Lecture 2: Processor Design, Single-Processor Performance

G63.2011.002/G22.2945.001 · September 14, 2010
Outline

Intro

The Basic Subsystems

Machine Language

The Memory Hierarchy

Pipelines
• Lec. 1 slides posted
• New here? Welcome! Please send in survey info (see lec. 1 slides) via email
• PASI
• Please subscribe to mailing list
• Near end of class: 5-min, 3-question ‘concept check’
Outline

Intro

The Basic Subsystems

Machine Language

The Memory Hierarchy

Pipelines
Goal for Today
High Performance Computing: Discuss the actual computer end of this... and its influence on performance
What’s in a computer?

Processor
Intel Q6600 Core2 Quad, 2.4 GHz

Die
(2 × 143 mm², 2 × 2 cores)
582,000,000 transistors
∼ 100W

Memory
What’s in a computer?

Processor

Intel Q6600 Core2 Quad, 2.4 GHz
What’s in a computer?

**Processor**

Intel Q6600 Core2 Quad, 2.4 GHz

**Die**

(2×) 143 mm², 2 × 2 cores

582,000,000 transistors

∼ 100W
What’s in a computer?

- Processor: Intel Q6600 Core2 Quad, 2.4 GHz
- Die: (2 × 143 mm², 2 × 2 cores)
- Transistors: 582,000,000
- Power: ∼100W

Intro Basics Assembly Memory Pipelines
What’s in a computer?

Processor
- Intel Q6600 Core2 Quad, 2.4 GHz
- Die: \((2 \times 143 \text{ mm})^2\), 2\times2 cores
- 582,000,000 transistors
- \(\sim 100W\)

Memory

Intro Basics Assembly Memory Pipelines
Outline

Intro

The Basic Subsystems

Machine Language

The Memory Hierarchy

Pipelines
A Basic Processor

(loosely based on Intel 8086)
A Basic Processor

Bonus Question: What's a bus?

(loosely based on Intel 8086)
How all of this fits together

Everything synchronizes to the *Clock*.

*Control Unit* ("CU"): The brains of the operation. Everything connects to it.

Bus entries/exits are *gated* and (potentially) *buffered*.

CU controls gates, tells other units about ‘what’ and ‘how’:

- What operation?
- Which register?
- Which addressing mode?
What is... an ALU?

**Arithmetic Logic Unit**
One or two operands A, B
Operation selector (Op):

- (Integer) Addition, Subtraction
- (Logical) And, Or, Not
- (Bitwise) Shifts (equivalent to multiplication by power of two)
- (Integer) Multiplication, Division

**Specialized ALUs:**
- Floating Point Unit (FPU)
- Address ALU

Operates on binary representations of numbers. Negative numbers represented by two’s complement.
**What is... a Register File?**

**Registers** are *On-Chip Memory*

- Directly usable as operands in Machine Language
- Often “general-purpose”
- Sometimes special-purpose: Floating point, Indexing, Accumulator
- Small: x86_64: 16×64 bit GPRs
- Very fast (near-zero latency)

<table>
<thead>
<tr>
<th>%r0</th>
<th>%r1</th>
<th>%r2</th>
<th>%r3</th>
<th>%r4</th>
<th>%r5</th>
<th>%r6</th>
<th>%r7</th>
</tr>
</thead>
</table>

How does computer memory work?

One (reading) memory transaction (simplified):
How does computer memory work?

One (reading) memory transaction (simplified):

- **Processor**
  - D0..15
  - A0..15
  - R/W
  - CLK

- **Memory**
How does computer memory work?

One (reading) memory transaction (simplified):

Processor

Memory

CLK

R/\tilde{W}

A0..15

D0..15

Observation: Access (and addressing) happens in bus-width-size “chunks.”
How does computer memory work?

One (reading) memory transaction (simplified):

- Processor
- Memory
- D0..15
- A0..15
- R/W
- CLK

Observation: Access (and addressing) happens in bus-width-size “chunks.”
How does computer memory work?

One (reading) memory transaction (simplified):
How does computer memory work?

One (reading) memory transaction (simplified):

Processor

Memory

CLK

R/\text{\~W}

A0..15

D0..15

Observation: Access (and addressing) happens in bus-width-size "chunks".
How does computer memory work?

One (reading) memory transaction (simplified):

Observation: Access (and addressing) happens in bus-width-size “chunks”.
What is... a Memory Interface?

**Memory Interface** gets and stores binary words in off-chip memory.

Smallest granularity: Bus width

Tells outside memory
- “where” through *address bus*
- “what” through *data bus*

Computer main memory is “Dynamic RAM” (*DRAM*): Slow, but small and cheap.
Outline

Intro

The Basic Subsystems

Machine Language

The Memory Hierarchy

Pipelines
A Very Simple Program

```c
int a = 5;
int b = 17;
int z = a * b;
```

Things to know:

- **Addressing modes** (Immediate, Register, Base plus Offset)
- **0xHexadecimal**
- “AT&T Form”: (we’ll use this)
  
  \[
  \text{<opcode><size> <source>, <dest>}
  \]
Another Look

Intro Basics Assembly Memory Pipelines

Internal Bus

Address ALU

Register File

Flags

Data ALU

Address ALU

Memory Interface

Address Bus

Data Bus

Insn. fetch

Control Unit

PC

Data ALU
Another Look

Internal Bus

Register File

Flags

Data ALU

Address ALU

PC

Instruction fetch

Control Unit

4: c7 45 f4 05 00 00 00 movl $0x5,-0xc(%rbp)

b: c7 45 f8 11 00 00 00 movl $0x11,-0x8(%rbp)

12: 8b 45 f4 mov -0xc(%rbp),%eax

15: 0f af 45 f8 imul -0x8(%rbp),%eax

19: 89 45 fc mov %eax,-0x4(%rbp)

1c: 8b 45 fc mov -0x4(%rbp),%eax

Data Bus

Memory Interface

Insn. fetch

Intro Basics Assembly Memory Pipelines
A Very Simple Program: Intel Form

- “Intel Form”: (you might see this on the net) 
  `<opcode> <sized dest>, <sized source>`
- Goal: Reading comprehension.
- Don’t understand an opcode? 
  Google “<opcode> intel instruction”.

<table>
<thead>
<tr>
<th></th>
<th>opcode</th>
<th>destination size</th>
<th>source size</th>
<th>destination</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>c7 45 f4 05 00 00 00</td>
<td>mov</td>
<td>DWORD PTR [rbp−0xc], 0x5</td>
<td></td>
</tr>
<tr>
<td>b</td>
<td>c7 45 f8 11 00 00 00</td>
<td>mov</td>
<td>DWORD PTR [rbp−0x8], 0x11</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>8b 45 f4</td>
<td>mov</td>
<td>eax, DWORD PTR [rbp−0xc]</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>0f af 45 f8</td>
<td>imul</td>
<td>eax, DWORD PTR [rbp−0x8]</td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>89 45 fc</td>
<td>mov</td>
<td>DWORD PTR [rbp−0x4], eax</td>
<td></td>
</tr>
<tr>
<td>1c</td>
<td>8b 45 fc</td>
<td>mov</td>
<td>eax, DWORD PTR [rbp−0x4]</td>
<td></td>
</tr>
</tbody>
</table>
# Machine Language Loops

```c
int main()
{
    int y = 0, i;
    for (i = 0; y < 10; ++i)
        y += i;
    return y;
}
```

![Machine Language Code]

### Things to know:

- **Condition Codes (Flags)**: Zero, Sign, Carry, etc.
- **Call Stack**: Stack frame, stack pointer, base pointer
- **ABI**: Calling conventions

Want to make those yourself? Write `myprogram.c`.

```
$ cc -c myprogram.c
$ objdump --disassemble myprogram.o
```

Intro Basics Assembly Memory Pipelines
```c
int main()
{
    int y = 0, i;
    for (i = 0; y < 10; ++i)
        y += i;
    return y;
}
```
We know how a computer works!

All of this can be built in about 4000 transistors.
(e.g. MOS 6502 in Apple II, Commodore 64, Atari 2600)

So what exactly is Intel doing with the other 581,996,000 transistors?

Answer:
We know how a computer works!

All of this can be built in about 4000 transistors. (e.g. MOS 6502 in Apple II, Commodore 64, Atari 2600)

So what exactly is Intel doing with the other 581,996,000 transistors?

Answer: *Make things go faster!*
All of this can be built in about 4000 transistors. (e.g. MOS 6502 in Apple II, Commodore 64, Atari 2600)

So what exactly is Intel doing with the other 581,996,000 transistors?

Answer: *Make things go faster!*

**Goal now:**
Understand sources of slowness, and how they get addressed.

Remember: *High Performance Computing*
The High-Performance Mindset

Writing high-performance Codes

Mindset: What is going to be the limiting factor?

- ALU?
- Memory?
- Communication? (if multi-machine)

Benchmark the assumed limiting factor right away.

Evaluate

- Know your peak throughputs (roughly)
- Are you getting close?
- Are you tracking the right limiting factor?
Outline

Intro

The Basic Subsystems

Machine Language

The Memory Hierarchy

Pipelines
Source of Slowness: Memory

Memory is slow.

Distinguish two different versions of “slow”:
- Bandwidth
- Latency

→ Memory has long latency, but can have large bandwidth.

Size of die vs. distance to memory: big!
Dynamic RAM: long intrinsic latency!
Source of Slowness: Memory

Memory is slow.

Distinguish two different versions of “slow”:
- Bandwidth
- Latency

→ Memory has *long latency*, but can have *large bandwidth*.

Size of die vs. distance to memory: big!
Dynamic RAM: long intrinsic latency!

Idea:

Put a look-up table of recently-used data onto the chip.

→ “Cache”
The Memory Hierarchy

Hierarchy of increasingly bigger, slower memories:

- Registers: 1 kB, 1 cycle
- L1 Cache: 10 kB, 10 cycles
- L2 Cache: 1 MB, 100 cycles
- DRAM: 1 GB, 1000 cycles
- Virtual Memory (hard drive): 1 TB, 1 M cycles

How might data locality factor into this?

What is a working set?
The Memory Hierarchy

Hierarchy of increasingly bigger, slower memories:

- Registers
  - 1 kB, 1 cycle

- L1 Cache
  - 10 kB, 10 cycles

- L2 Cache
  - 1 MB, 100 cycles

- DRAM
  - 1 GB, 1000 cycles

- Virtual Memory (hard drive)

How might *data locality* factor into this?

What is a *working set*?
Cache: Actual Implementation

Demands on cache implementation:

- Fast, small, cheap, low-power
- Fine-grained
- High “hit”-rate (few “misses”)

**Problem:**
Goals at odds with each other: Access matching logic expensive!

**Solution 1:** More data per unit of access matching logic
→ Larger “Cache Lines”

**Solution 2:** Simpler/less access matching logic
→ Less than full “Associativity”

Other choices: Eviction strategy, size
Cache: Associativity

Direct Mapped

Memory |
---|---
0
1
2
3
4
5
6

Cache |
---|---
0
1
2
3

2-way set associative

Memory |
---|---
0
1
2
3
4
5
6

Cache |
---|---
0
1
2
3

Miss rate versus cache size on the Integer portion of SPEC CPU2000 [Cantin, Hill 2003]
### Miss rate versus cache size on the Integer portion of SPEC CPU2000 [Cantin, Hill 2003]

![Graph showing miss rate versus cache size](image)

- **Direct**
- **2-way**
- **4-way**
- **8-way**
- **Full**

<table>
<thead>
<tr>
<th>Cache Size</th>
<th>Miss Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>1K</td>
<td>1e-006</td>
</tr>
<tr>
<td>4K</td>
<td>1e-005</td>
</tr>
<tr>
<td>16K</td>
<td>0.0001</td>
</tr>
<tr>
<td>64K</td>
<td>0.001</td>
</tr>
<tr>
<td>256K</td>
<td>0.01</td>
</tr>
<tr>
<td>1M</td>
<td>Inf</td>
</tr>
</tbody>
</table>

**Legend:**
- **Direct**
- **2-way**
- **4-way**
- **8-way**
- **Full**

**Cache Size:**
- 1K
- 4K
- 16K
- 64K
- 256K
- 1M
- Inf

**Legend Colors:**
- Direct: Red
- 2-way: Green
- 4-way: Cyan
- 8-way: Blue
- Full: Black
Cache Example: Intel Q6600/Core2 Quad

--- L1 data cache ---
fully associative cache = false
threads sharing this cache = 0x0 (0)
processor cores on this die = 0x3 (3)
system coherency line size = 0x3f (63)
ways of associativity = 0x7 (7)
number of sets - 1 (s) = 63

--- L1 instruction ---
fully associative cache = false
threads sharing this cache = 0x0 (0)
processor cores on this die = 0x3 (3)
system coherency line size = 0x3f (63)
ways of associativity = 0x7 (7)
number of sets - 1 (s) = 63

--- L2 unified cache ---
fully associative cache = false
threads sharing this cache = 0x1 (1)
processor cores on this die = 0x3 (3)
system coherency line size = 0x3f (63)
ways of associativity = 0xf (15)
number of sets - 1 (s) = 4095

More than you care to know about your CPU:
http://www.etallen.com/cpuid.html
void go(unsigned count, unsigned stride)
{
    const unsigned arr_size = 64 * 1024 * 1024;
    int *ary = (int *) malloc(sizeof(int) * arr_size);

    for (unsigned it = 0; it < count; ++it)
    {
        for (unsigned i = 0; i < arr_size; i += stride)
            ary[i] *= 17;
    }

    free(ary);
}
void go(unsigned count, unsigned stride) {
    const unsigned arr size = 64 \times 1024 \times 1024;
    int *ary = (int *) malloc(sizeof(int) \times arr size);
    for (unsigned it = 0; it < count; ++it) {
        for (unsigned i = 0; i < arr size; i += stride)
            ary[i] = 17;
    }
    free(ary);
}

<table>
<thead>
<tr>
<th>Stride</th>
<th>Time [s]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.02</td>
<td>0.16</td>
</tr>
<tr>
<td>0.04</td>
<td>0.14</td>
</tr>
<tr>
<td>0.06</td>
<td>0.12</td>
</tr>
<tr>
<td>0.08</td>
<td>0.10</td>
</tr>
<tr>
<td>0.10</td>
<td>0.08</td>
</tr>
<tr>
<td>0.12</td>
<td>0.06</td>
</tr>
<tr>
<td>0.14</td>
<td>0.04</td>
</tr>
<tr>
<td>0.16</td>
<td>0.02</td>
</tr>
</tbody>
</table>
void go(unsigned array_size, unsigned steps)
{
    int *ary = (int *) malloc(sizeof(int) * array_size);
    unsigned asm1 = array_size - 1;

    for (unsigned i = 0; i < steps; ++i)
        ary[(i*16) & asm1] ++;

    free(ary);
}
Measuring the Cache II

```c
void go(unsigned array size, unsigned steps) {
    int *ary = (int *) malloc(sizeof(int) * array size);
    unsigned asm1 = array size - 1;
    for (unsigned i = 0; i < steps; ++i)
        ary[(i * 16) & asm1] ++;
    free(ary);
}
```
void go(unsigned array_size, unsigned stride, unsigned steps) {
    char *ary = (char *) malloc(sizeof(int) * array_size);

    unsigned p = 0;
    for (unsigned i = 0; i < steps; ++i) {
        ary[p] ++;
        p += stride;
        if (p >= array_size) {
            p = 0;
        }
    }

    free(ary);
}
Measuring the Cache III

```c
void go(unsigned array size, unsigned stride, unsigned steps)
{
    char *ary = (char *) malloc(sizeof(int) * array size);
    unsigned p = 0;
    for (unsigned i = 0; i < steps; ++i)
    {
        ary[p] = 0;
        p += stride;
        if (p >= array size)
            p = 0;
    }
    free(ary);
}
```
Programming for the Cache

How can we rearrange programs to be cache-friendly?

Examples:

- Large vectors $x$, $a$, $b$
  Compute

  $$x \leftarrow x + 3a - 5b.$$
Programming for the Cache

How can we rearrange programs to be cache-friendly?

Examples:

- Large vectors $x, a, b$
  Compute
  \[ x \leftarrow x + 3a - 5b. \]

- Matrix-Matrix Multiplication
  → Homework 1, posted.
Outline

Intro

The Basic Subsystems

Machine Language

The Memory Hierarchy

Pipelines
Source of Slowness: Sequential Operation

- **IF**: Instruction fetch
- **ID**: Instruction Decode
- **EX**: Execution
- **MEM**: Memory Read/Write
- **WB**: Result Writeback
Solution: Pipelining

```
IF  ID  EX  MEM  WB
---
i  IF  ID  EX  MEM  WB

IF  ID  EX  MEM  WB
IF  ID  EX  MEM  WB
IF  ID  EX  MEM  WB
```

Pipelines
Pipelining (MIPS, 110,000 transistors)
Issues with Pipelines

Pipelines generally help performance—but not always.

Possible issues:

- Stalls
- Dependent Instructions
- Branches (+Prediction)
- Self-Modifying Code

“Solution”: Bubbling, extra circuitry
Intel Q6600 Pipeline

Instruction Fetch
32 Byte Pre-Decoded, Fetch Buffer
18 Entry Instruction Queue
7+ Entry μop Buffer
Register Alias Table and Allocator
96 Entry Reorder Buffer (ROB)

32 Entry Reservation Station
Port 0
Port 1
Port 2
Port 3
Port 4
Port 5
ALU
SSE Shuffle ALU
SSE Shuffle MUL
ALU Branch
SSE ALU
Store Address
Store Data
Load Address
Memory Ordering Buffer (MOB)

Memory Interface

New concept: Instruction-level parallelism ("Superscalar")
Intro Basics Assembly Memory Pipelines
New concept: Instruction-level parallelism ("Superscalar")
How to upset a processor pipeline:

```c
for (int i = 0; i < 1000; ++i)
    for (int j = 0; j < 1000; ++j)
    {
        if (j % 2 == 0)
            do_something(i, j);
    }
```

... why is this bad?
int steps = 256 * 1024 * 1024;
int [] a = new int[2];

// Loop 1
for (int i=0; i<steps; i++) { a[0]++; a[0]++; }

// Loop 2
for (int i=0; i<steps; i++) { a[0]++; a[1]++; }

Which is faster?

... and why?
Two useful Strategies

**Loop unrolling:**

```c
for (int i = 0; i < 1000; ++i)
  do_something(i);
```

→

```c
for (int i = 0; i < 500; i+=2)
{
  do_something(i);
  do_something(i+1);
}
```

**Software pipelining:**

```c
for (int i = 0; i < 1000; ++i)
{
  do_a(i);
  do_b(i);
}
```

→

```c
for (int i = 0; i < 500; i+=2)
{
  do_a(i);
  do_a(i+1);
  do_b(i);
  do_b(i+1);
}
```
Control Units are large and expensive.

Functional Units are simple and cheap.

→ Increase the Function/Control ratio:
Control several functional units with one control unit.

All execute same operation.

GCC vector extensions:

```c
typedef int v4si __attribute__((vector_size (16)));

v4si a, b, c;
c = a + b;
// +, −, *, /, unary minus, ^, |, &, ~, %
```

Will revisit for OpenCL, GPUs.
About HW1

• Open-ended! Want: coherent thought about caches, memory access ordering, latencies, bandwidths, etc. See Berkeley CS267 (lec. 3, ...) for more hints.
• Also: Introduction to the machinery. (Clusters, running on them, SSH, git, forge) Why so much machinery?
• Linux lab machines: WWH 229/230
Rearranging Matrix-Matrix Multiplication

Matrix Multiplication:

\[ C_{ij} = \sum_k A_{ik} B_{kj} \]
Rearranging Matrix-Matrix Multiplication

Matrix Multiplication:

\[ C_{ij} = \sum_k A_{ik} B_{kj} \]
Matrix Multiplication:

\[ C_{ij} = \sum_k A_{ik} B_{kj} \]
Matrix Multiplication:

\[ C_{ij} = \sum_k A_{ik} B_{kj} \]
Matrix Multiplication:

\[ C_{ij} = \sum_k A_{ik} B_{kj} \]
Rearranging Matrix-Matrix Multiplication

Matrix Multiplication:

$$C_{ij} = \sum_k A_{ik} B_{kj}$$
Questions?
Image Credits

- Q6600 Wikimedia Commons
- Mainboard: Wikimedia Commons
- DIMM: sxc.hu/gobran11
- Q6600 back: Wikimedia Commons
- Core 2 die: Intel Corp. / lesliewong.us
- Cache associativity: based on Wikipedia
- Q6600 Wikimedia Commons
- Cache Measurements: Igor Ostrovsky
- Pipeline stuff: Wikipedia