2)+</td>
<td>R4 ← R4 + M[R2]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>R2 ← R2 + d</td>
</tr>
<tr>
<td>9. Autodecrement</td>
<td>add R4, (R2)−</td>
<td>R4 ← R4 + M[R2]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>R2 ← R2 − d</td>
</tr>
<tr>
<td>10. Scaled</td>
<td>add R4, 100(R2)[R3]</td>
<td>R4 ← R4 + M[100 + R2 + R3*d]</td>
</tr>
</tbody>
</table>
Which modes are actually used?

- Study by Clark and Emer
  - Modes 1–4 account for 93% of all VAX operands

- Register mode responsible for roughly half

- Memory modes
  - An example of making the common case fast
    - RISC machines typically implement register, immediate, and displacement
    - Synthesize all other modes in software
Addressing Modes for Signal Processing

Two additional modes, motivated by DSP applications

- **Modulo or circular addressing**
  - DSPs deal with large (infinite, continuous) streams of data
  - Typically encoded as a circular buffer (mode allows auto-reset)
  - Keeps start and end registers with each address register
  - Allows autoincrement/autodecrement modes to reset the address

- **Bit reverse addressing**
  - Permit permutations on address (to support kernels such as FFT)
  - E.g., address/displ. (100) results in access to address/displ. (001)

- Used in assembly implementations of some libraries

- Renewed importance of autoincrement and autodecrement modes
  - Study on TI TMS320C54x DSP finds autoincrement mode use of ~18.8%
Closer look at ISA Aspects

- Operand location
- Addressing modes
- Types of instructions
### ISA Aspect (3): Types of Operations

<table>
<thead>
<tr>
<th>Type</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arithmetic and Logic:</td>
<td>AND, ADD</td>
</tr>
<tr>
<td>Data Transfer:</td>
<td>MOVE, LOAD, STORE</td>
</tr>
<tr>
<td>Control</td>
<td>BRANCH, JUMP, CALL</td>
</tr>
<tr>
<td>System</td>
<td>OS CALL, VM</td>
</tr>
<tr>
<td>Floating Point</td>
<td>ADDF, MULF, DIVF</td>
</tr>
<tr>
<td>Decimal</td>
<td>ADDD, CONVERT</td>
</tr>
<tr>
<td>String</td>
<td>MOVE, COMPARE</td>
</tr>
<tr>
<td>Graphics</td>
<td>(DE)COMPRESS</td>
</tr>
</tbody>
</table>
Relative Frequency of Instructions

- For the 80x86, averaged over five SPECint92 programs

<table>
<thead>
<tr>
<th>Rank</th>
<th>Instruction</th>
<th>Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>load</td>
<td>22%</td>
</tr>
<tr>
<td>2</td>
<td>branch</td>
<td>20%</td>
</tr>
<tr>
<td>3</td>
<td>compare</td>
<td>16%</td>
</tr>
<tr>
<td>4</td>
<td>store</td>
<td>12%</td>
</tr>
<tr>
<td>5</td>
<td>add</td>
<td>8%</td>
</tr>
<tr>
<td>6</td>
<td>and</td>
<td>6%</td>
</tr>
<tr>
<td>7</td>
<td>sub</td>
<td>5%</td>
</tr>
<tr>
<td>8</td>
<td>register move</td>
<td>4%</td>
</tr>
<tr>
<td>9</td>
<td>call</td>
<td>1%</td>
</tr>
<tr>
<td>10</td>
<td>return</td>
<td>1%</td>
</tr>
<tr>
<td>Total</td>
<td></td>
<td>96%</td>
</tr>
</tbody>
</table>

- Simple instructions dominate
Control Instructions

• Instructions that change the value of the PC

• Three kinds of instructions
  – (conditional) branches, (unconditional) jumps
  – Function calls, function returns
  – System calls, system returns

• Questions
  – What kinds of conditions are supported? How are the conditions set?
  – How is the target address specified?
  – For function and system calls, how is the return address specified?
    • Most recent processors use an implicit register
      + Simple scheme
      – Software needs to save/restore register contents
  – Which instructions save/restore CPU state?
Control Instructions (1): Conditions

- **Compare and branch**
  - Single instruction branches
    - Needs ALU stage for branch instructions as well; May be too much work for pipelined execution
- **Condition codes** (e.g., zero, negative, overflow)
  - Condition code set by ALU operations
    - Sometimes set for free
      - Part of the CPU state, needs to be saved and restored
      - Constrains the ordering of instructions
    - “compare” instruction can be used
- **Recent processors** (e.g., Alpha, IA-64) offer **predicated** instructions
  - combine comparison, branch, and ALU operations
  - more about these later in the course
Control Instructions (2): Target Address

Four choices

• **PC-relative** with immediate
  + Position independent, all info present for computing target
  + Short immediate sufficient (compact instructions)
    – Target must be known statically
    *Uses*: branches/jumps within function

• **Arbitrary** immediate
  *Uses*: function calls

• **Register**
  + Short specifier, target can be dynamic
    – Extra instruction to load register
    *Uses*: indirect calls (e.g., DLLs, virtual functions), returns, switches

• ** Vectored** traps
  + Protection (hence, heavyweight)
    *Uses*: system calls
Control Instructions (3): Save and Restore State

- Only call (function and system) instructions need to save state
  - Function calls: save registers
  - System calls: save registers, flags, PC, PSW, ...

Two choices
- **Software** saving/restoring
  - Calling convention distinguishes between caller- and callee-save registers
- **Hardware** saving/restoring
  - Explicit instructions: VAX
  - Implicit: SPARC register windows
    - 32 registers: 8 input, 8 output, 8 local, 8 global
    - On call: 8 output of caller become 8 input of callee
      - local/output registers are new set (no need to save/restore)
      - No save/restore on shallow call graphs
      - Makes advanced architectural techniques (e.g., register renaming) hard
Operations for Media and Signal Processing

- Results of media processing gauged in terms of human perception
  - Narrow data items (8-16 bits) as opposed to 32 or 64-bit words
  - Lower precision requirements
  - Real-time requirements → cannot cause overflow traps

- Several ISAs have been extended to support graphics and multimedia
  - Intel: MMX, Streaming SIMD Extensions (SSE, SSE2, and SSE3)
  - AMD: 3DNow!
  - Sun: Visual Instruction Set (VIS)
  - Motorola and IBM: AltiVec

- Common theme
  - Partitioned operations (SIMD), pack/unpack operations
  - Multiply-accumulate (MAC) instructions (dot products and vector multiplies)

- Not useful for most programs
The RISC vs. CISC Debate

• Early 80s: Several projects challenged how processors were being built
  – Berkeley RISC-I (Patterson), Stanford MIPS (Hennessy), IBM 801
  – Contrasted their design, RISC (Reduced Instruction Set Computer) with what began to be called CISC (Complex Instruction Set Computer)

• RISC argument
  – CISC is too complex to ever be implemented well
    • too many addressing modes, variable format instructions, many multi-cycle operations, microcoding, hand-assembled programs
  – RISC characterized by
    • Single-cycle operation
    • Hardwired control
    • Load/store organization
    • Fixed instruction format
    • Few modes
    • Reliance on compiler optimization

Motivated by quantitative studies of program behavior
Focus on optimizing the common case
The Reality of RISC vs. CISC

- RISC does help compiler optimizations
  - Load/store architecture supports register allocation
    - Explicit choices of what values reside where in the memory hierarchy
  - Simple instructions make instruction selection, optimization easier

- However, CISC does not have any fundamental implementation flaws
  - Fixable with more transistors
    - Good CISC pipeline can be constructed with modest increase in transistors
    - Not an issue given Moore’s law

- Most commercially successful ISA is x86 (CISC)
  - Current-day Pentium processors translate CISC instructions into sequences of RISC micro-ops
    - Internal microarchitecture is actually RISC
  - Better substrate for implementing advanced architectural techniques
Announcements

- Assignment 1 out today; Due date: By next class (Sep. 19th @ 5pm)
Pipelining
Computer Pipelines

- Computers execute billions of instructions, so instruction throughput is what matters

Main idea behind pipelining
- Divide instruction execution across several stages
  - each stage accesses only a subset of the CPU’s resources
- Example: Classic 5-stage RISC pipeline
  
  IF ➔ ID ➔ EX ➔ MEM ➔ WB

- Simultaneously have different instructions in different stages
  - Ideally, can issue a new instruction every cycle
  - Cycle time determined by longest stage

- Pipelining only improves throughput, not latency
  - Each instruction still needs to go through each stage
Some key properties of RISC architectures simplify implementation:
- All instructions same length
- Registers located in same place in instruction format
- Memory operands only in loads or stores

E.g., MIPS

**Register-Register (R-type)**
ADD R1, R2, R3

```
31 26 25 21 20 16 15 11 10 6 5 0
Op rs1 rs2 rd func
```

**Register-Immediate (I-type)**
SUB R1, R2, #3

```
31 26 25 21 20 16 15 0
Op rs1 rd immediate
```

**Jump / Call (J-type)**
JUMP end

```
31 26 25 0
Op PC-region target address
```

(jump, jump and link, trap and return from exception)
Unpipelined Implementation of a RISC ISA

- Most MIPS instructions can be implemented in 5 cycles
Unpipelined Implementation (1) – Instruction Fetch

Instruction Fetch

Next PC

Adder

Memory

Next SEQ PC

Instr. Decode

Reg. Fetch

Execute

Addr. Calc

Memory Access

Write Back

Load instruction

Update program counter

Next PC

Reg File

Data Memory

Zero?

ALU

memory

LD

MEM

WB Data
Unpipelined Implementation (2) – Instruction Decode

Instruction Fetch

Instr. Decode
Reg. Fetch

Execute
Addr. Calc

Memory
Access

Write
Back

Next PC

Adder

Memory

Next SEQ PC

Reg File

alu

Memory

Write
Back

Fetch source registers
Sign extend immediate field
Unpipelined Implementation (3) – Execute

Instruction Fetch

Instruction Decode
Reg. Fetch

Execute
Addr. Calc

Memory Access

Write Back

Next PC

Adder

Next SEQ PC

Memory

Reg File

LD/ST: Compute effective address
ALU operations
Branch target/condition evaluation

Data Memory

Zero?

Reg. Fetch

RS1

RS2

RD

Instr. Decode

Imm

RD

Sign Extend

Instr. Fetch
Unpipelined Implementation (4) – Memory Access

Instruction Fetch

Instr. Decode
Reg. Fetch

Execute
Addr. Calc

Memory
Access

Write
Back

Next PC

Adder

Memory

Next SEQ PC

Reg File

ALU

LD/ST

Branch: Set PC value

PC

Next PC

4

RS1

RS2

RD

Imm

Sign
Extend

Zero?

Data
Memory

LMD

MUX
Unpipelined Implementation (4) – Write Back

Instruction Fetch

Next PC

Adder

Memory

Instruction Decode Reg. Fetch

Next SEQ PC

RS1

RS2

RD

Reg File

Zero?

Execute Addr. Calc

Update register file for LD, ALU ops

Next PC

Instruction Fetch

Reg File

Sign Extend

ALU

Data Memory

Write Back

Update register file for LD, ALU ops

Update register file for LD, ALU ops
CPI for the Multiple-Cycle RISC Implementation

• Branches and stores: 4 cycles (no WB), all other instructions: 5 cycles
  – If 20% of the instructions are branches or loads
    \[ CPI = 0.8 \times 5 + 0.2 \times 4 = 4.80 \]
• ALU operations can be allowed to complete in 4 cycles (no MEM)
  – If 40% of the instructions are ALU operations
    \[ CPI = 0.4 \times 5 + 0.6 \times 4 = 4.40 \]
• Pipelining the implementation can help reduce the CPI

<table>
<thead>
<tr>
<th>Instr.</th>
<th>Clock Number</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td>i</td>
<td>IF</td>
</tr>
<tr>
<td>i+1</td>
<td>IF</td>
</tr>
<tr>
<td>i+2</td>
<td>IF</td>
</tr>
<tr>
<td>i+3</td>
<td>IF</td>
</tr>
<tr>
<td>i+4</td>
<td>IF</td>
</tr>
</tbody>
</table>
Pipelined Implementation of a RISC ISA
Pipelined Implementation of a RISC ISA (1): Separation of Instruction and Data Memories

Eliminates conflict for a single memory
Register file accessed in two stages: Writes in first half, Reads in second half
Visualizing Pipelining

Time (clock cycles)

Cycle 1  Cycle 2  Cycle 3  Cycle 4  Cycle 5  Cycle 6  Cycle 7

Instr. order
Speedup from Pipelining

• Assume that a multiple cycle RISC implementation has a 10 ns clock cycle, loads take 5 clock cycles and account for 40% of the instructions, and all other instructions take 4 clock cycles.

• If pipelining the machine adds 1 ns to the clock cycle, how much speedup in instruction execution rate do we get from pipelining?

\[
\text{MC Ave. Instr. Time} = \text{Clock cycle} \times \text{Average CPI} \\
= 10 \text{ ns} \times (0.6 \times 4 + 0.4 \times 5) \\
= 44 \text{ ns}
\]

\[
\text{PL Ave. Instr. Time} = 10 + 1 = 11 \text{ ns}
\]

Speedup = 44 / 11 = 4

• The above expression assumes a pipelining CPI of 1

Should we expect this in practice? Any complications?