Advanced Computing for Engineering Applications

Dan Negrut

Simulation-Based Engineering Lab
Wisconsin Applied Computing Center
Department of Mechanical Engineering
Department of Electrical and Computer Engineering
University of Wisconsin-Madison

Milano
18-23 November
2013
Before We Get Started…

● Goal
  ● Spend five days getting familiar with how parallel computing can help you

● Reaching this goal
  ● Cover some basics about computing at large (day 1)
  ● Spend two days on parallel computing with GPU cards (day 2 and 3)
  ● Spend one day on parallel computing with OpenMP (day 4)
  ● Spend last day on parallel computing with MPI (day 5)

● The material will probably be outside your comfort zone
  ● Take it as an opportunity to break into something new
A Couple of Things About Myself

- 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 an HPC class I’m teaching this semester at UW-Madison
  - GPU Computing, OpenMP, and MPI
  - Class material available online (slides & audio streaming): [http://sbel.wisc.edu/Courses/ME964/2013/](http://sbel.wisc.edu/Courses/ME964/2013/)

- Today’s material is based on information available here:
  [http://sbel.wisc.edu/Courses/ME964/Literature/primerHW-SWinterface.pdf](http://sbel.wisc.edu/Courses/ME964/Literature/primerHW-SWinterface.pdf)
Today’s Computer

- Follows 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
From Code to Instructions

- There is a difference between a line of code and a processor instruction.

Example:
- Line of C code:
  
  \[
  a[4] = \text{delta} + a[3]; //\text{line of C code}
  \]

- MIPS assembly code generated by the compiler:

  \[
  \begin{align*}
  \text{lw} & \quad $t0, 12($s2) \quad \# \text{reg } \$t0 \text{ gets value stored 12 bytes from address in } \$s2 \\
  \text{add} & \quad \$t0, \quad \$s4, \quad \$t0 \quad \# \text{reg } \$t0 \text{ gets the sum of values stored in } \$s4 \text{ and } \$t0 \\
  \text{sw} & \quad \$t0, \quad 16($s2) \quad \# \ a[4] \text{ gets the sum } \text{delta} + a[3]
  \end{align*}
  \]

- Set of three corresponding MIPS instructions produced by the compiler:

  \[
  \begin{align*}
  1000111001001001000000000000001100 \\
  00000010100010001000000100000010000 \\
  1010111001001000000000000000010000
  \end{align*}
  \]
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 (most of the time)
- Assembly line of code can be regarded as an instruction that is expressed in a way that humans can relatively easily 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 draw on two different Instruction Set Architectures (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**

```assembly
call ___main
fldl LC0
fstpl -40(%ebp)
fldl LC1
fstpl -32(%ebp)
fldl LC2
fstpl -24(%ebp)
fldl LC0
faddl -24(%ebp)
ldl LC0
faddl %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 1079902208
```

---

**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"
```
Any sufficiently advanced technology is indistinguishable from magic.
-- Arthur C. Clark
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
RISC vs. CISC

- **RISC Architecture – Reduced Instruction Set Computing Architecture**
  - Usually each instruction is coded into a set of 32 bits
  - Recently a move to 64 bits
  - Each executable has fixed length instruction be it 32 or 64 (no mixing)
  - The key attribute: the length of the instruction is fixed
  - Promoted by: ARM Holding, company that started as ARM (Advanced RISC Machines)
    - Use in: embedded systems, smart phones – Intel, NVIDIA, Samsung, Qualcomm, Texas Instruments
    - Somewhere between 8 and 10 billion chips based on ARM manufactured annually

- **CISC Architecture – Complex Instruction Set Computing Architecture**
  - Instructions have various lengths
    - Examples: 32 bit instruction followed by 256 bit instruction followed later on by 128 bit instruction, etc.
  - Intel’s X86 is the most common example
  - Promoted by: Intel, AMD
    - Used in: laptops, desktops, workstations, supercomputers
RISC vs. CISC

- RISC is simpler to comprehend, provision for, and work with

- Decoding CISC leads to extra power consumption and makes things more complicated

- A CISC instruction is usually broken down into several micro-operations (uops)

- CISC Architectures invite spaghetti type evolution of the ISA and require complex microarchitecture
  - Provide the freedom to do as you wish
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 this (on 32 bits, MIPS, \texttt{lw \$t0, 12($s2))}:
  
  \begin{verbatim}
  10001110010010000000000000001100
  \end{verbatim}

- 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

  \begin{verbatim}
  op (6 bits) | rs (5 bits) | rt (5 bits) | constant or address (16 bits)
  \end{verbatim}

  32 bits
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 has 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
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 (this is the ALU) starts to cook a meal
  - 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 passing of meat, passing of pasta, the movement of the sautéed meat to chopping board, 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)
FDX Cycle: The Execution Part
It All Boils Down to Transistors…

- Why are transistors important?

- Transistors can be organized to produce complex logical units that have the ability to execute instructions

- More transistors increase opportunities for building/implementing in silicon functional units that can operate at the same time towards a shared goal
Transistors at Work: AND, OR, NOT

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

<table>
<thead>
<tr>
<th>AND</th>
<th>in₂=0</th>
<th>in₂=1</th>
</tr>
</thead>
<tbody>
<tr>
<td>in₁=0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>in₁=1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>OR</th>
<th>in₂=0</th>
<th>in₂=1</th>
</tr>
</thead>
<tbody>
<tr>
<td>in₁=0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>in₁=1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>NOT</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>in₁=0</td>
<td>1</td>
</tr>
<tr>
<td>in₁=1</td>
<td>0</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

<table>
<thead>
<tr>
<th>Truth Table</th>
<th>Logic Equation:</th>
</tr>
</thead>
<tbody>
<tr>
<td>in&lt;sub&gt;1&lt;/sub&gt;</td>
<td>in&lt;sub&gt;2&lt;/sub&gt;</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

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

- 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
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>( \text{in}_1 )</td>
<td>( \text{in}_2 )</td>
<td>( \text{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>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

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

- 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

- Transistor
  - Take one or combine two of them
  - Examples: AND, OR, NOT

- Gate
  - Combine a couple of them
  - Requires a “Control Signal” as input
  - Example: __
    - \( \text{out} = in_1 + in_2 \)

- Mux (selector)
  - Combine a couple of them

- Logical Unit

- Complex Combinational Block
  - Example: One bit adder
  - Place instances in an array
  - Example: 32 bit adder

- Array of Logic Elements

From simple to complex…
Every 18 months, the number of transistors per unit area doubles (Moore’s Law)

- Current technology (2013): feature length is 22 nm (by Intel)
- Next wave (2014): feature length is 14 nm

Example

- NVIDIA Fermi architecture of 2010:
  - 40 nm technology
  - Chips w/ 3 billion transistors → more than 500 scalar processors, 0.5 TFlops
- NVIDIA Kepler architecture 2012:
  - 28 nm technology
  - Chips w/ 7 billion transistors → more than 2000 scalar processors, 1.5 TFlops
Chip Feature Length: Intel’s Roadmap

- Moore’s law at work
  - 2013 – 22 nm
  - 2015 – 14 nm
  - 2017 – 10 nm
  - 2019 – 7 nm
  - 2021 – 5 nm
  - 2023 – ??? (carbon nanotubes?)
Registers
Registers

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

- CU – responsible for controlling the process that will deliver the request baked into the instruction

- 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
  - This data needs to be stored somewhere
Registers

- Registers, quick facts:
  - A register is an entity whose role is that of storing information
  - A register is the type of storage with shortest latency – it’s closest to the ALU
  - Typically, one cannot control what gets kept in registers (with a few exceptions)

- The number AND size of registers used are specific to a ISA
  - Prime example of how ISA decides on something and the microarchitecture has to do what it takes to implement this design decision

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

- Discussion herein covers only 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 memory or, alternatively, produced by the CPU and waiting to be stored in memory

- Memory Address Register (MAR) – the address of the memory location in 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.]

- Registers on previous two slides are a staple in most chip designs.

- There are several other registers that are common to many 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
Register Types [Cntd.]

- 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 (top of the stack)
  - Frame Pointer (fp) - a register that holds an address that points to the beginning of the procedure frame (for instance, the previous sp before this function changed it’s value)
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 bit 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 design of the chip (the microarchitecture)
  - Need to work out the control flow
Pipelining
Charlie Chaplin - Modern Times (1936)
Pipelining, or the Assembly Line Concept

- Henry Ford: perfected the assembly line idea on an industrial scale and in the process shaped the automotive industry (Ford Model T)

- 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
Intro

- 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: Data access
  - Stage 5: Write-back into register file

[Patterson, 4th edition]→
Pipelining, Basic Idea

- At the cornerstone of pipelining is the observation that the following tasks can be worked upon 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
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 the 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 by overlapping different stages
Pipelining Hazards

- Q: if deep pipelines are good, why not have them deeper and deeper?
- A: deep pipelines plagued by “pipelining hazard”
  - These “hazards” come in three flavors
    - Structural hazards
    - Data hazards
    - Control hazards
Pipeline Structural Hazards

- The instruction pipelining analogy with 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
  - Not encountered when you assemble the same car model (things are deterministic in this case)
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 the same register to store a temporary variable.
  ● Resolution: there should be enough registers provisioned so that no combination of instructions in the pipeline leads 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
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 (1st 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 (2nd 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.
- Two examples to show when this might come in handy:
  - 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;
float c, d;
// some code a, b, c, d
a += b;
c += d;
```
Pipelining vs. Multiple-Issue

- On average, more than one instruction is processed by the same core in the same clock cycle from the same instruction stream.

- Multiple-Issue can be done statically or dynamically:
  - Static multiple-issue:
    - Predefined, doesn't change at run time
    - Who uses it: NVIDIA - very common in parallel computing on the GPU
  - Dynamic multiple-issue:
    - Changed at run time by using hardware resources that can take additional work
    - Who uses it: Intel, uses it heavily

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

- The data dependencies between instructions being processed take place at run time.
- Checking for dependencies is complex, requires high cost in time and energy.
- We always have to check to make sure the result is ok.
- 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 tasks, 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:
  
  \[
  \text{CPU Exec. Time} = \text{Instruction Count} \times \text{CPI} \times \text{Clock Cycle Time}
  \]

- Alternatively, using the clock rate

  \[
  \text{CPU Exec. Time} = \frac{\text{Instruction Count} \times \text{CPI}}{\text{Clock Rate}}
  \]
CPU Performance: How can we improve it?

- To improve performance the product of three factors should be reduced
- For a long time, we 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

### CINT2006 Programs

<table>
<thead>
<tr>
<th>Description</th>
<th>Name</th>
<th>Instruction count [×10⁹]</th>
<th>CPI</th>
<th>Clock Cycle Time [seconds ×10⁻⁹]</th>
<th>CPU Exec. 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>hmmer</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>

[Patterson, 4th edition]→
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
The Most Important Lesson

- The cost of memory transactions trumps by far the cost of number crunching

- Number crunching is free, sustaining the number crunching is the hard part
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 one 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

[Patterson & H]→
DRAM

- DRAM type memory: the signal is stored as a charge in a capacitor
  - No charge: 0 signal
  - Some charge: 1 signal

- The good: cheap, requires only one capacitor and one transistor

- The bad: 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]→
SRAM vs. DRAM: wrap-up

- Order of the SRAM access time: 0.5ns
  - Expensive but fast
  - It’s mostly on chip
  - Needs no refresh

- Order of the DRAM access time: 50ns
  - Less expensive but slow
  - It’s mostly 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></th>
<th>SRAM</th>
<th>DRAM</th>
<th>Flash</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Speed</strong></td>
<td>Very fast</td>
<td>Fast</td>
<td>Very slow</td>
</tr>
<tr>
<td><strong>Density</strong></td>
<td>Low</td>
<td>High</td>
<td>Very high</td>
</tr>
<tr>
<td><strong>Endurance</strong></td>
<td>Good</td>
<td>Good</td>
<td>Poor</td>
</tr>
<tr>
<td><strong>Power</strong></td>
<td>Low</td>
<td>High</td>
<td>Very low</td>
</tr>
<tr>
<td><strong>Refresh</strong></td>
<td>No</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td><strong>Retention</strong></td>
<td>Volatile</td>
<td>Volatile</td>
<td>Non-volatile</td>
</tr>
<tr>
<td><strong>Mechanism</strong></td>
<td>Bi-stable Latch</td>
<td>Capacitor</td>
<td>Fowler-Nordheim tunneling</td>
</tr>
</tbody>
</table>
Cost and Speed Implications

- 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 has 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)

- Today’s memory architectures typically have 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 that illustrates L3 cache
- For Intel Core 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 deep:

Moving on to talk about caches
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 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 P → M 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 plus overhead
More on Cache Misses…

- A cache miss refers to a failed attempt to read/write a piece of data from/to 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.

[Wikipedia]
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 for loop with some variables inside the loop! you keep accessing those variables as long as you 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
The Concept of Virtual Memory
Motivating Questions/Issues

- Assumption: we are not talking about embedded systems, which are running alone on a processor and basically do not require an operating system to play the role of the middle man.

- Question 1: On a 32 bit machine, how come you can have 512MB of main memory yet allocate an array of 1 GB?

- Question 2: How can you compile a program on a Windows workstation with 2 GB of memory and run it later on a different laptop with 512 MB of memory?

- Question 3: How can several processes run seemingly at the same time on a processor with one thread?
The three questions raised on previous slide answered by the interplay between the compiler, the operating system (OS), and the execution model adopted by the processor

When you compile a program there is no way to know where in the physical memory the code will get its data allocated

- There are other “tenants” that inhabit the memory, and they are there before you get there

The solution is for the code to be compiled and assumed to lead to a process that executes in a virtual world in which it has access to 4 GB of memory (on 32 bit systems).

- The “virtual world” is called the virtual memory space
Virtual vs. Physical Memory

- Virtual memory: this nice and immaculate space of $2^{32}$ addresses (on 32 bit architectures) in which a process sees its data being placed, the instructions stored, etc.

- Physical memory: a busy place that hosts at the same time data and instructions associated with tens of applications running on the system
Anatomy of the Virtual Memory

STACK segment
[stores a collection of frames, each associated w/ one function call]
[a stack frame stores function params., return addresses, local vars., etc.]
[last-in-first-out (LIFO) structure; push/pop managed]

STACK OVERFLOW
[if top of stack reaches beyond this logical address]

HEAP segment
[segment used when program allocates memory dynamically, at run time]
[managed by the OS in response to function calls like malloc, free, etc.]

BSS segment
[stores uninitialized global and static variables]

DATA segment
[stores static variables and initialized global variables]

TEXT segment
[stores instructions associated with the program]
The Anatomy of the Stack

- Function `bar` and associated stack frame

```
float bar(int a, float b)
{
    int initials[2];
    float t1, t2, t3;
    //...code here..
    //...no other variables..
    return t1;
}
```
Virtual memory allows the processor to work in a virtual world in which each process, when run by the processor, seems to have exclusive access to a very large memory space.

For 32 bits: memory space is 4 GB big.

This virtual world is connected back to the physical memory through a Page Table.
Anatomy of a Virtual Memory Address

- A virtual address has two parts: the page number, and the offset
Anatomy of a Virtual Memory Address

- A page of virtual memory corresponds to a frame of physical memory

- The size of a page (or frame, for that matter) is typically 4096 bytes

- $2^{12} = 4096$: 12 address bits are sufficient to relatively position each byte in a page
The Translation Process

- Example: imagine that your physical memory is 2 GB
- The physical address has 31 bits: $2^{31} = 2\text{GB}$
- Then the page table converts bits 12 through 31 of the virtual address into bits 12 through 30 of the physical address
Short Digression 1: The Unit of Address Resolution

- How many bits are available for data storage at each address?
- Example:
  - We have $2^{32}$ addresses that we can access
  - If each address points to a location that stores 8 bits (one byte) then we have 4 GB of addressable memory
  - However, if each address refers to a location that stores 2 bytes, we have 8 GB of addressable memory
- Intel and AMD CPUs: the unit of address resolution is 1 byte (8 bits)
- Consequence: the Intel 32 bit processors “see” a virtual memory space that can be 4 GB big
Short Digression 2: The 32 to 64 bit Migration

- If the architecture and OS have 32 bits to represent addresses, it means that $2^{32}$ addresses can be referenced.

- If unit of address resolution is 1 byte, that means that the size of the virtual memory space can be 4 GB.

- This is hardly enough today when programs are very large and the amounts of data they manipulate can be staggering.

- This motivated the push towards having addresses represented using 64 bits: the memory space balloons to $2^{64}$ bytes, that is 16 times 1152921504606846976 bytes.
Short Digression 3: The 32 to 64 Bit Migration

- Note that a 64 bit architecture typically calls for two things:

- From a **hardware** perspective, the size of the registers, integer size, and word size is 64 bits

- From a **software** perspective, the addresses are now 64 bits and therefore a program “operates” in a huge virtual memory space
  - The operating system (OS) is the party managing the execution of a program in the 64 bit universe
Comments on the Page Table
Preamble to TLB.

- The page table is the key ingredient that allows the translation of virtual addresses into physical addresses.

- Every single process executing on a processor and managed by the OS has its own page table.

- Page table is stored in main memory.
  - For a 32 bit operating system size of a page table can be up to 4 MB in size.
Comments on the Page Table. The TLB

- If Page Table stored in main memory it means that each address translation would require a trip to main memory
  - This would be very costly

- There is a “cache” for this translation process: TLB
  - Translation lookaside buffer: holds the translation of a small collection of virtual page numbers into frame IDs

- Best case scenario: the TLB leads to a hit and allows for quick translation
- Bad scenario: the TLB doesn’t have the required information cached and a trip to main memory is in order
- Worst scenario: the requested frame is not in main memory and a trip to secondary memory is in order
  - Called “page fault”
Illustration: The Role of the TLB

- A TLB is just like a cache
- A TLB miss leads to substantial overhead in the translation of an address
Memory Access: The Big Picture

- A simplified version of how a memory request is serviced presented below
Parallel Computing: Why? & Why Now?
This Segment’s Main Points

- Sequential computing has been losing steam recently

- The immediate future seems to belong to parallel computing
Acknowledgements

- Material presented today includes content due to
  - Hennessy and Patterson (Computer Architecture, 4th edition)
  - John Owens, UC-Davis
  - Darío Suárez, Universidad de Zaragoza
  - John Cavazos, University of Delaware
  - Others, as indicated on various slides
  - I apologize if I included a slide and didn’t give credit where was due
CPU Speed Evolution

[log scale]
...we can expect very little improvement in serial performance of general purpose CPUs. So if we are to continue to enjoy improvements in software capability at the rate we have become accustomed to, we must use parallel computing. This will have a profound effect on commercial software development including the languages, compilers, operating systems, and software development tools, which will in turn have an equally profound effect on computer and computational scientists.

John L. Manferdelli, Microsoft Corporation Distinguished Engineer, leads the eXtreme Computing Group (XCG) System, Security and Quantum Computing Research Group
Three Walls to Serial Performance

- Memory Wall
- Instruction Level Parallelism (ILP) Wall
- Power Wall

Not necessarily walls, but increasingly steep hills to climb


http://www.ctwatch.org/quarterly/articles/2007/02/the-many-core-inflection-point-for-mass-market-computer-systems/
Memory Wall

- Memory Wall: What is it?
  - The growing disparity of speed between CPU and memory outside the CPU chip.

- Memory latency is a barrier to computer performance improvements
  - Current architectures have ever growing caches to improve the average memory reference time to fetch or write instructions or data

- Memory Wall: due to latency and limited communication bandwidth beyond chip boundaries.
  - From 1986 to 2000, CPU speed improved at an annual rate of 55% while memory access speed only improved at 10%
Memory Bandwidths
[typical embedded, desktop and server computers]
Memory Speed:
Widening of the Processor-DRAM Performance Gap

- The processor: victim of its own success
  - So fast it left the memory behind
  - A system (CPU-Memory duo) can’t move as fast as you’d like (based on CPU top speeds) with a sluggish memory

- Plot on next slide shows on a *log* scale the increasing gap between CPU and memory

- The memory baseline: 64 KB DRAM in 1980

- Memory speed increasing at a rate of approx 1.07/year
  - However, processors improved
    - 1.25/year (1980-1986)
    - 1.52/year (1986-2004)
    - 1.20/year (2004-2010)
Memory Speed: Widening of the Processor-DRAM Performance Gap

Courtesy of Elsevier, Computer Architecture, Hennessey and Patterson, fourth edition
Memory Latency vs. Memory Bandwidth

- **Latency**: the amount of time it takes for an operation to complete
  - Measured in seconds
  - The utility “ping” in Linux measures the latency of a network
  - For memory transactions: send 32 bits to destination and back, measure how much time it takes! gives you latency

- **Bandwidth**: how much data can be transferred per second
  - You can talk about bandwidth for memory but also for a network (Ethernet, Infiniband, modem, DSL, etc.)

- **Improving Latency and Bandwidth**
  - The job of colleagues in Electrical Engineering
  - Once in a while, Materials Science colleagues deliver a breakthrough
  - Promising technology: optic networks and layered memory on top of chip
Memory Latency vs. Memory Bandwidth

- Memory Access Latency is significantly more challenging to improve as opposed to improving Memory Bandwidth

- Improving Bandwidth: add more “pipes”.
  - Requires more pins that come out of the chip for DRAM, for instance.
  - Adding more pins is not simple – very crowded real estate plus the technology is tricky

- Improving Latency: no easy answer here

- Analogy:
  - If you carry commuters with a train, add more cars to a train to increase bandwidth
  - Improving latency requires the construction of high speed trains
    - Very expensive
    - Requires qualitatively new technology (Elon Musk’s Hyperloop)
Latency vs. Bandwidth Improvements Over the Last 25 years

Courtesy of Elsevier, Computer Architecture, Hennessey and Patterson, fourth edition
The 3D Memory Cube
[possible breakthrough?]

- Micron's Hybrid Memory Cube (HMC) features a stack of individual chips connected by vertical pipelines or “vias,” shown in the pic.
- 2014: Micron to release 2 and 4 GB HMC
  - Major step forward

- HMC prototypes clock in with bandwidth of 128 gigabytes per second (GB/s).
  - By comparison, current devices deliver roughly 15-25 GB/s.
- HMC also requires 70 percent less energy to transfer data
- HMC offers a small form factor — just 10 percent of the footprint of conventional memory.

CPU – HMC Interface

Hybrid Memory Cube (HMC)

- Abstraction Protocol
- Through-Silicon Vias (TSV)
- High-Speed Links

Notes: Tb/s = Terabits / second
HMC height is exaggerated
Memory Wall, Conclusions

[IMPORTANT SLIDE]

- Memory trashing is what kills execution speed

- Many times you will see that when you run your application:
  - You are far away from reaching top speed of the chip
  - You are at top speed for your memory
    - If this is the case, you are trashing the memory
    - Means that basically you are doing one or both of the following
      - Move large amounts of data around
      - Move data often

<table>
<thead>
<tr>
<th>Memory Access Patterns</th>
</tr>
</thead>
<tbody>
<tr>
<td>To/From Registers</td>
</tr>
<tr>
<td>To/From Cache</td>
</tr>
<tr>
<td>To/From RAM</td>
</tr>
<tr>
<td>To/From Disk</td>
</tr>
</tbody>
</table>

![Memory Access Diagram]

- CPU
  - Registers
  - Cache
  - Memory
  - I/O bus
  - I/O devices

<table>
<thead>
<tr>
<th>Type</th>
<th>Size</th>
<th>Speed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register</td>
<td>500 bytes</td>
<td>250 ps</td>
</tr>
<tr>
<td>Cache</td>
<td>64 KB</td>
<td>1 ns</td>
</tr>
<tr>
<td>Memory</td>
<td>1 GB</td>
<td>100 ns</td>
</tr>
<tr>
<td>Disk</td>
<td>1 TB</td>
<td>10 ms</td>
</tr>
</tbody>
</table>

© 2007 Elsevier, Inc. All rights reserved.
Instruction Level Parallelism (ILP)

- ILP: a relevant factor in reducing execution times after 1985

- The basic idea:
  - Overlap execution of independent instructions to improve overall performance
  - During the same clock cycle many instructions are being worked upon

- Two approaches to discovering ILP
  - Dynamic: relies on hardware to discover/exploit parallelism dynamically at run time
    - It is the dominant one in the market
  - Static: relies on compiler to identify parallelism in the code and leverage it (VLIW)

- Examples where ILP expected to improve efficiency

  for( int=0; i<1000; i++)
  
x[i] = x[i] + y[i];

  1. e = a + b
  2. f = c + d
  3. g = e * f
ILP: Various Angles of Attack

- **Instruction pipelining**: the execution of multiple instructions can be partially overlapped; where each instruction is divided into series of sub-steps (termed: micro-operations)

- **Superscalar execution**: multiple execution units are used to execute multiple instructions in parallel

- **Out-of-order execution**: instructions execute in any order but without violating data dependencies

- **Register renaming**: a technique used to avoid data hazards and thus lead to unnecessary serialization of program instructions caused by the reuse of registers

- **Speculative execution**: allows the execution of complete instructions or parts of instructions before being sure whether this execution is required

- **Branch prediction**: used to avoid delays (termed: stalls). Used in combination with speculative execution.
How Microarchitecture Reflected into Execution

- Squeeze the most out of each cycle…
- Vertical axis: a summary of hardware assets
- Horizontal axis: time
The ILP Wall

- ILP, the good:
  - **Existing** programs enjoy performance benefits without any modification
  - Recompiling them is beneficial but entirely up to you as long as you stick with the same ISA (for instance, if you go from Pentium 2 to Pentium 4 you don’t have to recompile your executable)

- ILP, the bad:
  - Improvements are difficult to forecast since the “speculation” success is difficult to predict
  - Moreover, ILP causes a super-linear increase in execution unit complexity (and associated power consumption) without linear speedup.

- ILP, the ugly: serial performance acceleration using ILP plateauing because of these effects
The Power Wall

- Power, and not manufacturing, limits traditional general purpose microarchitecture improvements (F. Pollack, Intel Fellow)

- Leakage power dissipation gets worse as gates get smaller, because gate dielectric thicknesses must proportionately decrease

![Graph showing the Power Wall](image)

Adapted from F. Pollack (MICRO’99)
The Power Wall

- Power dissipation in clocked digital devices is related to the clock frequency and feature length imposing a natural limit on clock rates.

- Significant increase in clock speed without heroic (and expensive) cooling is not possible. Chips would simply melt.

- Clock speed increased by a factor of 4,000 in less than two decades.
  - The ability of manufacturers to dissipate heat is limited though...
  - Look back at the last five years, the clock rates are pretty much flat.

- Problem might be addressed one day by a Materials Science breakthrough.
Trivia

- AMD Phenom II X4 955 (4 core load)
  - 236 Watts

- Intel Core i7 920 (8 thread load)
  - 213 Watts

- Human Brain
  - 20 W
  - Represents 2% of our mass
  - Burns 20% of all energy in the body at rest