CS 33

Architecture and Optimization (2)
Modern CPU Design

Instruction Control

Fetch Control

Instruction Cache

Instruction Decode

Fetch

Address

Instructions

Operations

Prediction OK?

Register Updates

Retirement Unit

Register File

Functional Units

Operation Results

Integer/ Branch

General Integer

FP Add

FP Mult/Div

Load

Store

Data Cache

Addr.

Data

Predic？

OK?

Integer

Add

Mult/Div

Data

Addr.

Addr.

Data

Data

Addr.
Superscalar Processor

• **Definition:** A superscalar processor can issue and execute *multiple instructions in one cycle*
  – instructions are retrieved from a sequential instruction stream and are usually scheduled dynamically
    » instructions may be executed *out of order*

• **Benefit:** without programming effort, superscalar processors can take advantage of the *instruction-level parallelism* that most programs have

• Most CPUs since about 1998 are superscalar
• Intel: since Pentium Pro (1995)
Multiple Operations per Instruction

- `addl %eax, %edx`
  - a single operation
- `addl %eax, 4(%edx)`
  - three operations
    » load value from memory
    » add to it the contents of %eax
    » store result in memory
Instruction-Level Parallelism

• `addl 4(%eax), %eax`  
  `addl %ebx, %edx`  
  – can be executed simultaneously: completely independent

• `addl 4(%eax), %ebx`  
  `addl %ebx, %edx`  
  – can also be executed simultaneously, but some coordination is required
Out-of-Order Execution

- `movss (%rbp), %xmm0`
- `mulss (%rax, %rdx, 4), %xmm0`
- `movvss %xmm0, (%rbp)`
- `addq %r8d, %r9d`
- `imulq %rcx, %r12d`
- `addq $1, %rdx`

these can be executed without waiting for the first three to finish
Speculative Execution

80489f3:    movl $0x1,%ecx
80489f8:    xorl %edx,%edx
80489fa:    cmpl %esi,%edx
80489fc:    jnl 8048a25
80489fe:    movl %esi,%esi
8048a00:    imull (%eax,%edx,4),%ecx

perhaps execute these instructions
Nehalem CPU

- Multiple instructions can execute in parallel
  1 load, with address computation
  1 store, with address computation
  2 simple integer (one may be branch)
  1 complex integer (multiply/divide)
  1 FP Multiply
  1 FP Add

- Some instructions take > 1 cycle, but can be pipelined

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Latency</th>
<th>Cycles/Issue</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load / Store</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>Integer Add</td>
<td>1</td>
<td>.33</td>
</tr>
<tr>
<td>Integer Multiply</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>Integer/Long Divide</td>
<td>11–21</td>
<td>11–21</td>
</tr>
<tr>
<td>Single/Double FP Multiply</td>
<td>4/5</td>
<td>1</td>
</tr>
<tr>
<td>Single/Double FP Add</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>Single/Double FP Divide</td>
<td>10–23</td>
<td>10–23</td>
</tr>
</tbody>
</table>
x86-64 Compilation of Combine4

- Inner loop (case: integer multiply)

```
.L519:
imull (%rax,%rdx,4), %ecx       # t = t * d[i]
addq $1, %rdx                   # i++
cmpq %rdx, %rbp                # Compare length:i
jg .L519                       # If >, goto Loop
```

<table>
<thead>
<tr>
<th>Method</th>
<th>Integer</th>
<th>Double FP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operation</td>
<td>Add</td>
<td>Mult</td>
</tr>
<tr>
<td>Combine4</td>
<td>2.0</td>
<td>3.0</td>
</tr>
<tr>
<td>Latency bound</td>
<td>1.0</td>
<td>3.0</td>
</tr>
<tr>
<td>Throughput bound</td>
<td>1.0</td>
<td>1.0</td>
</tr>
</tbody>
</table>
Inner Loop

\[
\begin{align*}
%rax & \rightarrow \text{load} \\
%rbp & \rightarrow \text{mul} \\
%rdx & \rightarrow \text{add} \\
%rbp & \rightarrow \text{cmp} \\
%rax & \rightarrow \text{jg} \\
%rbp & \rightarrow \text{mulss} (\%rax,\%rdx,4), \%xmm0 \\
%rdx & \rightarrow \text{addq} \ 1,\%rdx \\
%rbp & \rightarrow \text{cmpq} \ 1,\%rbp \\
%rax & \rightarrow \text{jg} \ \text{loop}
\end{align*}
\]
Data-Flow Graphs of Inner Loop

---

**Diagram:**

```
<table>
<thead>
<tr>
<th>%xmm0</th>
<th>%rax</th>
<th>%rbp</th>
<th>%rdx</th>
</tr>
</thead>
<tbody>
<tr>
<td>load</td>
<td>mul</td>
<td>cmp</td>
<td>jg</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>%xmm0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>%rdx</td>
</tr>
</tbody>
</table>

Data[\textit{i}] 

```

```
Relative Execution Times

%xmm0

%rdx

data[i]

load

mul

%xmm0

%rdx

add
Data Flow Over Multiple Iterations

Critical path
Pipelined Data-Flow Over Multiple Iterations
Pipelined Data-Flow Over Multiple Iterations

Diagram:

- **load** → **mul** → **load** → **mul** → **load** → **add** → **add** → **add**
Pipelined Data-Flow Over Multiple Iterations
Combine4 = Serial Computation (OP = *)

- Computation (length=8)
  \[ ((((((1 \times d[0]) \times d[1]) \times d[2]) \times d[3]) \times d[4]) \times d[5]) \times d[6]) \times d[7]) \]

- Sequential dependence
  - performance: determined by latency of OP
Loop Unrolling

- Perform 2x more useful work per iteration
Loop Unrolling

- Perform 2x more useful work per iteration

```c
void unroll2x(vec_ptr_t v, data_t *dest)
{
    int length = vec_length(v);
    int limit = length-1;
    data_t *d = get_vec_start(v);
    data_t x = IDENT;
    int i;
    /* Combine 2 elements at a time */
    for (i = 0; i < limit; i+=2) {
        x = (x OP d[i]) OP d[i+1];
    }
    /* Finish any remaining elements */
    for (; i < length; i++) {
        x = x OP d[i];
    }
    *dest = x;
}
```

Quiz 1

Does it speed things up?

a) yes  
b) no
### Effect of Loop Unrolling

<table>
<thead>
<tr>
<th>Method</th>
<th>Integer</th>
<th>Double FP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operation</td>
<td>Add</td>
<td>Mult</td>
</tr>
<tr>
<td>Combine4</td>
<td>2.0</td>
<td>3.0</td>
</tr>
<tr>
<td>Unroll 2x</td>
<td>2.0</td>
<td>1.5</td>
</tr>
<tr>
<td>Latency bound</td>
<td>1.0</td>
<td>3.0</td>
</tr>
<tr>
<td>Throughput bound</td>
<td>1.0</td>
<td>1.0</td>
</tr>
</tbody>
</table>

- Helps integer multiply
  - below latency bound
  - compiler does clever optimization

- Others don’t improve. *Why?*
  - still sequential dependency

\[ x = (x \text{ OP } d[i]) \text{ OP } d[i+1]; \]
Loop Unrolling with Reassociation

```c
void unroll2xra(vec_ptr_t v, data_t *dest)
{
    int length = vec_length(v);
    int limit = length-1;
    data_t *d = get_vec_start(v);
    data_t x = IDENT;
    int i;
    /* Combine 2 elements at a time */
    for (i = 0; i < limit; i+=2) {
        x = x OP (d[i] OP d[i+1]);
    }
    /* Finish any remaining elements */
    for (; i < length; i++) {
        x = x OP d[i];
    }
    *dest = x;
}
```

- Can this change the result of the computation?
- Yes, for FP. **Why?**
Effect of Reassociation

<table>
<thead>
<tr>
<th>Method</th>
<th>Integer</th>
<th>Double FP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operation</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Add</td>
<td>2.0</td>
<td>3.0</td>
</tr>
<tr>
<td>Mult</td>
<td>3.0</td>
<td>5.0</td>
</tr>
<tr>
<td>Combine4</td>
<td>2.0</td>
<td>3.0</td>
</tr>
<tr>
<td>Unroll 2x</td>
<td>2.0</td>
<td>3.0</td>
</tr>
<tr>
<td>Unroll 2x, reassociate</td>
<td>2.0</td>
<td>3.0</td>
</tr>
<tr>
<td>Latency bound</td>
<td>1.0</td>
<td>3.0</td>
</tr>
<tr>
<td>Throughput bound</td>
<td>1.0</td>
<td>1.0</td>
</tr>
</tbody>
</table>

- Nearly 2x speedup for int*, FP +, FP *
  - reason: breaks sequential dependency

\[ x = x \text{ OP} \left( d[i] \text{ OP} d[i+1] \right); \]

- why is that? (next slide)
Reassociated Computation

\[ x = x \text{ OP} (d[i] \text{ OP} d[i+1]); \]

- **What changed:**
  - ops in the next iteration can be started early (no dependency)

- **Overall Performance**
  - N elements, D cycles latency/op
  - should be \((N/2+1)*D\) cycles:
    \[ \text{CPE} = D/2 \]
  - measured CPE slightly worse for FP mult
Loop Unrolling with Separate Accumulators

```c
void unroll2xp2x(vec_ptr_t v, data_t *dest)
{
    int length = vec_length(v);
    int limit = length-1;
    data_t *d = get_vec_start(v);
    data_t x0 = IDENT;
    data_t x1 = IDENT;
    int i;
    /* Combine 2 elements at a time */
    for (i = 0; i < limit; i+=2) {
        x0 = x0 OP d[i];
        x1 = x1 OP d[i+1];
    }
    /* Finish any remaining elements */
    for (; i < length; i++) {
        x0 = x0 OP d[i];
    }
    *dest = x0 OP x1;
}
```

- **Different form of reassociation**
### Effect of Separate Accumulators

<table>
<thead>
<tr>
<th>Method</th>
<th>Integer</th>
<th>Double FP</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Operation</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Combine4</td>
<td>2.0</td>
<td>3.0</td>
</tr>
<tr>
<td>Unroll 2x</td>
<td>2.0</td>
<td>1.5</td>
</tr>
<tr>
<td>Unroll 2x, reassociate</td>
<td>2.0</td>
<td>1.5</td>
</tr>
<tr>
<td>Unroll 2x parallel 2x</td>
<td>1.5</td>
<td>1.5</td>
</tr>
<tr>
<td>Latency bound</td>
<td>1.0</td>
<td>3.0</td>
</tr>
<tr>
<td>Throughput bound</td>
<td>1.0</td>
<td>1.0</td>
</tr>
</tbody>
</table>

- 2x speedup (over unroll2x) for int *, FP +, FP *
  - breaks sequential dependency in a “cleaner,” more obvious way

```
x0 = x0 OP d[i];
x1 = x1 OP d[i+1];
```
Separate Accumulators

\[
x_0 = x_0 \text{ OP } d[i]; \\
x_1 = x_1 \text{ OP } d[i+1];
\]

- **What changed:**
  - two independent “streams” of operations

- **Overall Performance**
  - N elements, D cycles latency/op
  - should be \((N/2+1) \times D\) cycles:
    \[
    \text{CPE} = \frac{D}{2}
    \]
  - CPE matches prediction!

**What Now?**
Quiz 2

With 3 accumulators there will be 3 independent streams of instructions; with 4 accumulators 4 independent streams of instructions, etc.

Thus with n accumulators we can have a speedup of O(n), as long as n is no greater than the number of available registers.

a) true

b) false
Unrolling & Accumulating

• Idea
  – can unroll to any degree L
  – can accumulate K results in parallel
  – L must be multiple of K

• Limitations
  – diminishing returns
    » cannot go beyond throughput limitations of execution units
  – large overhead for short lengths
    » finish off iterations sequentially
Performance

- K-way loop unrolling with K accumulators
Achievable Performance

<table>
<thead>
<tr>
<th>Method</th>
<th>Integer</th>
<th>Double FP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operation</td>
<td>Add</td>
<td>Mult</td>
</tr>
<tr>
<td>Scalar optimum</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>Latency bound</td>
<td>1.00</td>
<td>3.00</td>
</tr>
<tr>
<td>Throughput bound</td>
<td>1.00</td>
<td>1.00</td>
</tr>
</tbody>
</table>

- Limited only by throughput of functional units
- Up to 29X improvement over original, unoptimized code
Using Vector Instructions

<table>
<thead>
<tr>
<th>Method</th>
<th>Integer</th>
<th></th>
<th>Double FP</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Operation</td>
<td>Add</td>
<td>Mult</td>
<td>Add</td>
<td>Mult</td>
</tr>
<tr>
<td>Scalar optimum</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>Vector optimum</td>
<td>0.25</td>
<td>0.53</td>
<td>0.53</td>
<td>0.57</td>
</tr>
<tr>
<td>Latency bound</td>
<td>1.00</td>
<td>3.00</td>
<td>3.00</td>
<td>5.00</td>
</tr>
<tr>
<td>Throughput bound</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>Vec throughput bound</td>
<td>0.25</td>
<td>0.50</td>
<td>0.50</td>
<td>0.50</td>
</tr>
</tbody>
</table>

- Make use of SSE Instructions
  - parallel operations on multiple data elements
What About Branches?

• Challenge
  – *instruction control unit* must work well ahead of *execution unit* to generate enough operations to keep EU busy

```assembly
80489f3:  movl  $0x1,%ecx
80489f8:  xorl  %edx,%edx
80489fa:  cmpl  %esi,%edx
80489fc:  jnl   8048a25
80489fe:  movl  %esi,%esi
8048a00:  imull (%eax,%edx,4),%ecx
```

– when it encounters conditional branch, cannot reliably determine where to continue fetching

**Executing**

**How to continue?**
Modern CPU Design

Instruction Control

Instruction Cache

Fetch Control

Instruction Decode

Retirement Unit

Register File

Register Updates

Prediction OK?

Operation Results

Operation

Predict? OK?

Addr.

Data

Instruction

Cache

Data

Addr.

Integer/Branch

General Integer

FP Add

FP Mult/Div

Load

Store

Functional Units

Data Cache

Addr.

Data
Branch Outcomes

• When encounter conditional branch, cannot determine where to continue fetching
  – branch taken: transfer control to branch target
  – branch not-taken: continue with next instruction in sequence
• Cannot resolve until outcome determined by branch/integer unit

```
80489f3: movl  $0x1,%ecx
80489f8: xorl  %edx,%edx
80489fa: cmpl  %esi,%edx
80489fc: jnl   8048a25
80489fe: movl  %esi,%esi
8048a00: imull (%eax,%edx,4),%ecx

8048a25: cmpl  %edi,%edx
8048a27: jl    8048a20
8048a29: movl  0xc(%ebp),%eax
8048a2c: leal  0xfffffffffffffff8(%ebp),%esp
8048a2f: movl  %ecx,(%eax)
```
Branch Prediction

• Idea
  – guess which way branch will go
  – begin executing instructions at predicted position
    » but don’t actually modify register or memory data

80489f3: movl $0x1, %ecx
80489f8: xorl %edx, %edx
80489fa: cmpl %esi, %edx
80489fc: jnl 8048a25
...

8048a25: cmpl %edi, %edx
8048a27: jl 8048a20
8048a29: movl 0xc(%ebp), %eax
8048a2c: leal 0xfffffffffe8(%ebp), %esp
8048a2f: movl %ecx, (%eax)
**Branch Prediction Through Loop**

```
80488b1:  movl   (%ecx,%edx,4),%eax  
80488b4:  addl   %eax,(%edi)        
80488b6:  incl   %edx              
80488b7:  cmpl   %esi,%edx         
80488b9:  jl     80488b1

Assume
vector length = 100

Predict taken (OK)

80488b1:  movl   (%ecx,%edx,4),%eax  
80488b4:  addl   %eax,(%edi)        
80488b6:  incl   %edx              
80488b7:  cmpl   %esi,%edx         
80488b9:  jl     80488b1

Predict taken (oops)

80488b1:  movl   (%ecx,%edx,4),%eax  
80488b4:  addl   %eax,(%edi)        
80488b6:  incl   %edx              
80488b7:  cmpl   %esi,%edx         
80488b9:  jl     80488b1

Read invalid location

80488b1:  movl   (%ecx,%edx,4),%eax  
80488b4:  addl   %eax,(%edi)        
80488b6:  incl   %edx              
80488b7:  cmpl   %esi,%edx         
80488b9:  jl     80488b1

Executed

80488b1:  movl   (%ecx,%edx,4),%eax  
80488b4:  addl   %eax,(%edi)        
80488b6:  incl   %edx              
80488b7:  cmpl   %esi,%edx         
80488b9:  jl     80488b1

Fetched
```

Assume vector length = 100

Predict taken (OK)

Read invalid location

```
i = 98
i = 99
i = 100
i = 101
```
### Branch Misprediction Invalidation

Assume vector length = 100

| 80488b1: | movl (%ecx,%edx,4),%eax   |
| 80488b4: | addl %eax,(%edi)          |
| 80488b6: | incl %edx                 |
| 80488b7: | cmpl %esi,%edx            |
| 80488b9: | jl 80488b1                |
|           | Predict taken (OK)        |

\[ i = 98 \]

| 80488b1: | movl (%ecx,%edx,4),%eax   |
| 80488b4: | addl %eax,(%edi)          |
| 80488b6: | incl %edx                 |
| 80488b7: | cmpl %esi,%edx            |
| 80488b9: | jl 80488b1                |
|           | Predict taken (oops)       |

\[ i = 99 \]

| 80488b1: | movl (%ecx,%edx,4),%eax   |
| 80488b4: | addl %eax,(%edi)          |
| 80488b6: | incl %edx                 |
| 80488b7: | cmpl %esi,%edx            |
| 80488b9: | jl 80488b1                |
|           | Invalidate                 |

\[ i = 100 \]

| 80488b1: | movl (%ecx,%edx,4),%eax   |
| 80488b4: | addl %eax,(%edi)          |
| 80488b6: | incl %edx                 |
| 80488b7: | cmpl %esi,%edx            |
| 80488b9: | jl 80488b1                |
|           | Predict taken (OK)        |

\[ i = 101 \]
Branch Misprediction Recovery

- Performance Cost
  - multiple clock cycles on modern processor
  - can be a major performance limiter
Conditional Moves

```c
void minmax1(int *a, int *b, int n {
    int i;
    for (i=0; i<n; i++) {
        if (a[i] > b[i]) {
            int t = a[i];
            a[i] = b[i];
            b[i] = t;
        }
    }
}
```

```c
void minmax2(int *a, int *b, int n {
    int i;
    for (i=0; i<n; i++) {
        int min = a[i] < b[i]?
            a[i] : b[i];
        int max = a[i] < b[i]?
            b[i] : a[i];
        a[i] = min;
        b[i] = max;
    }
}
```

- Compiled code uses conditional branch
  - 14.5 CPE for random data
  - 2.0 – 4.0 CPE for predictable data
- Compiled code uses conditional move instruction
  - 5.0 CPE regardless of data’s pattern
Latency of Loads

typedef struct ELE {
    struct ELE *next;
    int data;
} list_ele, *list_ptr;

int list_len(list_ptr ls) {
    int len = 0;
    while (ls) {
        len++;
        ls = ls->next;
    }
    return len;
}
#define ITERS 100000000

int main() {
    volatile int dest[100];
    int iter;
    for (iter=0; iter<ITERS; iter++) {
        long i;
        for (i=0; i<100; i++)
            dest[i] = 0;
    }
}

Clearing an Array ...

#define ITERS 100000000
int main() {
    volatile int dest[100];
    int iter;
    for (iter=0; iter<ITERS; iter++) {
        long i;
        for (i=0; i<97; i+=4) {
            dest[i] = 0;
            dest[i+1] = 0;
            dest[i+2] = 0;
            dest[i+3] = 0;
        }
    }
}
Store/Load Interaction

```c
void write_read(int *src, int *dest, int n) {
    int cnt = n;
    int val = 0;

    while(cnt--) {
        *dest = val;
        val = (*src)+1;
    }
}
```
Store/Load Interaction

```c
int a[] = {-10, 17};
```

Example A: `write_read(&a[0],&a[1],3)`

<table>
<thead>
<tr>
<th>Iter.</th>
<th>cnt</th>
<th>a</th>
<th>val</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initial</td>
<td>3</td>
<td>-10</td>
<td>0</td>
</tr>
<tr>
<td>Iter. 1</td>
<td>2</td>
<td>-10</td>
<td>-9</td>
</tr>
<tr>
<td>Iter. 2</td>
<td>1</td>
<td>-10</td>
<td>-9</td>
</tr>
<tr>
<td>Iter. 3</td>
<td>0</td>
<td>-10</td>
<td>-9</td>
</tr>
</tbody>
</table>

Example B: `write_read(&a[0],&a[0],3)`

<table>
<thead>
<tr>
<th>Iter.</th>
<th>cnt</th>
<th>a</th>
<th>val</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initial</td>
<td>3</td>
<td>-10</td>
<td>0</td>
</tr>
<tr>
<td>Iter. 1</td>
<td>2</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Iter. 2</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>Iter. 3</td>
<td>0</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

- CPE 2.0
- CPE 6.0
Some Details of Load and Store

Load unit

Store unit

Data cache

Matching addresses

Address

Data

Address

Data

Address

Data

Address

Data
Inner-Loop Data Flow of Write_Read

Inner-Loop Data Flow of Write_Read

movl %eax, (%ecx)  *dest = val;
movl (%ebx), %eax  val = *src
addl $1, %eax  val++;
subl $1, %edx  cnt--;
jne loop
Inner-Loop Data Flow of Write_Read

1. %eax → s_addr
2. s_addr → %ecx
3. %ecx → load

load → add
add → %eax

%eax → sub
sub → %edx

%edx → jne
jne → %eax

%eax → %edx
%edx → s_data
s_data → load
load → add
add → %eax

%eax → %edx
Data Flow
Getting High Performance

• Good compiler and flags
• Don’t do anything stupid
  – watch out for hidden algorithmic inefficiencies
  – write compiler-friendly code
    » watch out for optimization blockers: procedure calls & memory references
  – look carefully at innermost loops (where most work is done)

• Tune code for machine
  – exploit instruction-level parallelism
  – avoid unpredictable branches
  – make code cache friendly (covered soon)
Multiple Cores

Chip

Instruction Control

Retirement Unit
Register File

Fetch Control

Instruction Decode

Instruction Cache

Address

Instructions

Operations

Functional Units

Integer/Branch
General Integer
FP Add
FP Mult/Div
Load
Store

Instruction Control

Retirement Unit
Register File

Fetch Control

Instruction Decode

Instruction Cache

Address

Instructions

Operations

Functional Units

Integer/Branch
General Integer
FP Add
FP Mult/Div
Load
Store

Execution

Operation Results

Data Cache

Other Stuff

More Cache

Other Stuff
Hyper Threading

**Execution**

Instruction Control
- Retirement Unit
- Fetch Control
- Instruction Cache
- Address
- Instructions

Instruction Control
- Retirement Unit
- Fetch Control
- Instruction Cache
- Address
- Instructions

Functional Units
- Integer/Branch
- General Integer
- FP Add
- FP Mult/Div
- Load
- Store

Operation Results
- Addr.
- Data

Data Cache