Parallel and Future Architecture

Hung-Wei
Recap: Pipeline SuperScalar/OoO/ROB

Front-end:
- Instruction Fetch
- Instruction Decode
- Register renaming logic
- Issue/Schedule

Back-end:
- ALU
- MUL/DIV 1
- FP1
- Address Resolution
- FP2
- MUL/DIV 2
- Address Queue
- MEM
- ROB

Branch predictor
Recap: Register renaming

- Provide a set of “physical registers” and a mapping table mapping “architectural registers” to “physical registers”
- Allocate a physical register for a new output
- Eliminate all false dependencies
- Stages
  - Dispatch (D) — allocate a “physical” for the output of a decoded instruction
  - Issue (I) — collect pending values/branch outcome from common data bus
  - Execute (INT, AQ/AQ/MEM, M1/M2/M3, BR) — send the instruction to its corresponding pipeline if no structural hazards
  - Write Back (WB) — broadcast the result through CDB
Recap: Speculative Execution

• Any execution of an instruction before any prior instruction finishes is considered as **speculative execution**

• Because it’s speculative, we need to preserve the capability to restore to the states before it’s executed
  • Branch mis-prediction
  • Exceptions
Recap: Reorder buffer/Commit stage

• Reorder buffer — a buffer keep track of the program order of instructions
  • Can be combined with IQ or physical registers — make either as a circular queue
• Commit stage — should the outcome of an instruction be realized
  • An instruction can only leave the pipeline if all it’s previous are committed
  • If any prior instruction failed to commit, the instruction should yield it’s ROB entry, restore all it’s architectural changes
Recap: Pipeline SuperScalar/OoO/ROB

Front-end:
- Instruction Fetch
- Instruction Decode
- Register renaming logic
- Issue/Schedule
- Address Resolution
- Branch predictor

Back-end:
- FP1
- FP2
- ALU
- MUL/DIV 1
- MUL/DIV 2
- Address Queue
- MEM
- ROB
How good is SS/OoO/ROB with this code?

- Consider the following dynamic instructions:
  - `lw $1, 0($10)`
  - `addi $10, $10, 8`
  - `add $20, $20, $1`
  - `bne $10, $2, LOOP`
  - `lw $1, 0($10)`
  - `addi $10, $10, 8`
  - `add $20, $20, $1`
  - `bne $10, $2, LOOP`

Assume a superscalar processor with issue width as 2 & unlimited physical registers that can fetch up to 4 instructions per cycle, 2 cycles for an integer instruction, 3 cycles to execute a memory instruction. How many cycles it takes to issue all instructions?

A. 2
B. 4
C. 6
D. 8
E. 10
Recap: Intel Sandy Bridge
ZEN MICROARCHITECTURE

- Fetch Four x86 instructions
- Op Cache instructions
- 4 Integer units
  - Large rename space – 168 Registers
  - 192 instructions in flight/8 wide retire
- 2 Load/Store units
  - 72 Out-of-Order Loads supported
- 2 Floating Point units x 128 FMACs
  - built as 4 pipes, 2 Fadd, 2 Fmul
- I-Cache 64K, 4-way
- D-Cache 32K, 8-way
- L2 Cache 512K, 8-way
- Large shared L3 cache
- 2 threads per core
Recap: What about “linked list”

Static instructions

- **LOOP:**
  - `ld  X10, 8(X10)`
  - `addi X7, X7, 1`
  - `bne X10, X0, LOOP`

Dynamic instructions

- `ld  X10, 8(X10)`
- `addi X7, X7, 1`
- `bne X10, X0, LOOP`
- `ld  X10, 8(X10)`
- `addi X7, X7, 1`
- `bne X10, X0, LOOP`
- `ld  X10, 8(X10)`
- `addi X7, X7, 1`
- `bne X10, X0, LOOP`
- `ld  X10, 8(X10)`
- `addi X7, X7, 1`
- `bne X10, X0, LOOP`

ILP is low because of data dependencies
2-way, 2-issue, Simultaneous multithreading

① ld X10, 8(X10)
② addi X7, X7, 1
③ bne X10, X0, LOOP
④ ld X10, 8(X10)
⑤ addi X7, X7, 1
⑥ bne X10, X0, LOOP
⑦ ld X10, 8(X10)
⑧ addi X7, X7, 1
⑨ bne X10, X0, LOOP
⑩ ld X1, 0(X10)
⑪ addi X10, X10, 8
⑫ add X20, X20, X1
⑬ bne X10, X2, LOOP
⑭ ld X1, 0(X10)
⑮ addi X10, X10, 8
⑯ add X20, X20, X1
⑰ bne X10, X2, LOOP
⑱ ld X1, 0(X10)
⑲ addi X10, X10, 8
⑳ add X20, X20, X1
㉑ bne X10, X2, LOOP
Recap: SMT

- Improve the throughput of execution
  - May increase the latency of a single thread
- Less branch penalty per thread
- Increase hardware utilization
- Simple hardware design: Only need to duplicate PC/Register Files
- Real Case:
  - Intel HyperThreading (supports up to two threads per core)
    - Intel Pentium 4, Intel Atom, Intel Core i7
  - AMD Ryzen
Dynamic/Active Power

• The power consumption due to the switching of transistor states

• Dynamic power per transistor

\[ P_{\text{dynamic}} \sim \alpha \times C \times V^2 \times f \times N \]

• \( \alpha \): average switches per cycle
• \( C \): capacitance
• \( V \): voltage
• \( f \): frequency, usually linear with \( V \)
• \( N \): the number of transistors
Static/Leakage Power

- The power consumption due to leakage — transistors do not turn all the way off during no operation
- Becomes the **dominant** factor in the most advanced process technologies.

\[ P_{\text{leakage}} \sim N \times V \times e^{-V_t} \]

- \(N\): number of transistors
- \(V\): voltage
- \(V_t\): threshold voltage where transistor conducts (begins to switch)

*Figure 1: Leakage power becomes a growing problem as demands for more performance and functionality drive chipmakers to nanometer-scale process nodes (Source: IBS).*
The “power/energy cost” of doubling the clocking rate

\[
\begin{align*}
\text{Power}_{\text{new}} &= \text{Power}_{\text{old}} \times \left( \frac{f_{\text{new}}}{f_{\text{old}}} \right)^3 \\
\text{Power}_{\text{new}} &= \text{Power}_{\text{old}} \times (2)^3 = \text{Power}_{\text{old}} \times 8 \\
\text{Speedup} &= \frac{\text{Execution Time}_{\text{baseline}}}{\text{Execution Time}_{\text{enhanced}}} = \frac{5}{4} = 1.25 \\
\text{Energy}_{\text{new}} &= \text{Power}_{\text{new}} \times ET_{\text{new}} \\
&= \text{Power}_{\text{new}} \times \frac{ET_{\text{old}}}{\text{Speedup}} \\
&= \text{Power}_{\text{old}} \times 8 \times \frac{ET_{\text{old}}}{1.25} = 6.4 \times \text{Power}_{\text{old}} \times ET_{\text{old}} \\
&= 6.4 \times \text{Energy}_{\text{old}}
\end{align*}
\]
What if we double the number of cores?

$$\text{Power}_{\text{new}} = \text{Power}_{\text{old}} \times \text{number\_of\_cores} = \text{Power}_{\text{old}} \times 2$$

Assume 40% of execution time can be parallelized

$$\text{Speedup}_{\text{parallel}}(f_{\text{parallelizable}}, n) = \frac{1}{(1 - f_{\text{parallelizable}}) + \frac{f_{\text{parallelizable}}}{n}}$$

$$= \frac{1}{(1 - 40\%) + \frac{40\%}{2}} = 1.25$$

$$\text{Energy}_{\text{new}} = \text{Power}_{\text{new}} \times \frac{\text{ET}_{\text{new}}}{\text{ET}_{\text{old}}}$$

$$= \text{Power}_{\text{new}} \times \frac{\text{ET}_{\text{old}}}{\text{Speedup}}$$

$$= \text{Power}_{\text{old}} \times 2 \times \frac{\text{ET}_{\text{old}}}{1.25} = 1.6 \times \text{Power}_{\text{old}} \times \text{ET}_{\text{old}}$$

A better deal in terms of energy!

Same performance gain!
More cores per chip, slower per core

<table>
<thead>
<tr>
<th>Status</th>
<th>Intel® Xeon® Processor E7-8890 v4</th>
<th>Intel® Xeon® Processor E7-8880 v4</th>
<th>Intel® Xeon® Processor E7-8890 v4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Launch Date</td>
<td>Launched</td>
<td>Launched</td>
<td>Launched</td>
</tr>
<tr>
<td>Lithography</td>
<td>14 nm</td>
<td>14 nm</td>
<td>14 nm</td>
</tr>
</tbody>
</table>

Performance

| # of Cores | 24 | 4 | 22 |
| # of Threads | 48 | 8 | 44 |
| Processor Base Frequency | 2.20 GHz | 3.20 GHz | 2.20 GHz |
| Max Turbo Frequency | 3.40 GHz | 3.50 GHz | 3.30 GHz |
| Cache | 50 MB | 60 MB | 55 MB |
| Bus Speed | 9.6 GT/s | 5.6 GT/s | 9.6 GT/s |
| # of QPI Links | 3 | 3 | 3 |
| TDP | 165 W | 140 W | 150 W |
Concept of CMP

Processor

Core
Registers
L1-$
L2-$

Core
Registers
L1-$
L2-$

Core
Registers
L1-$
L2-$

Core
Registers
L1-$
L2-$

Last-level $ (LLC)
Both CMP & SMT exploit thread-level or task-level parallelism. Assuming both application X and application Y have similar instruction combination, say 60% ALU, 20% load/store, and 20% branches. Consider two processors:

P1: CMP with a 2-issue pipeline on each core. Each core has a private L1 32KB D-cache

P2: SMT with a 4-issue pipeline. 64KB L1 D-cache

Which one do you think is better?
A. P1
B. P2
SMT v.s. CMP

- Both CMP & SMT exploit thread-level or task-level parallelism. Assuming both application X and application Y have similar instruction combination, say 60% ALU, 20% load/store, and 20% branches. Consider two processors:

  P1: CMP with a 2-issue pipeline on each core. Each core has a private L1 32KB D-cache

  P2: SMT with a 4-issue pipeline. 64KB L1 D-cache

Which one do you think is better?
  A. P1
  B. P2
SMT v.s. CMP

- Both CMP & SMT exploit thread-level or task-level parallelism. Assuming both application X and application Y have similar instruction combination, say 60% ALU, 20% load/store, and 20% branches. Consider two processors:

  P1: CMP with a 2-issue pipeline on each core. Each core has a private L1 32KB D-cache

  P2: SMT with a 4-issue pipeline. 64KB L1 D-cache

Which one do you think is better?

A. P1
B. P2
Outline

• Parallel programming
• Dark Silicon and Future Computer System Design
Architectural Support for Parallel Programming
Parallel programming

• To exploit parallelism you need to break your computation into multiple “processes” or multiple “threads”

• Processes (in OS/software systems)
  • Separate programs actually running (not sitting idle) on your computer at the same time.
  • Each process will have its own virtual memory space and you need explicitly exchange data using inter-process communication APIs

• Threads (in OS/software systems)
  • Independent portions of your program that can run in parallel
  • All threads share the same virtual memory space

• We will refer to these collectively as “threads”
  • A typical user system might have 1-8 actively running threads.
  • Servers can have more if needed (the sysadmins will hopefully configure it that way)
What software thinks about “multiprogramming” hardware

[Diagram showing a shared virtual address space and multiple cores with associated registers and cache levels (L1-$ and L2-$).]
What software thinks about “multiprogramming” hardware

```
for(i=0;i<size/4;i++)
  sum += a[i];
sum = 0
for(i=size/4;i<size/2;i++)
  sum += a[i];
for(i=size/2;i<3*size/4;i++)
  sum += a[i];
for(i=3*size/4;i<size;i++)
  sum += a[i];
sum = 0

Others do not see the updated value in the cache and keep working — incorrect result!
```
Coherency & Consistency

• Coherency — Guarantees all processors see the same value for a variable/memory address in the system when the processors need the value at the same time
  • What value should be seen
• Consistency — All threads see the change of data in the same order
  • When the memory operation should be done
Simple cache coherency protocol

- Snooping protocol
  - Each processor broadcasts / listens to cache misses
- State associate with each block (cacheline)
  - Invalid
    - The data in the current block is invalid
  - Shared
    - The processor can read the data
    - The data may also exist on other processors
  - Exclusive
    - The processor has full permission on the data
    - The processor is the only one that has up-to-date data
Coherent way-associative cache

memory address: 0x0
memory address: 0b0000100000100100

States

<table>
<thead>
<tr>
<th>D</th>
<th>tag</th>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td>01 1</td>
<td>0x29</td>
<td>IIJJKKLLMMNNOOPP</td>
</tr>
<tr>
<td>01 1</td>
<td>0xDE</td>
<td>QQRRSSTTUUVVVwXX</td>
</tr>
<tr>
<td>01 0</td>
<td>0x10</td>
<td>YYYZAABBCCDDEEFF</td>
</tr>
<tr>
<td>00 1</td>
<td>0x8A</td>
<td>AABBCCDDEEGGFFHH</td>
</tr>
<tr>
<td>10 1</td>
<td>0x60</td>
<td>IIJJKKLLMMNNOOPP</td>
</tr>
<tr>
<td>10 1</td>
<td>0x70</td>
<td>QQRRSSTTUUVVVwXX</td>
</tr>
<tr>
<td>10 1</td>
<td>0x10</td>
<td>QQRRSSTTUUVVVwXX</td>
</tr>
<tr>
<td>10 1</td>
<td>0x11</td>
<td>YYYZAABBCCDDEEFF</td>
</tr>
</tbody>
</table>

States

<table>
<thead>
<tr>
<th>D</th>
<th>tag</th>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td>01 1</td>
<td>0x00</td>
<td>AABBCCDDEEGGFFHH</td>
</tr>
<tr>
<td>01 1</td>
<td>0x10</td>
<td>IIJJKKLLMMNNOOPP</td>
</tr>
<tr>
<td>01 0</td>
<td>0xA1</td>
<td>QQRRSSTTUUVVVwXX</td>
</tr>
<tr>
<td>00 1</td>
<td>0x10</td>
<td>YYYZAABBCCDDEEFF</td>
</tr>
<tr>
<td>10 1</td>
<td>0x31</td>
<td>AABBCCDDEEGGFFHH</td>
</tr>
<tr>
<td>10 1</td>
<td>0x45</td>
<td>IIJJKKLLMMNNOOPP</td>
</tr>
<tr>
<td>10 1</td>
<td>0x41</td>
<td>QQRRSSTTUUVVVwXX</td>
</tr>
<tr>
<td>10 1</td>
<td>0x68</td>
<td>YYYZAABBCCDDEEFF</td>
</tr>
</tbody>
</table>

hit??
Snooping Protocol

- **Invalid**
  - read/write miss (bus)
  - write miss (processor)

- **Shared**
  - read miss (processor)
  - write miss (bus)

- **Exclusive**
  - write hit
  - write miss (bus)
  - write request (processor)
  - write back data

- **Transition**
  - read miss/hit
  - write request (processor)
  - write back data

- **Actions**
  - read miss (bus)
  - write hit
What happens when we write in coherent caches?

```
for (i=0; i<size/4; i++)
    sum += a[i];
sum = 0
for (i=size/4; i<size/2; i++)
    sum += a[i];
for (i=size/2; i<3*size/4; i++)
    sum += a[i];
for (i=3*size/4; i<size; i++)
    sum += a[i];
```

**Thread**

- Registers
- L1-$
- L2-$
- Shared Virtual Address Space

**Shared Virtual Address Space**

- sum = 0xDEADBEEF
- sum = 0
- sum = 0
- sum = 0
- sum = 0

write miss/invalidate

write back

read miss
Assuming that we are running the following code on a CMP with a cache coherency protocol, how many of the following outputs are possible? (a is initialized to 0 as assume we will output more than 10 numbers)

<table>
<thead>
<tr>
<th></th>
<th>thread 1</th>
<th>thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>①</td>
<td>0 1 2 3 4 5 6 7 8 9</td>
<td>while(1)</td>
</tr>
<tr>
<td>②</td>
<td>1 2 5 9 3 6 8 10 12 13</td>
<td>a++;</td>
</tr>
<tr>
<td>③</td>
<td>1 1 1 1 1 1 1 64 100</td>
<td></td>
</tr>
<tr>
<td>④</td>
<td>1 1 1 1 1 1 1 1 1 100</td>
<td></td>
</tr>
</tbody>
</table>

A. 0  
B. 1  
C. 2  
D. 3  
E. 4
Cache coherency

• Assuming that we are running the following code on a CMP with a cache coherency protocol, how many of the following outputs are possible? (a is initialized to 0 as assume we will output more than 10 numbers)

<table>
<thead>
<tr>
<th>Output</th>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>① 0 1 2 3 4 5 6 7 8 9</td>
<td>while (1) printf(&quot;%d &quot;, a);</td>
<td>while (1) a++;</td>
</tr>
<tr>
<td>② 1 2 5 9 3 6 8 10 12 13</td>
<td></td>
<td></td>
</tr>
<tr>
<td>③ 1 1 1 1 1 1 64 100</td>
<td></td>
<td></td>
</tr>
<tr>
<td>④ 1 1 1 1 1 1 1 1 1 1 100</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

A. 0
B. 1
C. 2
D. 3
E. 4
• Assuming that we are running the following code on a CMP with a cache coherency protocol, how many of the following outputs are possible? (a is initialized to 0 as assume we will output more than 10 numbers)

<table>
<thead>
<tr>
<th>thread 1</th>
<th>thread 2</th>
</tr>
</thead>
</table>
| `while(1)`
  `printf("%d ",a);`                        | `while(1)`
  `a++;`                                      |

① 0 1 2 3 4 5 6 7 8 9
② 1 2 5 9 3 6 8 10 12 13
③ 1 1 1 1 1 1 1 64 100
④ 1 1 1 1 1 1 1 1 1 100

A. 0
B. 1
C. 2
D. 3
E. 4
False sharing

Shared Virtual Address Space

Thread

Thread

Thread

Thread

A[0] = 0
A[1] = 0
A[2] = 0
A[3] = 0

A[0] = 0
A[1] = 0
A[2] = 0
A[3] = 0

A[0] = 0
A[1] = 0
A[2] = 0
A[3] = 0

A[0] = 0
A[1] = 0
A[2] = 0
A[3] = 0

write back

write miss/
 invalidate

A[0] = 0xDEADBEEF
A[1] = 0
A[2] = 0
A[3] = 0

A[0] = 0
A[1] = 0
A[2] = 0
A[3] = 0

A[0] = 0
A[1] = 0
A[2] = 0
A[3] = 0

A[0] = 0
A[1] = 0
A[2] = 0
A[3] = 0

read miss

Shared Memory

Core

Core

Core

Core

Registers

Registers

Registers

Registers

L1-

L1-

L1-

L1-

$ 

$ 

$ 

$ 

write miss/
Comparing implementations of thread_vadd — L and R, please identify which one will be performing better and why

A. L is better, because the cache miss rate is lower
B. R is better, because the cache miss rate is lower
C. L is better, because the instruction count is lower
D. R is better, because the instruction count is lower
E. Both are about the same
Comparing implementations of thread_vadd — L and R, please identify which one will be performing better and why

A. L is better, because the cache miss rate is lower
B. R is better, because the cache miss rate is lower
C. L is better, because the instruction count is lower
D. R is better, because the instruction count is lower
E. Both are about the same

Main thread

Version L

```
void *threaded_vadd(void *thread_id)
{
    int tid = *(int *)thread_id;
    int i;
    for(i=tid;i<ARRAY_SIZE;i+=NUM_OF_THREADS)
    {
        c[i] = a[i] + b[i];
    }
    return NULL;
}
```

Version R

```
void *threaded_vadd(void *thread_id)
{
    int tid = *(int *)thread_id;
    int i;
    for(i=tid*(ARRAY_SIZE/NUM_OF_THREADS);i<(tid+1)*(ARRAY_SIZE/NUM_OF_THREADS);i++)
    {
        c[i] = a[i] + b[i];
    }
    return NULL;
}
```
L v.s. R

Version L

```c
void *threaded_vadd(void *thread_id)
{
    int tid = *(int *)thread_id;
    int i;
    for(i=tid;i<ARRAY_SIZE;i+=NUM_OF_THREADS)
    {
        c[i] = a[i] + b[i];
    }
    return NULL;
}
```

Version R

```c
void *threaded_vadd(void *thread_id)
{
    int tid = *(int *)thread_id;
    int i;
    for(i=tid*(ARRAY_SIZE/NUM_OF_THREADS);i<(tid+1)*(ARRAY_SIZE/NUM_OF_THREADS);i++)
    {
        c[i] = a[i] + b[i];
    }
    return NULL;
}
```
4Cs of cache misses

• 3Cs:
  • Compulsory, Conflict, Capacity
• Coherency miss:
  • A “block” invalidated because of the sharing among processors.
False sharing

• True sharing
  • Processor A modifies X, processor B also want to access X.

• False sharing
  • Processor A modifies X, processor B also want to access Y. However, Y is invalidated because X and Y are in the same block!
Comparing implementations of thread_vadd — L and R, please identify which one will be performing better and why

A. L is better, because the cache miss rate is lower
B. R is better, because the cache miss rate is lower
C. L is better, because the instruction count is lower
D. R is better, because the instruction count is lower
E. Both are about the same
```c
int loop;

int main()
{
    pthread_t thread;
    loop = 1;

    pthread_create(&thread, NULL, modifyloop, NULL);
    while(loop == 1)
    {
        continue;
    }
    pthread_join(thread, NULL);
    fprintf(stderr,"User input: %d\n", loop);
    return 0;
}

void* modifyloop(void *x)
{
    sleep(1);
    printf("Please input a number:\n");
    scanf("%d",&loop);
    return NULL;
}
```
volatile int loop;

int main()
{
    pthread_t thread;
    loop = 1;

    pthread_create(&thread, NULL, modifyloop, NULL);
    while(loop == 1)
    {
        continue;
    }
    pthread_join(thread, NULL);
    fprintf(stderr,"User input: %d\n", loop);
    return 0;
}

void* modifyloop(void *x)
{
    sleep(1);
    printf("Please input a number:\n");
    scanf("%d", &loop);
    return NULL;
}
Consider the given program. You can safely assume the caches are coherent. How many of the following outputs will you see?

1. (0, 0)
2. (0, 1)
3. (1, 0)
4. (1, 1)

A. 0
B. 1
C. 2
D. 3
E. 4

---

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>

volatile int a,b;
volatile int x,y;
volatile int f;

void* modifya(void *z) {
    a=1;
    x=b;
    return NULL;
}

void* modifyb(void *z) {
    b=1;
    y=a;
    return NULL;
}

int main() {
    int i;
    pthread_t thread[2];
    pthread_create(&thread[0], NULL, modifya, NULL);
    pthread_create(&thread[1], NULL, modifyb, NULL);
    pthread_join(thread[0], NULL);
    pthread_join(thread[1], NULL);
    fprintf(stderr, "(%d, %d)\n",x,y);
    return 0;
}
```
Again — how many values are possible?

- Consider the given program. You can safely assume the caches are coherent. How many of the following outputs will you see?

1. (0, 0)
2. (0, 1)
3. (1, 0)
4. (1, 1)

A. 0  
B. 1  
C. 2  
D. 3  
E. 4

```c
#include <stdio.h>  
#include <stdlib.h>  
#include <pthread.h>  
#include <unistd.h>  

volatile int a, b;  
volatile int x, y;  
volatile int f;  

void* modifya(void *z) {  
    a = 1;  
    x = b;  
    return NULL;  
}

void* modifyb(void *z) {  
    b = 1;  
    y = a;  
    return NULL;  
}

int main() {  
    int i;  
    pthread_t thread[2];  
    pthread_create(&thread[0], NULL, modifya, NULL);  
    pthread_create(&thread[1], NULL, modifyb, NULL);  
    pthread_join(thread[0], NULL);  
    pthread_join(thread[1], NULL);  
    fprintf(stderr, "(%d, %d)\n", x, y);  
    return 0;  
}
```
Possible scenarios

**Thread 1**
- a=1;
- x=b;

**Thread 2**
- b=1;
- y=a;

(1,1)

**Thread 1**
- a=1;
- x=b;

**Thread 2**
- b=1;
- y=a;

(1,0)

**Thread 1**
- a=1;
- x=b;

**Thread 2**
- b=1;
- y=a;

(0,1)

**Thread 1**
- x=b;
- a=1;

**Thread 2**
- y=a;
- b=1;

(0,0)

OoO Scheduling!
Why (0,0)?

• Processor/compiler may reorder your memory operations/instructions
  • Coherence protocol can only guarantee the update of the same memory address
  • Processor can serve memory requests without cache miss first
  • Compiler may store values in registers and perform memory operations later
• Each processor core may not run at the same speed (cache misses, branch mis-prediction, I/O, voltage scaling and etc..)
• Threads may not be executed/scheduled right after it’s spawned
Again — how many values are possible?

- Consider the given program. You can safely assume the caches are coherent. How many of the following outputs will you see?

  1. (0, 0)
  2. (0, 1)
  3. (1, 0)
  4. (1, 1)

A. 0  
B. 1  
C. 2  
D. 3  
E. 4
fence instructions

- x86 provides an “mfence” instruction to prevent reordering across the fence instruction.
- x86 only supports this kind of “relaxed consistency” model. You still have to be careful enough to make sure that your code behaves as you expected.

<table>
<thead>
<tr>
<th>thread 1</th>
<th>thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>a=1; mfence</td>
<td>b=1; mfence</td>
</tr>
<tr>
<td>a=1 must occur/update before mfence</td>
<td>b=1 must occur/update before mfence</td>
</tr>
<tr>
<td>x=b;</td>
<td>y=a;</td>
</tr>
</tbody>
</table>
Take-away of parallel programming

- Processor behaviors are non-deterministic
  - You cannot predict which processor is going faster
  - You cannot predict when OS is going to schedule your thread
- Cache coherency only guarantees that everyone would eventually have a coherent view of data, but not when
- Cache consistency is hard to support
Dark Silicon
The power consumption due to leakage — transistors do not turn all the way off during no operation.

Becomes the dominant factor in the most advanced process technologies.

\[ P_{\text{leakage}} \sim N \times V \times e^{-V_t} \]

- \( N \): number of transistors
- \( V \): voltage
- \( V_t \): threshold voltage where transistor conducts (begins to switch)

Figure 1: Leakage power becomes a growing problem as demands for more performance and functionality drive chipmakers to nanometer-scale process nodes (Source: IBS).
Dennardian Scaling

- Given a scaling factor $S$

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Relation</th>
<th>Classical Scaling</th>
</tr>
</thead>
<tbody>
<tr>
<td>Power Budget</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>Chip Size</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>Vdd (Supply Voltage)</td>
<td>1/$S$</td>
<td>1/$S$</td>
</tr>
<tr>
<td>Vt (Threshold Voltage)</td>
<td>1/$S$</td>
<td>1/$S$</td>
</tr>
<tr>
<td>tex (oxide thickness)</td>
<td>1/$S$</td>
<td>1/$S$</td>
</tr>
<tr>
<td>$W, L$ (transistor dimensions)</td>
<td>1/$S$</td>
<td>1/$S$</td>
</tr>
<tr>
<td>Cgate (gate capacitance)</td>
<td>WL/tox</td>
<td>1/$S$</td>
</tr>
<tr>
<td>Isat (saturation current)</td>
<td>WVdd/tox</td>
<td>1/$S$</td>
</tr>
<tr>
<td>$F$ (device frequency)</td>
<td>Isat/(CgateVdd)</td>
<td>$S$</td>
</tr>
<tr>
<td>$D$ (Device/Area)</td>
<td>1/(WL)</td>
<td>$S^2$</td>
</tr>
<tr>
<td>$p$ (device power)</td>
<td>IsatVdd</td>
<td>1/$S^2$</td>
</tr>
<tr>
<td>$P$ (chip power)</td>
<td>Dp</td>
<td>1</td>
</tr>
<tr>
<td>$U$ (utilization)</td>
<td>1/P</td>
<td>1</td>
</tr>
</tbody>
</table>
If we are able to cram more transistors within the same chip area (Moore’s law continues), but the power consumption per transistor remains the same. Right now, if put more transistors in the same area because the technology allows us to. How many of the following statements are true?

① The power consumption per chip will increase
② The power density of the chip will increase
③ Given the same power budget, we may not able to power on all chip area if we maintain the same clock rate
④ Given the same power budget, we may have to lower the clock rate of circuits to power on all chip area

A. 0
B. 1
C. 2
D. 3
E. 4
If we are able to cram more transistors within the same chip area (Moore’s law continues), but the power consumption per transistor remains the same. Right now, if put more transistors in the same area because the technology allows us to. How many of the following statements are true?

① The power consumption per chip will increase
② The power density of the chip will increase
③ Given the same power budget, we may not able to power on all chip area if we maintain the same clock rate
④ Given the same power budget, we may have to lower the clock rate of circuits to power on all chip area

A. 0
B. 1
C. 2
D. 3
E. 4

What happens if power doesn’t scale with process technologies?
## Dennardian Broken

- Given a scaling factor $S$

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Relation</th>
<th>Classical Scaling</th>
<th>Leakage Limited</th>
</tr>
</thead>
<tbody>
<tr>
<td>Power Budget</td>
<td></td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Chip Size</td>
<td></td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>$V_{dd}$ (Supply Voltage)</td>
<td>$1/S$</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>$V_{t}$ (Threshold Voltage)</td>
<td>$1/S$</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>$t_{ex}$ (oxide thickness)</td>
<td>$1/S$</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>$W, L$ (transistor dimensions)</td>
<td>$1/S$</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>$C_{gate}$ (gate capacitance)</td>
<td>$WL/tox$</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>$I_{sat}$ (saturation current)</td>
<td>$WV_{dd}/tox$</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>$F$ (device frequency)</td>
<td>$I_{sat}/(C_{gate}V_{dd})$</td>
<td>$S$</td>
<td>$S$</td>
</tr>
<tr>
<td>$D$ (Device/Area)</td>
<td>$1/(WL)$</td>
<td>$S^2$</td>
<td>$S^2$</td>
</tr>
<tr>
<td>$p$ (device power)</td>
<td>$I_{sat}V_{dd}$</td>
<td>$1/S^2$</td>
<td>1</td>
</tr>
<tr>
<td>$P$ (chip power)</td>
<td>$D_p$</td>
<td>1</td>
<td>$S^2$</td>
</tr>
<tr>
<td>$U$ (utilization)</td>
<td>$1/P$</td>
<td>1</td>
<td>$1/S^2$</td>
</tr>
</tbody>
</table>
Power consumption to light on all transistors

<table>
<thead>
<tr>
<th>Chip</th>
<th>Dennardian Scaling</th>
<th>Dennardian Broken</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 1 1 1 1 1 1</td>
<td>0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5</td>
<td>1 1 1 1 1 1 1 1 1</td>
</tr>
<tr>
<td>1 1 1 1 1 1 1 1 1</td>
<td>0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5</td>
<td>1 1 1 1 1 1 1 1 1</td>
</tr>
<tr>
<td>1 1 1 1 1 1 1 1 1</td>
<td>0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5</td>
<td>1 1 1 1 1 1 1 1 1</td>
</tr>
<tr>
<td>1 1 1 1 1 1 1 1 1</td>
<td>0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5</td>
<td>1 1 1 1 1 1 1 1 1</td>
</tr>
<tr>
<td>1 1 1 1 1 1 1 1 1</td>
<td>0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5</td>
<td>1 1 1 1 1 1 1 1 1</td>
</tr>
<tr>
<td>1 1 1 1 1 1 1 1 1</td>
<td>0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5</td>
<td>1 1 1 1 1 1 1 1 1</td>
</tr>
<tr>
<td>1 1 1 1 1 1 1 1 1</td>
<td>0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5</td>
<td>1 1 1 1 1 1 1 1 1</td>
</tr>
<tr>
<td>1 1 1 1 1 1 1 1 1</td>
<td>0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5</td>
<td>1 1 1 1 1 1 1 1 1</td>
</tr>
<tr>
<td>1 1 1 1 1 1 1 1 1</td>
<td>0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5</td>
<td>1 1 1 1 1 1 1 1 1</td>
</tr>
</tbody>
</table>

=49W  
=50W  
=100W!
What happens if power doesn’t scale with process technologies?

• If we are able to cram more transistors within the same chip area (Moore’s law continues), but the power consumption per transistor remains the same. Right now, if we power the chip with the same power consumption but put more transistors in the same area because the technology allows us to. How many of the following statements are true?

① The power consumption per chip will increase
② The power density of the chip will increase
③ Given the same power budget, we may not able to power on all chip area if we maintain the same clock rate
④ Given the same power budget, we may have to lower the clock rate of circuits to power on all chip area

A. 0
B. 1
C. 2
D. 3
E. 4
Dynamic/Active Power

• The power consumption due to the switching of transistor states

• Dynamic power per transistor

\[ P_{\text{dynamic}} \sim \alpha \times C \times V^2 \times f \times N \]

  • \( \alpha \): average switches per cycle
  • \( C \): capacitance
  • \( V \): voltage
  • \( f \): frequency, usually linear with \( V \)
  • \( N \): the number of transistors
• The power consumption due to leakage — transistors do not turn all the way off during no operation
• Becomes the dominant factor in the most advanced process technologies.

\[ P_{\text{leakage}} \sim N \times V \times e^{-V_t} \]

How about static power?

Figure 1: Leakage power becomes a growing problem as demands for more performance and functionality drive chipmakers to nanometer-scale process nodes (Source: IBS).
Disable circuits if not-in-use
NVIDIA’s Turing Architecture
Use tensor cores

cublasErrCheck(cublasSetMathMode(cublasHandle, CUBLAS_TENSOR_OP_MATH));

Make them 16-bit

cvtColorFp32ToFp16 << (MATRIX_M * MATRIX_K + 255) / 256, 256 >>> (a_fp16, a_fp32, MATRIX_M * MATRIX_K);
    convertFp32ToFp16 <<< (MATRIX_K * MATRIX_N + 255) / 256, 256 >>> (b_fp16, b_fp32, MATRIX_K * MATRIX_N);


call Gemm
You can only use either type of these ALUs, but not all of them.
The rise of ASICs
Say, we want to implement $a[i] += a[i+1]*20$

- This is what we need in RISC-V in each iteration

```
ld   X1, 0(X0)  IF ID EX MEM WB
ld   X2, 8(X0)  IF ID EX MEM WB
add  X3, X31, #20 IF ID EX MEM WB
mul  X2, X2, X3 IF ID EX MEM WB
add  X1, X1, X2 IF ID EX MEM WB
sd   X1, 0(X0)  IF ID EX MEM WB
```
This is what you need for these instructions
Specialize the circuit

We don’t need instruction fetch given it’s a fixed function
Specialize the circuit

We don’t need these many registers, complex control, decode

We don’t need instruction fetch given it’s a fixed function
Specialize the circuit

We don’t need ALUs, branches, hazard detections...

We don’t need these many registers, complex control, decode

We don’t need instruction fetch given it’s a fixed function
Specialize the circuit

We don’t need big ALUs, branches, hazard detections...

We don’t need these many registers, complex control, decode

We don’t need instruction fetch given it’s a fixed function
Rearranging the datapath

```
ld   X1, 0(X0)
ld   X2, 8(X0)
add  X3, X31, #20
mul  X2, X2, X3
add  X1, X1, X2
sd   X1, 0(X0)
```
The pipeline for $a[i] += a[i+1]*20$

Each stage can still be as fast as the pipelined processor.

But each stage is now working on what the original 6 instructions would do.
What TPU looks like
TPU Floorplan

Local Unified Buffer for Activations
(96Kx256x8b = 24 MiB)
29% of chip

Matrix Multiply Unit
(256x256x8b=64K MAC)
24%

Host Interf. 2%

Accumulators (4Kx256x32b =4 MiB) 6%

Control 2%

Activation Pipeline 6%

PCle Interface 3%

Misc. I/O 1%

75
TPU Block diagram
Final words
Conclusion

- Computer architecture is more important than you can ever imagine
- Being a “programmer“ is easy. You need to know architecture a lot to be a “performance programmer”
  - Branch prediction
  - Cache
- Multicore era — to get your multithreaded program correct and perform well, you need to take care of coherence and consistency
- We’re now in the “dark silicon era”
  - Single-core isn’t getting any faster
  - Multi-core doesn’t scale anymore
  - We will see more and more ASICs
  - You need to write more “system-level” programs to use these new ASICs.
Announcements

• CAPE/Survey
  • Screenshot of your CAPE
  • Fill the survey
  • Count as a full-credit assignment and we’re dropping your lowest two assignments now.

• Assignment 5 is up — a mini final
  • Given that we’re dropping 2 lowest assignment grade and give you a full credit one once you submitted your post-CAPE survey, you probably don’t need to turn in that.
  • Strongly encourage to practice that since it covers the material for the last week of class

• Regarding final exam —
  • 9/4 8am—6pm — any consecutive, non-stop 3-hour slot you pick
  • Open books, open notes, but it’s going to be twice longer than the midterm
  • Not using Lockdown browser — since some of you having troubles with that
  • No zoom, no response to piazza posts for fairness