Before We Get Started…

- **Goal**
  - Spend five days getting familiar with what advanced computing do for you
  - Justification: using computers well requires an understanding of how they work

- **Reaching this goal**
  - Cover some basics about computing at large (first day)
  - Spend three days on GPU computing
  - Last day: cover multi-core computing w/ OpenMP last day or more hands-on

- **The material will probably be outside your comfort zone**
  - Take it as an opportunity to break into something new
**Day 1 – Monday, December 10**

9:00 – 12:00 Lecture [Intro, computing]
- Introduction: Example use of advanced computing in multibody dynamics
- Sequential computing, the hardware/software interface:
  - The processor: the fetch–decode–execute cycle, registers, pipelining, execution performance
12:00 – 13:30 Lunch
13:30 – 16:30 Hands–on component
  - Memory issues: memory hierarchy, DRAM/SRAM memory, caches, cache coherence
- Quick overview of trends in parallel computing (multi–core and GPU computing); Top500 list
- Accessing Euler
- Compiling on Euler
- Debugging on Euler

**Day 2 – Tuesday, December 11**

9:00 – 12:00 Lecture [CUDA intro]
- NVIDIA’s CUDA intro: computation model and execution configuration
- CUDA memory allocation
- CUDA example: matrix multiplication
12:00–13:30 Lunch
13:30–16:30 Hands–on component
- Timing a GPU application
- Scaling a vector on the GPU
- Vector addition
- Dot product
- Work on assignment: Parallel reduce operation
Day 3 – Wednesday, December 12
9:00 – 12:00 Lecture [More advanced CUDA features]
- CUDA Memory model: registers and global, constant, texture, shared, local memories
- CUDA execution scheduling; thread divergence
- CUDA streams
12:00–13:30 Lunch
13:30–16:30 Hands–on component
- Dot product, revisited – using shared memory to improve performance
- Matrix multiplication: Large, tiled matrix–matrix multiplication with and without shared memory
- Work on assignment: Matrix convolution

Day 4 – Thursday, December 13
9:00 – 12:00 Lecture [Productivity tools, GPU computing]
- Wrap up CUDA: optimization rules of thumb
- The CUDA thrust library
- CUDA profiling
- CUDA debugging
- The CUDA library landscape
12:00–13:30 Lunch
13:30–16:30 Hands–on component
- example of using the thrust library: reduction and prefix scan operations
- profiling of dot product using nvvp
- Work on assignment: Array processing using thrust

Day 5 – Friday, December 14
9:00 – 12:00 Lecture [Multi-core programming using OpenMP]
- OpenMP, introduction
- Work sharing: sections, tasks
- OpenMP variable sharing
12:00–13:30 Lunch
13:30–15:30 Hands–on component
- Integral evaluation using OpenMP
- Work on assignment: Matrix Convolution, comparison w/ GPU implementation
Dan Negrut

- Bucharest Polytechnic University, Romania
  - B.S. – Aerospace Engineering (1992)

- University of Iowa
  - Ph.D. – Mechanical Engineering (1998)

- MSC.Software
  - Product Development Engineer 1998-2005

- University of Michigan, Ann Arbor
  - Adjunct Assistant Professor, Dept. of Mathematics (2004)

- DOE’s Argonne National Laboratory, Division of Mathematics and Computer Science

- University of Wisconsin-Madison, since Nov. 2005
  - Associate Professor, Mechanical Engineering & Electrical and Computer Engineering
  - Research Focus: Computational Dynamics (Dynamics of Multi-body Systems)
  - Technical lead, Simulation-Based Engineering Lab (http://sbel.wisc.edu)
  - Director, Wisconsin Applied Computing Center (http://wacc.wisc.edu)
Pointers for Information

- Slides will be made available at
  [http://outreach.sbel.wisc.edu/Workshops/GPUworkshop/](http://outreach.sbel.wisc.edu/Workshops/GPUworkshop/)

- This material is part of a HPC class I taught in Spring 2012 at UW-Madison
  - GPU Computing, OpenMP, and MPI
  - Class material available online (slides & video streaming): [http://sbel.wisc.edu/Courses/ME964/2012/](http://sbel.wisc.edu/Courses/ME964/2012/)
Pointers for Information

- Slides will be made available at
  http://outreach.sbel.wisc.edu/Workshops/GPUworkshop/

- This material is part of a HPC class I taught in Spring 2012 at UW-Madison
  - GPU Computing, OpenMP, and MPI
  - Class material available online (slides & video streaming): http://sbel.wisc.edu/Courses/ME964/2012/

- Today’s material is heavily based on information available in Patterson’s book
Today’s Computer

- Follows computation paradigm formalized by Von Neumann in late 1940s

- The von Neumann model:
  - There is no distinction between data and instructions
  - Data and instructions are stored in memory as a string of 0 and 1 bits
    - Instructions are fetched + decoded + executed
    - Data is used to produce results according to rules specified by the instructions

Picture credits: Blaise Barney
Lawrence Livermore National Laboratory

Monday, December 10, 12
From Code to Instructions

- There is a difference between a line of code and a processor instruction.
- Example:
  - Line of C code:
    
    ```
    a[4] = delta + a[3]; //line of C code
    ```
  - Set of MIPS assembly code generated by the compiler:
    ```
    lw $t0, 12($s2)  # reg $t0 gets value stored 12 bytes from address in $s2
    add $t0, $s4, $t0  # reg $t0 gets the sum of values stored in $s4 and $t0
    sw $t0, 16($s2)  # a[4] gets the sum delta + a[3]
    ```
  - Set of three corresponding MIPS instructions produced by the compiler:
    ```
    10001110010010000000000000001100
    0000001010001000010000000100000
    1010111001001000000000000010000
    ```
From Code to Instructions

- C code – what you write to implement an algorithm
- Assembly code – what your code gets translated into by the compiler
- Instructions – what the assembly code gets translated into by the compiler

Observations:
- The compiler typically goes from C code directly to machine instructions
- Machine instructions: what you see in an editor like Notepad or vim or emacs if you open up an executable file
- There is a one-to-one correspondence between an assembly line of code and an instruction
- Assembly line of code can be regarded as an instruction that is expressed in a way that humans can relatively easy figure out what happens
- Back in the day people wrote assembly code
- Today coding in assembly done only for the super critical parts of a program if you want to optimize and don’t trust the compiler
Instruction Set Architecture (ISA)

- The same line a C code can lead to a different set of instructions on two different computers.

- This is so because two CPUs might implement a different Instruction Set Architecture (ISA).

- ISA: defines the “language” that expresses at a very low level the actions of a processor.

- Example:
  - Microsoft’s Surface Tablet
    - RT version: uses a Tegra chip, which implements an ARM Instruction Set
    - Pro version: uses an Intel Atom chip, which implements x86 Instruction Set.
Example: the same C code leads to different assembly code (and different set of machine instructions, not shown here)

```c
int main()
{
    const double fctr = 3.14/180.0;
    double a = 60.0;
    double b = 120.0;
    double c;
    c = fctr*(a + b);
    return 0;
}
```

### x86 ISA
```
call  ___main
  fldl   LC0
  fstpl  -40(%ebp)
  fldl   LC1
  fstpl  -32(%ebp)
  fldl   LC2
  fstpl  -24(%ebp)
  fldl   LC0
  fmulp  %st, %st(1)
  fstpl  -16(%ebp)
  movl   $0, %eax
  addl   $36, %esp
  popl   %ecx
  popl   %ebp
  leal   -4(%ecx),
%esp
  ret
LC0:  .long  387883269
    .long  1066524452
    .align 8
LC1:  .long  0
    .long  1078853632
    .align 8
LC2:  .long  0
    .long
```

### MIPS ISA
```
main:       .frame $fp,48,$31  # vars= 32, regs= 1/0, args= 0,
gp= 8       .mask 0x40000000,-4
            .fmask 0x00000000,0
            .set noreorder
            .set nomacro
            addiu $sp,$sp,-48
            sw $fp,44($sp)
            move $fp,$sp
            lui $2,%hi($LC0)
            lwc1
            ...
            mul.d $f0,$f2,$f0
            swc1 $f0,32($fp)
            swc1 $f1,36($fp)
            move $2,$0
            move $sp,$fp
            lw $fp,44($sp)
            addiu $sp,$sp,48
            j $31

$LC0:       .word 3649767765
            .word 1066523892
            .align 3
$LC1:       .word 0
            .word 1078853632
            .align 3
$LC2:       .word 0
            .word 1079902208
            .ident "GCC: (Gentoo 4.6.3 p1.6, pie-0.5.2) 4.6.3"
```
Example: the same C code leads to different assembly code (and different set of machine instructions, not shown here)

```c
int main()
{
    const double fctr = 3.14/180.0;
    double a = 60.0;
    double b = 120.0;
    double c;
    c = fctr*(a + b);
    return 0;
}
```

---

**x86 ISA**

```assembly
call   ___main
     fldl  LC0
     fstpl -40(%ebp)
     fldl  LC1
     fstpl -32(%ebp)
     fldl  LC2
     fstpl -24(%ebp)
     faddl -24(%ebp)
     fldl  LC0
     fmulp   %st, %st(1)
     fstpl   -16(%ebp)
     movl   $0, %eax
     addl   $36, %esp
     popl   %ecx
     popl   %ebp
     leal   -4(%ecx),
     %esp
     ret
LC0:  .long    387883269
     .long    1066524452
     .align 8
LC1:  .long    0
     .long    1078853632
     .align 8
LC2:  .long    0
     .long
```

**MIPS ISA**

```assembly
main:     .frame    $fp,48,$31   # vars=32, regs=1/0, args=0, gp=8
          .mask    0x40000000,-4
          .fmask   0x00000000,0
          .set     noreorder
          .set     nomacro
          addiu    $sp,$sp,-48
          sw       $fp,44($sp)
          move     $fp,$sp
          lui      $2,%hi($LC0)
          lwc1...
          mul.d    $f0,$f2,$f0
          swc1     $f0,32($fp)
          swc1     $f1,36($fp)
          move     $2,$0
          move     $sp,$fp
          lw       $fp,44($sp)
          addiu    $sp,$sp,48
          j         $31

$LC0:    .word    3649767765
          .word    1066523892
          .align 3
$LC1:    .word    0
          .word    1078853632
          .align 3
$LC2:    .word    0
          .word    1079902208
          .ident "GCC: (Gentoo 4.6.3 p1.6, pie-0.5.2) 4.6.3"
```
Example: the same C code leads to different assembly code (and different set of machine instructions, not shown here)

```c
int main(){
    const double fctr = 3.14/180.0;
    double a = 60.0;
    double b = 120.0;
    double c;
    c = fctr*(a + b);
    return 0;
}
```

### x86 ISA

- `call ___main`  
- `fldl LC0`  
- `fstpl -40(%ebp)`  
- `fldl LC1`  
- `fstpl -32(%ebp)`  
- `fldl LC2`  
- `fstpl -24(%ebp)`  
- `faddl -24(%ebp)`  
- `fldl LC0`  
- `fmulp %st, %st(1)`  
- `fstpl -16(%ebp)`  
- `movl $0, %eax`  
- `addl $36, %esp`  
- `popl %ecx`  
- `popl %ebp`  
- `lea -4(%ecx), %esp`  
- `ret`

### MIPS ISA

- `.frame $fp,48,$31`  
- `.mask 0x40000000,-4`  
- `.fmask 0x00000000,0`  
- `.set noreorder`  
- `.set nomacro`  
- `.addiu $sp,$sp,-48`  
- `.sw $fp,44($sp)`  
- `.move $fp,$sp`  
- `.lui $2,%hi($LC0)`  
- `.lw` ...

### Assembly Code

- `.mul.d $f0,$f2,$f0`  
- `.swc1 $f0,32($fp)`  
- `.swc1 $f1,36($fp)`  
- `.move $2,$0`  
- `.move $sp,$fp`  
- `.lw $fp,44($sp)`  
- `.addiu $sp,$sp,48`  
- `.j $31`  
- `$LC0:`  
- `.long 387883269`  
- `.long 1066524452`  
- `.align 8`  
- `$LC1:`  
- `.long 0`  
- `.long 1078853632`  
- `.align 8`  
- `$LC2:`  
- `.long 0`  
- `.long 1079902208`  
- `.ident "GCC: (Gentoo 4.6.3 p1.6, pie-0.5.2) 4.6.3"
Example: the same C code leads to different assembly code (and different set of machine instructions, not shown here)

```c
int main(){
    const double fctr = 3.14/180.0;
    double a = 60.0;
    double b = 120.0;
    double c;
    c = fctr*(a + b);
    return 0;
}
```

**x86 ISA**

- call __main
- fldl LC0
- fstpl -40(%ebp)
- fldl LC1
- fstpl -32(%ebp)
- fldl LC2
- fstpl -24(%ebp)
- faddl -24(%ebp)
- fldl LC0
- fmulp %st, %st(1)
- fstpl -32(%ebp)
- movl $0, %eax
- addl $36, %esp
- popl %ecx
- popl %ebp
- leal -4(%ecx),
- %esp
- ret

**MIPS ISA**

- .frame $fp,48,$31 # vars= 32, regs= 1/0, args= 0,
- gp= 8
- .mask 0x40000000,-4
- .fmask 0x00000000,0
- .set noreorder
- .set nomacro
- addiu $sp,$sp,-48
- sw $fp,44($sp)
- move $fp,$sp
- lui $2,%hi($LC0)
- lw $fp,44($sp)
- addiu $sp,$sp,48
- j $31

**Label Definitions**

- $LC0:
  - .word 3649767765
  - .word 1066523892
  - .align 3
- $LC1:
  - .word 0
  - .word 1078853632
  - .align 3
- $LC2:
  - .word 0
  - .word 1079902208
  - .ident "GCC: (Gentoo 4.6.3 p1.6, pie-0.5.2) 4.6.3"
Instruction Set Architecture vs. Chip Microarchitecture

- ISA – can be regarded as a standard
  - Specifies what a processor should be able to do
    - Load, store, jump on less than, etc.

- Microarchitecture – how the silicon is organized to implement the functionality promised by ISA

- Example:
  - Intel and AMD both use the x86 ISA
  - Nonetheless, they have different microarchitectures
The CPU’s Control Unit (CU)

- Think of a CPU as a big kitchen
  - A work order comes in (this is an instruction)
  - The cook starts to cook a meal (this is the ALU)
  - Some ingredients are needed: meat, spinach, potatoes, etc. (this is the data)
  - Some ready to eat product goes out the kitchen: a soup (this is the result)

- The cook, the feeding of data, the movement of the boiled meat to chopping, boiling of pasta, etc. – they happen in a coordinated fashion (based on a kitchen clock) and is managed by the CU

- The CU manages/coordinates/controls based on information in the work order (the instruction)
The FDX Cycle

- FDX stands for Fetch-Decode-Execute
- This is what the CPU keeps doing to execute a sequence of instructions that combine to make up a program

- Fetch: an instruction is fetched from memory
  - Recall that it will look like (on 32 bits, MIPS, lw $t0, 12($s2)):
  - 10001110010010000000000000001100

- Decode: this strings of 1s and 0s are decoded by the CU
  - Example: here’s an “I” (eye) type instruction, made up of four fields
Decoding: Instructions Types

- Three types of instructions in MIPS ISA
  - Type I
  - Type R
  - Type J
Type I (MIPS ISA)

- The first six bits encode the basic operation; i.e., the opcode, that needs to be completed
  - Example adding two numbers (000000), subtracting two numbers (000001), dividing two numbers (000011), etc.
- The next group of five bits indicates in which register the first operand is stored
- The subsequent group of five bits indicates the register where the second operand is stored.
- Some instructions require an address or some constant offset. This information is stored in the last 16 bits
Type R (MIPS ISA)

- Type R gas the same first three fields op, rs, rt like I-type

- Packs three additional fields:
  - Five bit rd field (register destination)
  - Five bit shamt field (shift amount)
  - Six bit funct field, which is a function code that further qualifies the opcode
It All Boils Down to Transistors…

- Every 18 months, the number of transistors per unit area doubles
  - Current technology (2012): feature length is 22 nm
  - Next wave (2014): feature length is 14 nm

Example

- NVIDIA Fermi architecture of 2010:
  - 32 nm technology
  - Chips w/ 3 billion transistors → more than 500 scalar processors, 0.5 TFlops

- NVIDIA Fermi architecture 2012:
  - 22 nm technology
  - Chips w/ 7 billion transistors → more than 200 scalar processors, 1.5 TFlops
It All Boils Down to Transistors…

- Why are transistors important?
- More transistors increase opportunities for parallelism in emerging architectures
- Transistors can be organized to produce complex logical units that have the ability to execute instructions
AND, OR, NOT

- NOT logical operation is implemented using one transistor
- AND and OR logical ops requires two transistors

![AND circuit]

![OR circuit]

![NOT circuit]

- Truth tables for AND, OR, and NOT

<table>
<thead>
<tr>
<th>in₁</th>
<th>AND</th>
<th>in₂</th>
<th>OR</th>
<th>NOT</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>in₁</th>
<th>in₂</th>
<th>OR</th>
<th>NOT</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Example

- Design a digital logic block that receives three inputs via three bus wires and produces one signal that is 0 (low voltage) as soon as one of the three input signals is low voltage.
  - In other words, it should return 1 if and only if all three inputs are 1
Example

- Design a digital logic block that receives three inputs via three bus wires and produces one signal that is 0 (low voltage) as soon as one of the three input signals is low voltage.
  - In other words, it should return 1 if and only if all three inputs are 1

Truth Table

<table>
<thead>
<tr>
<th>in₁</th>
<th>in₂</th>
<th>in₃</th>
<th>Out</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Monday, December 10, 12
Example

- Design a digital logic block that receives three inputs via three bus wires and produces one signal that is 0 (low voltage) as soon as one of the three input signals is low voltage.
- In other words, it should return 1 if and only if all three inputs are 1.

<table>
<thead>
<tr>
<th>in₁</th>
<th>in₂</th>
<th>in₃</th>
<th>Out</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Truth Table

Logic Equation:

\[ out = \overline{in_3} + in_2 \cdot in_1 \]
Example

- Design a digital logic block that receives three inputs via three bus wires and produces one signal that is 0 (low voltage) as soon as one of the three input signals is low voltage.
  - In other words, it should return 1 if and only if all three inputs are 1

<table>
<thead>
<tr>
<th>in_1</th>
<th>in_2</th>
<th>in_3</th>
<th>Out</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Logic Equation:

\[
out = \overline{in_3} + \overline{in_2} \cdot in_1
\]

- Solution: digital logic block is a combination of AND, OR, and NOT gates
  - The NOT is represented as a circle O applied to signals moving down the bus

\[
in_1, \quad in_2
\]
Example

- Implement a digital circuit that produces the Carry-out digit in a one bit summation operation
Example

- Implement a digital circuit that produces the Carry-out digit in a one bit summation operation

### Truth Table

<table>
<thead>
<tr>
<th>Inputs</th>
<th>Outputs</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>in₁</td>
<td>in₂</td>
<td>CarryIn</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Example

- Implement a digital circuit that produces the Carry-out digit in a one bit summation operation

<table>
<thead>
<tr>
<th>Inputs</th>
<th>Outputs</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>in₁</td>
<td>in₂</td>
<td>CarryIn</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Truth Table

<table>
<thead>
<tr>
<th>Logic Equation:</th>
</tr>
</thead>
<tbody>
<tr>
<td>CarryOut = (in₁·CarryIn)+(in₂·CarryIn)+(in₁·in₂)</td>
</tr>
</tbody>
</table>

Monday, December 10, 12
Example

- Implement a digital circuit that produces the Carry-out digit in a one bit summation operation

### Truth Table

<table>
<thead>
<tr>
<th>Inputs</th>
<th>Outputs</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>in₁</td>
<td>in₂</td>
<td>CarryIn</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Sum is in base 2

0+0 is 0; the CarryIn kicks in, makes the sum 1.
0+0 is 0; the CarryIn kicks in, makes the sum 1.
0+0 is 0; the CarryIn kicks in, makes the sum 1.
0+1 is 1, but CarryIn is 1; sum ends up being 0, CarryOut is 1.
0+1 is 1, but CarryIn is 1; sum ends up being 0, CarryOut is 1.
0+1 is 1, but CarryIn is 1; sum ends up being 0, CarryOut is 1.
1+0 is 1, but CarryIn is 1; sum ends up being 0, CarryOut is 1.
1+1 is 0, carry 1.
1+1 is 0, carry 1.
1+1 is 0 and you CarryOut 1. Yet the CarryIn is 1, so the 0 in the sum becomes

### Logic Equation:

\[
\text{CarryOut} = (\text{in}_1 \cdot \text{CarryIn}) + (\text{in}_2 \cdot \text{CarryIn}) + (\text{in}_1 \cdot \text{in}_2)
\]
Integrated Circuits-A One Bit Combo: OR, AND, 1 Bit Adder

- 1 Bit Adder, the Sum part
Integrated Circuits-A One Bit Combo: OR, AND, 1 Bit Adder

- 1 Bit Adder, the Sum part

- Combo: OR, AND, 1 Bit Sum
  - Controlled by the input “Operation”
Integrated Circuits: Ripple Design of 32 Bit Combo

- Combine 32 of the 1 bit combos in an array of logic elements
  - Get one 32 bit unit that can do OR, AND, +
Integrated Circuits: From Transistors to CPU
Registers
Registers

- Instruction cycle: fetch-decode-execute (FDX)

- CU – responsible for controlling the process that will deliver the request baked into the instruction
  - Imagine this as sending the right signals to a MUX (SEE FIG ???)

- ALU – does the busy work to fulfill the request put forward by the instruction

- The instruction that is being executed should be stored somewhere

- Fulfilling the requests baked into an instruction usually involves handling input values and generates output values
  - These data needs to be stored somewhere
Registers

- Registers, quick facts:
  - A register is a type of memory
  - A register is the fastest memory available – closest to the ALU
  - Used to store the address in memory that contains the instruction that needs to be executed.
  - Might store the values of two operands that need to be added
  - You cannot control what gets kept in registers (with a few exceptions)

- The number AND size of registers used are specific to a ISA

- In MPIS ISA: there are 32 registers of 32 bits that are used to store critical information
Register Types

- Discuss several register types typically encountered in a CPU (abbreviation in parenthesis)
  - List not comprehensive, showing only the more important ones

- Instruction register (IR) – a register that holds the instruction that is executed
  - Sometimes known as “current instruction register” CIR

- Program Counter (PC) – a register that holds the address of the next instruction that will be executed
  - NOTE: unlike IR, PC contains an *address* of an instruction, not the actual instruction
Register Types [Cntd.]

- Memory Data Register (MDR) – register that holds data that has been read in from main memory or produced by the CPU and waiting to be stored in main memory.

- Memory Address Register (MAR) – the address of the memory location in main memory (RAM) where input/output data is supposed to be read in/written out.
  - NOTE: unlike MDR, MAR contains an *address* of a location in memory, not actual data.

- Return Address (RA) – the address where upon finishing a sequence of instructions, the execution should return and commence with the execution of subsequent instruction.
Register Types [Cntd.]

- The registers on previous two slides are a staple in any chip design.

- There are several other registers that are common to most chip designs yet they are encountered in different numbers.

- Since they come in larger numbers they don’t have an acronym:
  - Registers for Subroutine Arguments (4) – a0 through a3
  - Registers for temporary variables (10) – t0 through t9
  - Registers for saved temporary variables (8) – s0 through s7
    - Saved between function calls
Several other registers are involved in handling function calls

Summarized below, but their meaning is only apparent in conjunction with the organization of the virtual memory

- **Global Pointer (gp)** – a register that holds an address that points to the middle of a block of memory in the static data segment
- **Stack Pointer (sp)** – a register that holds an address that points to the last location on the stack
- **Frame Pointer (fp)** - a register that holds an address that points to the beginning of the procedure frame
- **Return Address (ra)** - a register that holds an address that points to instruction that will be executed upon return from the current procedure
Register, Departing Thoughts

- **Examples:**
  - In 32 bit MIPS ISA, there are 32 registers
  - On a GTX580 NVIDIA card there are more than 500,000 32 bits temporary variable registers to keep busy 512 Scalar Processors (SPs) that made up 16 Stream Multiprocessors (SMs)

- Registers are very precious resources

- Increasing their number is not straightforward
  - Need to change the entire design of the chip
  - Need to work out the control flow
The Fetch – Decode – Execute – Reset Cycle

CREDITS: http://www.ibsb.ro
Goal

- Describe in simple terms the fetch / decode / execute / reset cycle and the effects of the stages of the cycle on specific registers
Fetch-Decode-Execute-Reset Cycle

- The control unit controls and synchronises this cycle
  - Loads / copies / places / passes, decodes and executes.

- The cycle is reset at the end and the algorithm gets repeated for each instruction
Registers Involved in FDX

- **PC (also known as SQR)**
  - Program Counter / *Sequence Control Register*

- **MAR**
  - Memory Address Register

- **MDR (aka *MBR*)**
  - Memory Data Register / *Memory Buffer Register*

- **CIR**
  - Current Instruction Register

- **ALU’s Accumulator**

- **Temporary registers that the compiler deemed necessary to store intermediate information for indirect operations**
  - Example: reads.loads to addresses in main memory
Summary:
Fetch – Decode – Execute – Reset Cycle
The Fetch Stage

1. Load the address of next instruction from PC into MAR

2. Copy the instruction/data; i.e., information, that is in the memory address stored by MAR into MDR
   - NOTE: MDR used whenever information moved from the CPU to main memory or the other way around

3. Increment the PC by 1
   - PC now contains the address of the next instruction (assuming that the instructions are in consecutive locations)

4. Load the instruction/data that is now in MDR into CIR

NOTE: Journey of instruction : memory ! MDR ! CIR
5. Contents of CIR split into (a) operation code, e.g., store, add, jump, multiply, etc. instructions; and (b) operand information, if present

6. Decode the instruction that is in the CIR

NOTE 1: The process of decoding generates a chain reaction controlled by the CU. Based on the bit pattern in the operation code, the MUXs in the logic end up with precise selection inputs to implement, for instance, a sum operation using an appropriate complex hardware logic block (discussed previously)

NOTE 2: In picking up an operand, temporary registers come into play. As an example, the message coded into the instruction might be: the second operand is stored in the temporary register number 3. How it got there is something that you owe to one of the more recent instructions if not the previous one
The Execute Stage

7. What is involved in this step depends on the instruction being executed:

- For a jump:
  - Based on direct/indirect information baked into the CIR instruction, load the address of the instruction to be executed next into the PC
    - Direct: it explicitly states that you need to jump 7 instructions, for instance
    - Indirect: the address of the next instruction is stored in a temporary register

- For a direct input/load:
  - Take data input from MDR or some temporary register and place in AC

- For an indirect load from memory:
  - Based on info baked into the CIR instruction, load memory address of interest (the source) into MAR
  - Copy data from memory address held in MAR into MDR
  - Copy data in MDR into AC
For an indirect store instruction:
- Based on info baked into the CIR instruction, load memory address of interest (the destination) into MAR
- Copy data in AC to MDR
- Copy data in MDR into memory address held in MAR

For an indirect add instruction:
- Based on info baked into the CIR instruction, load memory address of interest (the source) into MAR
- Copy number from memory address held in MAR into MDR
- Add number in MDR to number in AC (accumulator will now hold the result)

For a direct output instruction (directly from AC):
- Output number in AC into MDR or some temporary register
The Execute Stage [end]

- For a direct output instruction (directly from AC):
  - Based on info baked into the CIR instruction, load memory address of interest (the source) into MAR
  - Copy data in AC to MDR
  - Copy data in MDR into memory address held in MAR
The Reset Stage

8. Cycle is reset (restarted) by moving on to process the instruction in PC (step 1 in Fetch Stage).
Fetch

CPU

PC

MAR

MDR

CIR
Fetch

CPU

PC → MAR

Copy of address of next instruction

MDR

CIR
Copy of instruction in memory address held in MAR

Fetch

CPU

PC

MAR

MDR

CIR
Fetch

CPU

PC

MAR

MDR

Instruction

CIR
Fetch

CPU

PC → Copy of address of next instruction → MAR

MDR

CIR
Fetch

CPU

PC

MAR

MDR

Instruction

CIR

Monday, December 10, 12
Execute instruction
(What is involved in this depends on the instruction being executed - demonstrated on the following slides).
Execute Diagram:

Jump Instruction
CPU

- PC
- MAR
- MDR
- CIR

Execute
Jump

Back to list of instructions

Monday, December 10, 12
Copy of address part instruction (address to jump to).
Execute Diagram:
Direct Input/Load into AC
Execute

Input / Load (number directly) into accumulator

CPU

PC  MAR  MDR  CIR
Execute

Input / Load (number directly) into accumulator

CPU

PC

MDR

CIR

Number inputted / to be loaded.

Back to list of instructions

Monday, December 10, 12
Execute

Input / Load (number directly) into accumulator

CPU

PC  MAR  MDR  CIR

Copy of number in MDR.

Accumulator
Execute Diagram:
Load From Memory Instruction
Execute

Load (from memory)
Execute

Load (from memory)
Execute

Load (from memory)
Execute

Load (from memory)

CPU

- PC
- MAR
- MDR
- CIR
- Accumulator

Copy of data in MDR

Back to list of instructions
Execute Diagram: 

Store

Assume data has either been inputted, loaded (directly or from memory) or a calculation has been performed. Any of the above will mean there is data in the accumulator and it is this data that will be stored.
Copy of data in MDR stored in memory address held in MAR

Monday, December 10, 12
Execute Diagram:

Direct Add

Assume a number has already been inputted or loaded (directly or from memory) into the accumulator
Execute

Add (a number directly)
Execute

Add (a number directly)
Add (a number directly)

NB. The ALU now does the arithmetic. Accumulator value is now the result of the addition.

i.e. Accumulator = Accumulator + contents of MDR
Execute Diagram:

Indirect Add

Assume a number has already been inputted or loaded (directly or from memory) into the accumulator
Execute

Add (from memory)
Execute

Add (from memory)

CPU

PC

MAR

Address part of instruction (of number to add).

MDR

CIR

Back to list of instructions
Execute

Add (from memory)

- PC
- MAR
- MDR
- CIR

Copy of number in memory address held in MAR

Back to list of instructions
execute

add (from memory)

CPU

PC

MAR

MDR

CIR

alu

accumulator

nb. the alu now does the arithmetic.

accumulator value is now the result of the addition.

i.e. accumulator = accumulator + contents of mdr
Execute Diagram:
Output, direct from AC
Execute

Output (directly from accumulator)
Execute

Output (directly from accumulator)
Execute Diagram:

Output, indirect from AC
Execute

Output (from memory)

CPU

Memory

PC

MAR

MDR

Accumulator

CIR

Address part of instruction (of data to output).

Back to list of instructions
CIR
CPU
Address part of instruction (of data to output).

Output (from memory)

MDR
Memory

Back to list of instructions
Execute

Output (from memory)

CPU

Memory

Store data in memory address held in MAR

Accumulator

MDR

PC

MAR

Back to list of instructions

Monday, December 10, 12
Cycle is reset (restarted) by passing control back to the PC.
The Bus
What is a Bus?

- A communication pathway connecting two or more devices

- Carries data
  - Remember that there is no difference between “data” and “instruction” at this level

- Often grouped
  - A number of channels in one bus
  - e.g. 32 bit data bus is 32 separate single bit channels

- Width is a key determinant of performance
  - 8, 16, 32, 64 bit
The Bus: Big and Yellow?

- What do buses look like?
  - Parallel lines on circuit boards
  - Ribbon cables
  - Strip connectors on mother boards
    - e.g. PCI
  - Sets of wires
Buses

- There are a number of possible interconnection systems

- Single and multiple BUS structures are most common
  - e.g. Control/Address/Data bus (on Personal Computers)
  - e.g. Unibus (on old DEC-PDP)
Single Bus Problems

- Lots of devices on one bus leads to propagation delays
  - Long data paths mean that co-ordination of bus use can adversely affect performance
  - Delays hit when aggregate data transfer approaches bus capacity

- Most systems use multiple buses to overcome these problems

- Some of these buses can rely information at different frequencies (speeds)
Address bus

- **Purpose:** Identify the source or destination of data

- **Example:**
  - CPU needs to read an instruction (data) from a given location in memory

- **Bus width determines maximum memory capacity of system**
  - **Example:**
    - 8080 has 16 bit address bus giving 64k address space
Two Other Bus Types
Two Other Bus Types

- **Control Bus**, purpose: control and timing of information
  - Memory read/write signal
  - Interrupt request
  - Clock signals
Two Other Bus Types

- **Control Bus**, purpose: control and timing of information
  - Memory read/write signal
  - Interrupt request
  - Clock signals

- **Data Bus**, purpose: move data back and forth
  - To/From CPU
  - To/From Main memory
  - To/From peripheral devices
Two Other Bus Types

- **Control Bus**, purpose: control and timing of information
  - Memory read/write signal
  - Interrupt request
  - Clock signals

- **Data Bus**, purpose: move data back and forth
  - To/From CPU
  - To/From Main memory
  - To/From peripheral devices

- **Note**: these are not the only types of buses, but are the most common and most important
Bus Interconnection Scheme
Traditional Architecture Layout (Plain Vanilla)
Higher End Architecture Layout

[Diagram of higher end architecture layout showing components like Processor, Local Bus, Cache/Bridge, System Bus, Main Memory, SCSI, P1394, Graphic, Video, LAN, FAX, Expansion bus interface, Modem, Serial, and Expansion Bus.]
Pipelining
• Recall the FDX cycle, carried out in conjunction with each instruction
  • Fetch, Decode, Execute

• A closer look at what gets fetched (instructions and data) and then what happens upon execution leads to a generic five stage process associated with an instruction

• “generic” means that in a first order approximation, these five stages can represent all instructions, although some instructions might not have all five stages:
  • Stage 1: Fetch an instruction
  • Stage 2: Decode the instruction while reading registers
  • Stage 3: Execute the operation (Ex.: might be a request to calculate an address)
  • Stage 4: Memory access for operand
  • Stage 5: Write-back into register file

• NOTE: In general, these are the five generic stages of a RISC architecture
  • BTW, MIPS is a special case of RISC
Not all types of instructions require all five stages

Example, based on the MIPS ISA:

- Number of cycles required
  - Load a word (lw): five clock cycles. In absolute terms, 800 ps (the register read/write burn only half of the time in a cycle)
  - Store a word (sw) as well as R-format instructions: four cycles
    - Sw: 700 ps
    - R-format instructions shown: 600 ps
  - J-type instruction (branch-on-equal): 3 cycles. In absolute terms, 500 ps

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Load word (lw)</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>800 ps</td>
</tr>
<tr>
<td>Store word (sw)</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
<td>700 ps</td>
</tr>
<tr>
<td>R-Format (add, sub)</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
<td>Y</td>
<td>600 ps</td>
</tr>
<tr>
<td>Branch (beq)</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
<td>N</td>
<td>500 ps</td>
</tr>
</tbody>
</table>
Pipelining, Basic Idea

- At the cornerstone of pipelining is the observation that the following processes can happen simultaneously when processing five instructions:
  - Instruction 1 is in the 5th stage of the FDX cycle
  - Instruction 2 is in the 4th stage of the FDX cycle
  - Instruction 3 is in the 3rd stage of the FDX cycle
  - Instruction 4 is in the 2nd stage of the FDX cycle
  - Instruction 5 is in the 1st stage of the FDX cycle

- The above is a five stage pipeline

- An ideal situation is when each of these stages takes the same amount of time for completion
  - The pipeline is balanced

- If there is a stage that takes a significantly longer time since it does significantly more than the other stages, it should be broken into two and the length of the pipeline increases by one stage
Pipelining, Example

- The first to use the idea of pipelining on an industrial scale and make a difference was Henry Ford.

- Vehicle assembly line: a good example of a pipelined process
  - The output of one stage (station) becomes the input for the downstream stage (station)
  - It is bad if one station takes too long to produce its output since all the other stations idle a bit at each cycle of the production
  - “Cycle” is the time it takes from the moment a station gets its input to the moment the output is out of the station
  - In this setup, an instruction (vehicle) gets executed (assembled) during each cycle
Example: Streaming for execution
3 SW instructions

sw $t0, 0($s2)
sw $t1, 32($s2)
sw $t2, 64($s2)

- Case 1: No pipelining – 2100 picoseconds [ps]
Example: Streaming for execution
3 SW instructions

sw $t0, 0($s2)
sw $t1, 32($s2)
sw $t2, 64($s2)

- Case 2: With pipelining – 1200 picoseconds [ps]
Pipelining, Benefits

- Assume that you have
  1. A very large number of instructions
  2. Balanced stages
     - Not the case in our example, since “Reg” wasted half of the pipeline stage time
  3. A pipeline that is larger than or equal to the number “p” of stages associated with the typical ISA instruction

- If 1 through 3 above hold, in a first order approximation, the speed-up you get out of pipelining is approximately “p”

- Benefit stems from parallel processing of FDX stages
  - This kind of parallel processing of stages is transparent to the user
    - Unlike GPU or multicore parallel computing, you don’t have to do anything to benefit of it
Pipelining, Benefits

- Why this speedup
  - Goes back to computing in parallel
  - All instructions have p stages and you have a pipeline of length p
  - Nonpipelined execution of N instructions: N*p cycles needed to finish
  - Pipelined execution of N instruction: during each cycle, p stages out of N*p are executed; you only need N cycles
  - This glosses over the fact that you need to prime the pipeline and there is a shutdown sequence that sees pipeline stages being empty
Pipelining, Good to Remember

- The amount of time required to complete one stage of the pipeline: one cycle
- Pipelined processor: one instruction processed in each cycle
- Nonpipelined processor: several cycles required to process an instruction:
  - Four cycles for SW, five for LW, four for add, etc.
- Important Remark:
  - Pipelining does not decrease the time to process one instruction but rather it increases the throughput of the processor
Pipelining Hazards

- Come in three flavors
  - Structural hazards
  - Data hazards
  - Control hazards
The instruction pipelining analogy w/ the vehicle assembly line breaks down at the following point:

- A real world assembly line assembles the same product for a period of time
- Might be quickly reconfigured to assemble a different product
- Instruction pipelining must process a broad spectrum of instructions that come one after another
  - Example: A J-type instruction coming after a R-type instruction, which comes after three I-Type instructions
  - If they were the same instructions (vehicles), designing a pipeline (assembly line) is straightforward

A structural hazard refers to the possibility of having a combination of instructions in the pipeline that are contending for the same piece of hardware.
Pipeline **Structural Hazards** [2/2]

- Possible Scenario: you have a six stage pipeline and the instruction in stage 1 and instruction in stage 5 both need to use a register to store a temporary variable.
  - Resolution: there should be enough registers provisioned so that no combination of instructions in the pipeline lead to RAW, WAR, etc. type issue
  - Alternative solution: serialize the access, basically stall the pipeline for a cycle so that there is no contention

- Note:
  - Adding more registers is a static solution; expensive and very consequential (requires a chip design change)
  - Stalling the pipeline at run time is a dynamic solution that is inexpensive but slows down the execution
Pipeline **Data** Hazards [1/2]

- Consider the following example in a five stage pipeline setup:

  ```
  add $t0, $t2, $t4  # $t0 = $t2 + $t4
  addi $t3, $t0, 16  # $t3 = $t0 + 16 (“add immediate”)
  ```

- The first instruction is processed in five stages

- Its output (value stored in register $t0) is needed in the very next instruction

- Data hazard: unavailability of $t0 to the second instruction, which references this register

- Resolution (less than ideal)
  - Pipeline stalls to wait for the first instruction to fully complete

Monday, December 10, 12
Pipeline **Data Hazards** [2/2]

- **add** $t0, $t2, $t4  # $t0 = $t2 + $t4
- **addi** $t3, $t0, 16  # $t3 = $t0 + 16 (“add immediate”)

- Alternative [the good] Resolution: use “forwarding” or “bypassing”

- Key observation: the value that will eventually be placed in $t0 is available after stage 3 of the pipeline (where the ALU actually computes this value)

- Provide the means for that value in the ALU to be made available to other stages of the pipeline right away
  - Nice thing: avoids stalling - don’t have to wait several other cycles before the value made its way in $t0
  - This process is called a forwarding of the value

- Supporting forwarding does not guarantee resolution of all scenarios
  - On relatively rare occasions the pipeline ends up stalled for a couple of cycles

- Note that the compiler can sometimes help by re-ordering instructions
  - Not always possible
Pipeline Control Hazards [Setup]

- What happens when there is an “if” statement in a piece of C code?
- A corresponding machine instruction decides the program flow
  - Specifically, should the “if” branch be taken or not?
- Processing this very instruction to figure out the next instruction (branch or no-branch) will take a number of cycles
- Should the pipeline stall while this instruction is fully processed and the branching decision becomes clear?
  - If yes: approach works, but it is slow
  - If no: you rely on branch prediction and proceed fast but cautiously
Pipeline Control Hazards: Branch Prediction

- Note that when you predict wrong you have to discard instruction[s] executed speculatively and take the correct execution path

- Static Branch Prediction (1\textsuperscript{st} strategy out of two):
  - Always predict that the branch will not be taken and schedule accordingly
  - There are other heuristics for proceeding: for instance, for a do-while construct it makes sense to always be jumping back at the beginning of the loop
    - Similar heuristics can be produced in other scenarios (a “for” loop, for instance)

- Dynamic Branch Prediction (2\textsuperscript{nd} strategy out of two):
  - At a branching point, the branch/no-branch decision can change during the life of a program based on recent history
  - In some cases branch prediction accuracy hits 90%
Pipelining vs. Multiple-Issue

- Pipelining should not be confused with “Multiple-Issue” as an alternative way of speeding up execution.
- A Multiple-Issue processor core is capable of processing more than one instruction at each cycle:
  - Example 1: performing an integer operation while performing a floating point operation – they require different resources and therefore can proceed simultaneously.
  - Example 2: the two lines of C code below lead to a set of instructions that can be executed at the same time:

```c
int a, b, c, d;
//some code setting a and b here
    c = a + b;
    d = a - b;
```
Pipelining vs. Multiple-Issue

- Multiple-Issue can be done statically or dynamically
  - Static multiple-issue:
    - Controlled (defined) at compile time
    - Used, for instance, by NVIDIA - very common in parallel computing on the GPU
  - Dynamic multiple-issue:
    - Controlled (defined) at run time, on the fly
    - Used heavily by Intel

- NOTE: Both pipelining and multiple-issue are representations of what is called Instruction-Level Parallelism (ILP)
Attributes of Dynamic Multiple-Issue

- Instructions are issued from one instruction stream
- More than one instruction is processed by the same core in the same clock cycle
- The data dependencies between instruction being processed takes place at run time
- NOTE: sometimes called a superscalar architecture
Measuring Computing Performance
Nomenclature

- **Program Execution Time** – sometimes called wall clock time, elapsed time, response time
  - Most meaningful indicator of performance
  - Amount of time from the beginning of a program to the end of the program
  - Includes (factors in) all the housekeeping (running other programs, OS issues, etc.) that the CPU has to do while running the said program

- **CPU Execution Time**
  - Like “Program Execution Time” but counting only the amount of time that is effectively dedicated to the said program
  - Requires a profiling tool to gauge

- On a dedicated machine; i.e., a quiet machine, Program Execution Time and CPU Execution Time would virtually be identical
Nomenclature [Cntd.]

- Qualifying CPU Execution Time further:
  - User time – the time spent processing instructions compiled out of code generated by the user or in libraries that are directly called by user code
  - System time – time spent in support of the user’s program but in instructions that were not generated out of code written by the user
    - OS support: open file for writing/reading, throw an exception, etc.
  - The line between the user time and system time is somewhat blurred, hard to delineate these two times at times

- Clock cycle, clock, cycle, tick – the length of the period for the processor clock; typically a constant value dictated by the frequency at which the processor operates
  - Example: 2 GHz processor has clock cycle of 500 picoseconds
The CPU Performance Equation

• The three ingredients of the CPU Performance Equation:
  • Number of instructions that your program executes (Instruction Count)
  • Average number of clock cycles per instructions (CPI)
  • Clock Cycle Time

• The CPU Performance Equation reads:
  \[
  \text{CPU Time} = \text{Instruction Count} \times \text{CPI} \times \text{Clock Cycle Time}
  \]

• Alternatively, using the clock rate
  \[
  \text{CPU Time} = \frac{\text{Instruction Count} \times \text{CPI}}{\text{Clock Rate}}
  \]
CPU Performance: What can be done?

- To improve performance the product of three factors should be reduced.

- For a long time, the community surfed the wave of “let’s increase the frequency”; i.e., reduce clock cycle time.
  - We eventually hit a wall this way (the “Power Wall”)

- As repeatedly demonstrated in practice, reducing the Instruction Count (IC) often times leads to an increase in CPI. And the other way around.
  - Ongoing argument: whether RISC or CISC is the better ISA
    - The former is simple and therefore can be optimized easily. Yet it requires a large number of instructions to accomplish something in your C code.
    - The latter is mind boggling complex but instructions are very expressive. Leads to few but expensive instructions to accomplish something in your C code.
    - Specific example: ARM vs. x86.
SPEC CPU Benchmarks

- There are benchmarks used to gauge the performance of a processor.

- Idea: gather a collection of programs that use a good mix of instructions and flex the muscles of the chip.

- These programs are meant to be representative of a class of applications that people are commonly using and not favor a chip manufacturer at the expense of another one.

- Example: a compiler is a program that is used extensively, so it makes sense to have it included in the benchmark.

- Two common benchmarks:
  - For programs that are dominated by floating point operations (CFP2006).
  - A second one is meant to be a representative sample of programs that are dominated by integer arithmetic (CINT2006).
## SPEC CPU Benchmark:
Example, highlights AMD performance

<table>
<thead>
<tr>
<th>Description</th>
<th>Name</th>
<th>Instruction count</th>
<th>CPI</th>
<th>Clock Cycle Time</th>
<th>Execution Time [seconds]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interpreted string processing</td>
<td>perl</td>
<td>2118</td>
<td>0.75</td>
<td>0.4</td>
<td>637</td>
</tr>
<tr>
<td>Block-sorting compression</td>
<td>bzip2</td>
<td>2389</td>
<td>0.85</td>
<td>0.4</td>
<td>817</td>
</tr>
<tr>
<td>GNU C compiler</td>
<td>gcc</td>
<td>1050</td>
<td>1.72</td>
<td>0.4</td>
<td>724</td>
</tr>
<tr>
<td>Combinational optimization</td>
<td>mcf</td>
<td>336</td>
<td>10.00</td>
<td>0.4</td>
<td>1,345</td>
</tr>
<tr>
<td>Go game (AI)</td>
<td>go</td>
<td>1658</td>
<td>1.09</td>
<td>0.4</td>
<td>721</td>
</tr>
<tr>
<td>Search gene sequence</td>
<td>hmer</td>
<td>2783</td>
<td>0.80</td>
<td>0.4</td>
<td>890</td>
</tr>
<tr>
<td>Chess game (AI)</td>
<td>sjeng</td>
<td>2176</td>
<td>0.96</td>
<td>0.4</td>
<td>837</td>
</tr>
<tr>
<td>Quantum computer simulation</td>
<td>libquantum</td>
<td>1623</td>
<td>1.61</td>
<td>0.4</td>
<td>1,047</td>
</tr>
<tr>
<td>Video compression</td>
<td>h264avc</td>
<td>3102</td>
<td>0.80</td>
<td>0.4</td>
<td>993</td>
</tr>
<tr>
<td>Discrete event simulation library</td>
<td>omnitpp</td>
<td>587</td>
<td>2.94</td>
<td>0.4</td>
<td>690</td>
</tr>
<tr>
<td>Games/path finding</td>
<td>aster</td>
<td>1082</td>
<td>1.79</td>
<td>0.4</td>
<td>773</td>
</tr>
<tr>
<td>XML parsing</td>
<td>xatancbmk</td>
<td>1058</td>
<td>2.70</td>
<td>0.4</td>
<td>1,143</td>
</tr>
</tbody>
</table>
SPEC CPU Benchmark:
Example, highlights AMD performance
SPEC CPU Benchmark:
Example, highlights AMD performance

- Comments:
  - There are programs for which the CPI is less than 1.
    - Suggests that multiple issue is at play
  - Why are there programs with CPI of 10?
    - The pipeline stalls a lot, most likely due to repeated cache misses and system memory transactions
Memory Aspects
SRAM

- **SRAM – Static Random Access Memory**
  - Integrated circuit whose elements combine to make up memory arrays
  - “Element”: is a special circuit, called flip-flop
  - One flip-flop requires four to six transistors
  - Each of these elements stores on bit of information
  - Very short access time: $\frac{1}{4}$ 1 ns (order of magnitude)
  - Uniform access time of any element in the array (yet it’s different to write than to read)
  - “Static” refers to the fact that once set, the element stores the value set as long as the element is powered
  - Bulky, since a storing element if “fat”; problematic to store a lot per unit area (compared to DRAM)
  - Expensive, since it requires four to six more transistors and different layout and support requirements
EXAMPLE: SRAM

- SRAM chip above stores 4 million elements, each with 8 bits of data - 4 MB
- To this end, you need
  - 22 bits to specify the address of the 8 bit slot that you want to address
  - Another control input that selects this chip ("Chip select" control signal)
  - A signal to indicate, when applicable, a write operation (Write enable)
  - A signal to indicate, when applicable, a read operation (Output enable)
  - 8 lanes for data to be sent in
  - 8 lanes for data to be collected
SRAM

- A write operation, for instance, requires a set-up time for the address line, a hold-time for the data line, and a “Write enable” pulse width

- Writing time is the combination of these three times
DRAM

• Idea behind DRAM type memory: the signal is stored as a charge in a capacitor
  - No charge: 0 signal
  - Some charge: 1 signal
• Drawback: capacitors leak, so the charge or lack of charge should be reinforced every so often) from where the name “dynamic” RAM
  - State of the capacitor should be refreshed every millisecond or so
• Refreshing requires a small delay in memory accesses
• Is this delay incurred often? (first order approximation answer)
  - Given frequency at which memory is accessed, refreshing every millisecond means issues might appear once every million cycles
  - Turns out that 99% of memory cycles are useful; refresh operations consume 1% of DRAM memory cycles

[Patterson & H]→

Monday, December 10, 12
DRAM

- Accessing data in SDRAM and DRAM memory chip different
- SRAM: dedicated address bus, any memory cell accessed rapidly (see Example & Figure)
- DRAM: uses two level decoder to access a memory cell
  - First a row in the memory chip is chosen
  - Then a column in the row is selected
- Same address line used for both operations
  - Makes things cheaper but slower; needs one more control signal
- The Row Access Strobe – RAS signal, indicates that the address line used to communicate a row address
- The Column Access Strobe – CAS signal, indicates that address line used to communicate offset in the row
Synchronous SRAM & DRAM

- Synchronous SRAM and DRAM: the same idea as SRAM and DRAM with one improvement: higher bandwidth of the memory access by sending out more data using a burst.
- Requires slight change of the hardware layout: a length should be made available to indicate how many consecutive bits are supposed to be involved in the burst memory transaction.
- Example: Reading 4 bytes at consecutive addresses:
  - Approach A: read them one a time - 4 addresses need to be provided, the operation is carried out 4 times.
  - Approach B (better): provide a start address and the number 4 to indicate that 4 bytes are involved in the transaction.
- Burst: in the example above, the process of streaming out 4 bytes in one transaction.
- SDRAM/SDRAM: the old SRAM/DRAM augmented with the ability to support burst transactions.
SRAM vs. DRAM: wrap-up

- Order of the SRAM access time: 0.5ns
  - Expensive but fast (On chip)
  - Needs no refresh

- Order of the DRAM access time: 50ns
  - Less expensive but slow (off chip)
  - Higher capacity per unit area
  - Needs refresh every 10-100 ms
  - Sensitive to disturbances

- Limit case: a 100X speedup if you can work off the SRAM
SRAM vs. DRAM: wrap-up

- Order of the SRAM access time: 0.5ns
  - Expensive but fast (On chip)
  - Needs no refresh

- Order of the DRAM access time: 50ns
  - Less expensive but slow (off chip)
  - Higher capacity per unit area
  - Needs refresh every 10-100 ms
  - Sensitive to disturbances

- Limit case: a 100X speedup if you can work off the SRAM

<table>
<thead>
<tr>
<th></th>
<th>Transistors per bit</th>
<th>Access Time</th>
<th>Persistent?</th>
<th>Sensitive?</th>
<th>Price</th>
<th>Applications</th>
</tr>
</thead>
<tbody>
<tr>
<td>SRAM</td>
<td>6</td>
<td>1X</td>
<td>Yes</td>
<td>No</td>
<td>100X</td>
<td>Cache memories</td>
</tr>
<tr>
<td>DRAM</td>
<td>1</td>
<td>10X</td>
<td>No</td>
<td>Yes</td>
<td>1X</td>
<td>Main Memory</td>
</tr>
</tbody>
</table>
# Feature Comparison Between Memory Types

<table>
<thead>
<tr>
<th>Feature</th>
<th>SRAM</th>
<th>DRAM</th>
<th>Flash</th>
</tr>
</thead>
<tbody>
<tr>
<td>Speed</td>
<td>Very fast</td>
<td>Fast</td>
<td>Very slow</td>
</tr>
<tr>
<td>Density</td>
<td>Low</td>
<td>High</td>
<td>Very high</td>
</tr>
<tr>
<td>Endurance</td>
<td>Good</td>
<td>Good</td>
<td>Poor</td>
</tr>
<tr>
<td>Power</td>
<td>Low</td>
<td>High</td>
<td>Very low</td>
</tr>
<tr>
<td>Refresh</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Retention</td>
<td>Volatile</td>
<td>Volatile</td>
<td>Non-volatile</td>
</tr>
<tr>
<td>Mechanism</td>
<td>Bi-stable Latch</td>
<td>Capacitor</td>
<td>Fowler-Nordheim tunneling</td>
</tr>
</tbody>
</table>
Consequence

- Since SRAM is expensive and bulkier, can’t have too much
  - Plagued by Space & Cost constraints

- Compromise:
  - Have some SRAM on-chip, making up what is called the “cache”
  - Have a lot of inexpensive DRAM off-chip, making up the “main memory”

- Hopefully your program will end up having a low “average memory access time” by hitting the cache repeatedly instead of taking costly trips to main memory
Fallout: Memory Hierarchy

- You now have a “memory hierarchy”

- Simplest memory hierarchy:
  - Main Memory + One Cache (typically called L1 cache)

- Middle of the road, consumer memory architectures typically have a deeper hierarchy, L1 + L2 + L3
  - L1 faster and smaller than L2
  - L2 faster and smaller than L3

- Note that all caches are typically on the chip
Example: Intel Chip Architecture

- Quad core Intel CPU die with the L3 cache highlighted
- For Intel I7 975 Extreme, cache hierarchy is as follows
  - 32 KB L1 cache / core
  - 256 KB L2 (Instruction & Data) cache / core
  - 8 MB L3 (Instruction & Data) shared by all cores
Memory Hierarchy

- Memory hierarchy is pretty deep:
Cache Types

- Two main types of cache

- **Data** caches feed processor with data manipulated during execution
  - If processor would rely on data provided by main memory the execution would be pitifully slow
    - Processor Clock significantly faster than the Memory Clock
    - Caches alleviate this memory pressure

- **Instruction** caches: used to store instructions
  - Much simpler to deal with compared to the data caches
    - Instruction use is much more predictable than data use

- In an ideal world, the processor would only communicate back and forth with the cache and avoid communication with the main memory
Split vs. Unified Caches

- Note that in the picture below L1 cache is split between data and instruction, which is typically the case.
- L2 and L3 (when present) typically unified.
How the Cache Works

- Assume simple setup with only one cache level L1

- Purpose of the cache: store for fast access a subset of the data stored in the main memory

- Data is moved at different resolutions between P $ C and between C $ M and
  - Between P and C: moved one word at a time
  - Between C and M: moved one block at a time (block called “cache line”)
Cache Hit vs. Cache Miss

- The processor typically agnostic about memory organization

- Middle man is the cache controller, which is an independent entity: it enables the “agnostic” attribute of the processor $\text{memory interaction}$

  - Processor requires data at some address
  - Cache Controller figures out if data is in a cache line
    - If yes: cache hit, processor served right away
    - If not: cache miss (data should be brought over from main memory! very slow)
  - Difference between cache hit and cache miss:
    - Performance hit related to SRAM vs. DRAM memory access
More on Cache Misses…

- A cache miss refers to a failed attempt to read or write a piece of data in the cache, which results in a main memory access with much longer latency.

- There are three kinds of cache misses:
  - Cache read miss from an instruction cache: generally causes the most delay, because the processor, or at least the thread of execution, has to wait (stall) until the instruction is fetched from main memory.
  - A cache read miss from a data cache: usually causes less delay, because instructions not dependent on the cache read can be issued and continue execution until the data is returned from main memory, and the dependent instructions can resume execution.
  - A cache write miss to a data cache: generally causes the least delay, because the write can be queued and there are few limitations on the execution of subsequent instructions. The processor can continue unless the queue is full and then it has to stall for the write buffer to partially drain.
Sensible Question

- Can you control what’s in the cache and anticipate future memory requests?
  - Typically not…
    - Any serious system has a hardware implemented cache controller with a mind of its own
  - There are ways to increase your chances of cache hits by designing software for high degree of memory access locality

- Two flavors of memory locality:
  - Spatial locality
  - Temporal locality
Spatial and Temporal Locality

- Spatial Locality for memory access by a program
  - A memory access pattern characterized by bursts of repeated requests for data that is physically located within the same memory region
  - “Bursts” because this accesses should happen in a sufficiently short interval of time (otherwise the cache line gets evicted)

- Temporal Locality for memory access by a program
  - Idea: If you access a variable at some time, then you’ll probably keep accessing the same variable for a while
  - Example: have a short for loop with some variables inside the loop! you keep accessing those variables as long as you run the loop
Cache Characteristics

- Size attributes: absolute cache size and cache line size
- Strategy for mapping of memory blocks to cache lines
- Cache line replacement algorithms
- Write-back policies

• NOTE: these characteristics carry over and become more convoluted when dealing with multilevel cache hierarchies
Cache Attributes:
Absolute Cache Size and Cache Line Size

- **Absolute Cache Size:** the bigger the better
  - It hasn’t gone up significantly in recent years
  - Typical sizes: 1-100 KB
  - Examples:
    - CPU: Intel I7 975 Extreme – 32KB L1 cache [per core]
    - GPU: NVIDIA Fermi Architecture – 64KB L1 cache [per SM]

- **Cache Line Size:**
  - Larger cache lines more tolerant in relation to spatial locality: good
  - Larger cache lines accommodate less cache lines: bad
  - A compromise should be found
    - Example: Intel Xeon X5650 – Line size is 64 Bytes, that is, 16 integers
    - Find this information on a Linux machine:
      ```bash
      $ cat /sys/devices/system/cpu/cpu0/cache/index0/coherency_line_size
      ```
Cache Attributes [1/3]:
Mapping of Memory Blocks to Cache Blocks

- Relevant when a memory transaction requires a main memory block to be brought in the cache and morph into a cache line
- The replacement policy decides where in the cache a copy of the block of main memory will go
- If the replacement policy is free to choose any entry in the cache to hold the copy, the cache is called **fully associative**.
- At the other extreme, if each entry in main memory can go in just one place in the cache, the cache is direct mapped.
- Many caches implement a compromise in which each entry in main memory can go to any one of N places in the cache, and are described as N-way set associative.

- Example: the L1 data cache in an AMD Athlon is 2-way set associative, which means that any particular location in main memory can be cached in either of two locations in the L1 data cache.
Cache Attributes [2/3]:
Mapping of Memory Blocks to Cache Blocks

- From “worse but simple” to “better but complex”:
  - Direct mapped cache—the best (fastest) hit times, and so the best tradeoff for "large" caches
  - 2-way set associative cache
  - 4-way set associative cache
  - Fully associative cache – the best (lowest) miss rates, and so the best tradeoff when the miss penalty is very high

- Why is fully associative the best?
  - If you can put anywhere you want, you don’t have to evict a cache line that in the past proved very successful
  - You might be forced to do so for a direct mapped cache since you have leeway there
If you have to evict a cache line and replace it with a main memory block, which cache line do you replace?

Question only relevant for a fully associative or set associative cache.

Two more common replacement methods:
- LRU: Least Recently Used
- LFU: Least Frequently Used

There is also a strategy that randomly chooses the line to evicted:
- Good since it is very simple to implement/support, unlike LRU and LFU.
Intel® Xeon® CPU E5520 Info

- Where applicable, figures report byte sizes

```bash
>> getconf -a | grep -i cache
```

<table>
<thead>
<tr>
<th>Cache Level</th>
<th>Size</th>
<th>Association</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1_ICACHE_SIZE</td>
<td>32768</td>
<td>4</td>
</tr>
<tr>
<td>L1_ICACHE_ASSOC</td>
<td></td>
<td></td>
</tr>
<tr>
<td>L1_ICACHE_LINESIZE</td>
<td>32</td>
<td></td>
</tr>
<tr>
<td>L1_DCACHE_SIZE</td>
<td>32768</td>
<td>8</td>
</tr>
<tr>
<td>L1_DCACHE_ASSOC</td>
<td></td>
<td></td>
</tr>
<tr>
<td>L1_DCACHE_LINESIZE</td>
<td>64</td>
<td></td>
</tr>
<tr>
<td>L2_CACHE_SIZE</td>
<td>262144</td>
<td>8</td>
</tr>
<tr>
<td>L2_CACHE_ASSOC</td>
<td></td>
<td></td>
</tr>
<tr>
<td>L2_CACHE_LINESIZE</td>
<td>64</td>
<td></td>
</tr>
<tr>
<td>L3_CACHE_SIZE</td>
<td>8388608</td>
<td>16</td>
</tr>
<tr>
<td>L3_CACHE_ASSOC</td>
<td></td>
<td></td>
</tr>
<tr>
<td>L3_CACHE_LINESIZE</td>
<td>64</td>
<td></td>
</tr>
</tbody>
</table>
Write Policies

- Framework:
  - A block of main memory, B, is stored as a cache line C
  - The actual value of a variable foo whose address is in B is modified by the processor through a “write access” instruction

- Q: When the new value of foo overwrites the old value of foo, how is this process implemented?

- A: Process implemented according to a “write policy”
Write Policies

- When issuing the write access instruction on foo, note that the cache is always updated.
  - The question becomes: do you update the main memory block B right away, or do you delay this action?

- Two common policies:
  - Write-through: you update B immediately after updating C
  - Write-back: you push the update of B back as much as possible
The Write-Through Policy

- The good part
  - As soon as C is modified, B is modified and therefore the correct value of foo becomes visible to anyone accessing the main memory (I/O devices, or a different processor, for instance)

- The tricky part
  - Implemented in a naïve form, the policy would stall processor execution since changing B after changing C requires hundreds of wasted cycles on the processor side

- Solution to tricky part
  - Have a “write buffer” that stores the new value of foo, which allows the control to go back to the processor issuing the write access instruction right away
  - Then, the content of the “write buffer” is flushed to the main memory in B
    - You hide the flushing process with useful execution since the processor gain control of execution and can issue instructions while the cache controller deals with the flushing of the write buffer
    - When things can go south: if the write buffer is filled faster than it can be drained. If buffer reaches limit the processor doesn’t gain control of the execution for a while (to make room in the “write buffer”)

Monday, December 10, 12
The Write-Back Policy

- The good part
  - You might actually update B very rarely although the entry in C has been updated multiple times

- The tricky part
  - You might have another processor or I/O device try read foo from the main memory (in B). The value can be stale, since a newer version is available in C

- Solution to tricky part
  - Any reads of data stored in B would be made through the cache: for instance, a different processor reading the value of foo, would go to C to pick up the actual value

- An implementation detail for write-back policy
  - When a cache line is evicted, the cache controller must be careful: was this line of cache modified? If yes, then the content needs to be updated in B. If not, do nothing
    - There is a “dirty cache” bit the cache controller plays with: if the content of the cache line is modified it is set to 1. Upon loading a line from main memory the “dirty cache” bit is set to zero
Side Note 1: On Write Policy

- Framework: Assume now that the processor issues a write access instruction on foo, and foo is not in cache but only in the main memory.

- Q: How is this scenario handled?

- A: There are two possible rules.

- Alternative 1: Bring B into C, and write foo in C. After that, fall back on the previous discussion by choosing one of the two write policy.
  - This is called the “write-allocate” method.

- Alternative 2: Simply write foo in B. The cache not part of this transaction.
  - This is called the “write-no-allocate” method.

- NOTE: Alternative 1 is the more common one (write-allocate).
Side Note 2: On Multiple Caches

- Most modern consumer grade chips contain several cores and several caches (very often 3 of them)

- The cache lines in L1 are a subset of the cache lines in L2, which are a subset of the cache lines in L3, which are subset of memory blocks in the main memory

- Caches: middle of the road sizes and latencies
  - L1: 16 – 64 KB, accessible in 2-3 clock cycles
  - L2: 256 KB – 4 MB, accessible in 20 cycles
  - L3: 6 – 8 MB, accessible in 100-200 clock cycles
New Topic: Cache Coherence

- Framework: Assume the following
  - Two processors share the same main memory
  - Each processor has a L1 cache
  - A write-back cache policy is observed

- What happens in the following scenario?
  - Main Memory Block B is represented as a L1 cache line C1 for the first processor and as a L1 cache line C2 for the second processor
  - Processor-1 (P1) changes value of foo, which is stored in C1, write back policy
  - Later on, but before the C1 write-back, Processor-2 (P2) wants to use foo.

- Question: Will P2 get a stale value or not?
  - In an architecture that is cache coherent, P2 uses the up-to-date value for foo, as stored in C1. That is, it gets the most recently written value for a specific memory address (associated with foo in this case).
Cache Coherence Issues: Not Write-Back Specific…

- Framework: Assume that
  - We have four time sequences $t_1 < t_2 < t_3 < t_4$
  - No cache eviction occurs on either of the two processors in this scenario

- Consider this scenario:
  - $t_1$: P1 reads $foo=0$ from main memory and places it in C1 since it uses it
  - $t_2$: P2 reads $foo=0$ from main memory and places it in C2 since it uses it
  - $t_3$: P1 modified $foo$ so that $foo=1$. Write-through policy in place, main memory gets updated right away with $foo=1$
  - $t_4$: P2 wants to use $foo$ for the second time.

- Question: Which $foo$ value will P2 end up using?
  - Cache coherence is relevant again
Cache Coherence: Formal Definition

- In a multi-processor architecture, a memory system is coherent if:

1. A read by a processor P to a location X that follows a write by P to X, with no writes of X by another processor occurring between the write and the read by P, always returns the value written by P.

2. A read by a processor to location X that follows a write by another processor to X returns the written value if the read and write are sufficiently separated in time and no other writes to X occur between the two accesses.

3. Writes to the same location are serialized; that is, two writes to the same location by any two processors are seen in the same order by all processors. For example, if the values 1 and then 2 are written to a location, processors can never read the value of the location as 2 and then later read it as 1.
Cache Coherence: Formal Definition

- In a multi-processor architecture, a memory system is coherent if:

  1. A read by a processor P to a location X that follows a write by P to X, with no writes of X by another processor occurring between the write and the read by P, always returns the value written by P.

  2. A read by a processor to location X that follows a write by another processor to X returns the written value if the read and write are sufficiently separated in time and no other writes to X occur between the two accesses.

  3. Writes to the same location are serialized; that is, two writes to the same location by any two processors are seen in the same order by all processors. For example, if the values 1 and then 2 are written to a location, processors can never read the value of the location as 2 and then later read it as 1.
The “Sufficiently Separated” Issue

1. If a write of X by \( P_i \) precedes a read of X by a different processor \( P_j \) by a very short period, it can be impossible to guarantee that the read by \( P_j \) would return the value generated by \( P_i \)

2. The issue of exactly when the value of X must be seen by \( P_j \) is specified as part of the “memory consistency model”
   - In the past, the memory consistency model was interpreted as the policy that places an early and late bound on when a new value can be propagated to any given processor [Gharachorloo 1995]

3. NOTE: On a multiprocessor platform, the concepts of “memory consistency model” and “cache coherence” combine to meet, among several requirements, a common sense expectation that any read of a data item returns the most recently written value of that data item
Cache Coherence Protocols

• Framework
  • An architecture w/ multiple processors
  • Each processor has one or several caches

• The need of a protocol
  • In spite of storing in its caches multiple copies of main memory blocks, each processor must have and also enable through its caches a coherent global view of the main memory

• Addressing the need
  • Two common cache coherence protocols have been used to track the state of shared blocks of main memory data
    • Snooping protocols
    • Directory-based protocols
Snooping Protocols

- Basic idea: in a multiprocessor architecture, all writes to main memory are channeled through a bus that is snooped by the cache controller[s] associated with each processor
Snooping Protocols

- Limitation: works only when there is a bus (broadcast medium) that can be snooped so that each cache controller becomes aware of *all* write accesses to main memory
  - At the cornerstone of the solution is the ability to broadcast to all cache controllers by snooping on a bus (or some other critical location) that serves as the conduit for all write accesses to the main memory

- Pro: Very simple to implement

- Con: Like any other solution that relies on a centralized step or stage (broadcast from a global bus, in this case), snooping is bound the become a bottleneck for systems with many processors
Snooping Protocols

- Snooping protocols belong to one of two classes
  - Protocols for Caches with Write-Through Policy
  - Protocols for Caches with Write-Back Policy

- Discussion focuses on a protocol for write-through policy
  - For snooping for caches with write-back policy, see references [H&P4th, C&S&G99, R&R2010]

- For write-through, snooping can implemented as
  - An update-based protocol
    - As soon as a $P_i$ cached value is changed in the main memory by processor $P_j$, the appropriate line of cache associated with $P_i$ is updated to reflect the new value
  - An invalidation-based protocol
    - As soon as a $P_i$ cached value is changed in the main memory by processor $P_j$, the appropriate line of cache associated with $P_i$ is marked as invalid
Snooping Protocol
[for Caches w/ Write-Through Policy]

- Note that an invalidation-based protocol is like a lazy update of the cached variable
  - The P_i cached value is actually updated only if and when it is used by P_i
  - Strategy can result in efficiency gains, since the value of the variable can be changed many times by many other processors before processor P_i actually needs to use the variable, and therefore must update its value from main memory

- How is this approach faring these days?
  - After 2006, every chip with more than two processors used an interconnect other than a shared bus [H&P4th]. Snooping becomes more complicated since there is no shared bus to snoop on. Moreover, this aspect hurts since the serialization attribute that induced atomicity to the protocol ceases to exist
  - For more details, see [H&P4th]
Directory-Based Protocols

- **Preamble:**
  - For snooping protocols, each processor, at the level of its cache controller, maintains the sharing status of each cache line ultimately tied to a certain block of the main memory.
  - All these distributed (across processors) status states should be updated globally as various processors issue write accesses to the main memory.
    - If many processors are present, this update represents a bottleneck.

- **Central idea, directory-based protocols**
  - The sharing status of a block of main memory is kept in just one location, called the “directory”. Not distributed anymore as it used to be the case for snooping protocols.

- **Reality check:**
  - Directory-based coherence has slightly higher implementation overhead than snooping, but it can scale to larger processor counts.
Directory-Based Protocols

- In a directory-based protocol, instead of snooping on a globally shared resource, such as a memory bus, a cache control can interrogate the directory for the state of a main memory block.

- A directory maintains the status of each main memory block that can be cached.

- The “status” includes information about which cache stores this block, the state in which this cache is, etc.

- The same directory protocol can be used for both an SMP architecture with shared memory, say a 16 core AMD Opteron, or for a distributed memory configuration that communicates through an Infiniband interconnect.
Directory-Based Protocols: The Basics

- In the most rudimentary implementation, the directory stores one field for each main memory block.

- Size is then proportional to the number of blocks times the number of processors.

- The directory is distributed (spread out) along with the memory:
  - In a shared memory, SMP type, it is interleaved with the memory banks.
  - Why this partitioning?
    - If there is only one “entry” point to this directory, it will act as a bottleneck and work against the fundamental scalability premise associated with a directory-based protocol.
    - NOTE: The key observation is that even when distributed, there is no redundancy in the directory – one memory block is registered with the directory only once.
Directory-Based Protocols: The Basics

- Assume there are $n$ processors. Example: $n = 16$ processors $P_1$ through $P_{16}$ and therefore there are 16 caches, $C_1$ through $C_{16}$
- Each processor $P_i$, $1 \leq i \leq n$, has a directory $D_i$ associated with the local memory $M_i$
- $M_i$ has a number of memory blocks $b_i$. Example: $b_i = 1024$ memory blocks in $M_i$
- $D_i$ maintains for each memory block $k$ an array $p_i[k]$ of length $n$ of so called “presence bits”
  - Specifically, for $1 \leq k \leq b_i$ and $1 \leq j \leq n$, $p_i[k][j] = 1$ if processor $P_j$ has a valid copy of local memory block $k$ in its local cache; $p_i[k][j] = 0$ otherwise
- Additionally, for each memory block in $M_i$, there is one more dirty bit $d b_k$, $1 \leq k \leq b_i$

- Each processor $P_i$ has a directory $D_i$ associated with the local memory $M_i$
- $M_i$ has a number of memory blocks $mB$. Example: $mB=1024$ memory blocks in $M_i$
Directory-Based Protocols: The Basics

- Each processor Pi has a directory Di associated with the local memory Mi.
- Mi has a number of memory blocks mB. Example: mB=1024 memory blocks in Mi.
- Assume there are n processors. Example: n=16 processors P1 through P16 and therefore there are 16 caches, C1 through C16.
- Di maintains for each memory block an array of “presence bits” of length n.
- Additionally, for each memory block,
From Cache Coherence to Memory Consistency Models

- Cache coherence focuses on *one* memory location and tries to put order into the chaos associated with accessing that location. So the focus is on the location of *one* variable X.

- The problem becomes more interesting if one is interested in understanding how the *entire* memory works when you have multiple processors accessing it.

- Moving from a local to a global perspective brings into forefront the issue of memory consistency model.

- The issue of consistency: if several processors share the same memory and they start changing variables, how soon and in what sequence would these changes become visible to other processors in the pool that share the memory?