VLIW Architectures

Introduction

Spring 2011
VLIW Introduction

- Why VLIW?
- Very Long Instruction Words
- Methods to Increase Parallelism: Loop Unrolling, Software Pipelining, Scheduling Regions
- Problems of VLIW
- Specific Architecture Features

lecture notes based on various sources (Krste Asanovic, UC Berkeley; “Limits of Instruction-Level Parallelism” by David Wall)
VLIW Introduction

- Little’s Law

Parallelism = Throughput * Latency

\[ \bar{N} = \bar{T} \times \bar{L} \]
VLIW Introduction

- Example: Pipelined ILP Machine

max throughput: six instructions per cycle

- latency in cycles
  - two ALUs 1 cycle latency
  - two load/store units 1 cycle latency
  - two FP units 4 cycles latency

how much instruction-level parallelism (ILP) is required to keep the machine pipelines busy?

\[
\bar{T} = 6 \quad \bar{L} = \frac{(2 \times 1 + 2 \times 3 + 2 \times 4)}{6} = 2 \frac{2}{3} \quad \bar{N} = 6 \times 2 \frac{2}{3} = 16
\]
VLIW Introduction

- **Superscalar Control Logic Scaling**

  Each issued instruction must be checked against $W \times L$ instructions, i.e., the growth in hardware $\propto W \times (W \times L)$

  - For in-order machines, $L$ is related to pipeline latencies
  - For out-of-order machines, $L$ also includes time spent in instruction buffers (instruction window or ROB)

  As $W$ increases, a larger instruction window is needed to find enough parallelism to keep the machine busy => greater $L$

  $\rightarrow$ *out-of-order control logic grows faster than $W^2$ (\textasciitilde W^3)*
VLIW Introduction

- Bottleneck of Sequential ISA

- Superscalar compiler
  - Find independent operations
  - Schedule operations

- Superscalar processor
  - Check instruction dependencies
  - Schedule execution

(sequential) source code

```c
a = foo(b);
for (i=0; i<N; i++)
...
```
VLIW Introduction

- Out-of-Order Control Complexity: MIPS R10000

[ SGI/MIPS Technologies Inc., 1995 ]
VLIW Introduction

- **VLIW: Very Long Instruction Word (J. Fisher)**

![Diagram showing VLIW architecture]

- multiple operations packed into one instruction
- each operation slot is for a fixed function
- constant operation latencies are specified
- architecture requires guarantee of:
  - parallelism within an instruction => no x-operation RAW check
  - no data use before data ready => no data interlocks
    
    → this is the job of the compiler
VLIW Introduction

- **VLIW Compiler Goals and Responsibilities**
  - schedule for maximum performance (→ maximize parallel execution)
  - schedule operation types in specific slots (NOP)
  - guarantee that data is available when it is used
  - guarantee that no data hazards (interlocks) occur
VLIW Introduction

- **Early VLIW Machines**
  - FPS AP120B (1976)
    - scientific attached array processor
    - first commercial wide instruction machine
    - hand-coded vector math libraries using software pipelining and loop unrolling
  - Multiflow Trace (1987)
    - commercialization of ideas from Fisher’s Yale group including “trace scheduling”
    - available in configurations with 7, 14, or 28 operations/instruction
    - 28 operations packed into a 1024-bit instruction word
  - Cydrome Cydra-5 (1987)
    - 7 operations encoded in 256-bit instruction word
    - rotating register file
VLIW Introduction

- Loop Execution

```c
for (i=0; i<N; i++)
```

Compile:
```
loop:  ld f1, 0(r1)
       add r1, 8
       fadd f2, f0, f1
       st f2, 0(r2)
       add r2, 8
       bne r1, r3, loop
```

How many FP ops/cycle?

1 fadd / 8 cycles = 0.125
VLIW Introduction

- Loop Execution: Unrolling

```c
for (i=0; i<N; i++)
```

unroll 4 times

```c
for (i=0; i<N; i+=4)
{
}
```

(requires extra code after the loop body to handle values of N that are not a multiple of the unroll factor)
### VLIW Introduction

- **Loop Execution: Unrolling**

```plaintext
loop:  ld f1, 0(r1)  
      ld f2, 8(r1)  
      ld f3, 16(r1)  
      ld f4, 24(r1)  
      add r1, 32  
      fadd f5, f0, f1  
      fadd f6, f0, f2  
      fadd f7, f0, f3  
      fadd f8, f0, f4  
      st f5, 0(r2)  
      st f6, 8(r2)  
      st f7, 16(r2)  
      st f8, 24(r2)  
      add r2, 32  
      bne r1, r3, loop
```

<table>
<thead>
<tr>
<th></th>
<th>Int1</th>
<th>Int2</th>
<th>M1</th>
<th>M2</th>
<th>FP+</th>
<th>FPx</th>
</tr>
</thead>
<tbody>
<tr>
<td>loop:</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>ld f1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>ld f2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>ld f3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>ld f4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>add r1, 32</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>fadd f5, f0, f1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>fadd f6, f0, f2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>fadd f7, f0, f3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>fadd f8, f0, f4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>st f5, 0(r2)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>st f6, 8(r2)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>st f7, 16(r2)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>st f8, 24(r2)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>add r2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>bne r1, r3, loop</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**How many FP ops/cycle?**

4 fadd / 11 cycles = 0.36
VLIW Introduction

- Loop Execution: Software Pipelining

How many FP ops/cycle?

4 fadd / 4 cycles = 1
Software pipelining pays startup/wind-down costs only once per loop, not once per iteration.
VLIW Introduction

- Techniques for Sequential Code

- Branches limit basic block size in control-flow intensive irregular code
- Difficult to find ILP in individual basic blocks
Trace Scheduling (Fisher, Ellis)

- Pick string of basic blocks, a *trace*, that represents most frequent branch path
- Use *profiling feedback* or compiler heuristics to find common branch paths
- Schedule whole “trace” at once
- Add fixup code to cope with branches jumping out of trace
VLIW Introduction

- If-Conversion, Hyperblocks, Superblocks, Region Scheduling
  - all techniques aiming at increasing the scheduling unit
  - hardware support required for some
  - complex compensation code
  - more details in the next classes
Introduction to VLIW

- Compiler plays a Key Role
  - without a decent compiler, a VLIW processor cannot achieve good performance

(B.Rau & J.Fisher)
VLIW Introduction

- **Problems with “Classic” VLIW**
  - Object-code compatibility
    - have to recompile all code for every machine, even for two machines in same generation
  - Object code size
    - instruction padding wastes instruction memory/cache
    - loop unrolling/software pipelining replicates code
  - Scheduling variable latency memory operations
    - caches and/or memory bank conflicts impose statically unpredictable variability
  - Knowing branch probabilities
    - Profiling requires an significant extra step in build process
  - Scheduling for statically unpredictable branches
    - optimal schedule varies with branch path
VLIW Introduction

- Object Code Compression

- Schemes to reduce effect of unused fields
  - Compressed format in memory, expand on I-cache refill
    » used in Multiflow Trace
    » introduces instruction addressing challenge
  - Mark parallel groups
    » used in TMS320C6x DSPs, Intel IA-64
  - Provide a single-op VLIW instruction
    » Cydra-5 UniOp instructions
VLIW Introduction

- **Architecture Support: Rotating Register Files**
  - Problem: scheduled loops require lots of registers, lots of duplicated code in prolog, epilog

```
ld r1, ()
add r2, r1, #1
st r2, ()
ld r1, ()
add r2, r1, #1
st r2, ()
ld r1, ()
add r2, r1, #1
st r2, ()
ld r1, ()
add r2, r1, #1
st r2, ()
```

- **Solution:** Allocate new set of registers for each loop iteration
VLIW Introduction

• **Architecture Support: Rotating Register Files**

  • Rotating Register Base (RRB) register points to base of current register set. Value added on to logical register specifier to give physical register number. Usually, split into rotating and non-rotating registers.

  ```
P0
P1
P2
P3
P4
P5
P6
P7

RRB=3

R1 +

P0
P1
P2
P3
P4
P5
P6
P7
```

<table>
<thead>
<tr>
<th>Prolog</th>
<th>Id r1, ()</th>
<th></th>
<th></th>
<th>dec RRB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Id r1, ()</td>
<td>add r3, r2, #1</td>
<td></td>
<td></td>
<td>dec RRB</td>
</tr>
<tr>
<td>Kernel</td>
<td>Id r1, ()</td>
<td>add r3, r2, #1</td>
<td>st r4, ()</td>
<td>bloop</td>
</tr>
<tr>
<td></td>
<td>add r2, r1, #1</td>
<td>st r4, ()</td>
<td>dec RRB</td>
<td></td>
</tr>
<tr>
<td>Epilog</td>
<td></td>
<td></td>
<td></td>
<td>dec RRB</td>
</tr>
</tbody>
</table>

Loop closing branch decrements RRB
### VLIW Introduction

- **Architecture Support: Rotating Register Files**

  - Three cycle load latency encoded as difference of 3 in register specifier number (f4 - f1 = 3)
  - Four cycle fadd latency encoded as difference of 4 in register specifier number (f9 – f5 = 4)

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Id P9, ()</td>
<td>fadd P13, P12,</td>
<td>st P17, ()</td>
<td>bloop</td>
<td></td>
</tr>
</tbody>
</table>
| Id P8, () | fadd P12, P11, | st P16, () | bloop | RRB=8
| Id P7, () | fadd P11, P10, | st P15, () | bloop | RRB=7
| Id P6, () | fadd P10, P9,  | st P14, () | bloop | RRB=6
| Id P5, () | fadd P9, P8,   | st P13, () | bloop | RRB=5
| Id P4, () | fadd P8, P7,   | st P12, () | bloop | RRB=4
| Id P3, () | fadd P7, P6,   | st P11, () | bloop | RRB=3
| Id P2, () | fadd P6, P5,   | st P10, () | bloop | RRB=2
|    |    |    |    |    |

- **Instruction Examples**

  - ld f1, ()
  - st f9, ()
  - fadd f5, f4, ...
  - bloop
Limits of Instruction-Level Parallelism

- Limits of Instruction-Level Parallelism

  David Wall, 1993

Limits to ILP

- Conflicting studies of amount
  - Benchmarks (vectorized Fortran FP vs. integer C programs)
  - Hardware sophistication
  - Compiler sophistication

- How much ILP is available using existing mechanisms with increasing HW budgets?

- Do we need to invent new HW/SW mechanisms to keep on processor performance curve?
  - Intel MMX, SSE (Streaming SIMD Extensions): 64 bit ints
  - Intel SSE2: 128 bit, including 2 64-bit Fl. Pt. per clock
  - Motorola Altivec: 128 bit ints and FPs
  - Supersparc Multimedia ops, etc.
Overcoming Limits

- Advances in compiler technology + significantly new and different hardware techniques may be able to overcome limitations assumed in studies.
- However, unlikely such advances when coupled with realistic hardware will overcome these limits in near future.
Limits to ILP

Initial HW Model here; MIPS compilers.

Assumptions for ideal/perfect machine to start:

1. *Register renaming* – infinite virtual registers
   => all register WAW & WAR hazards are avoided
2. *Branch prediction* – perfect; no mispredictions
3. *Jump prediction* – all jumps perfectly predicted (returns, case statements)
   2 & 3 ⇒ no control dependencies; perfect speculation & an unbounded buffer of instructions available
4. *Memory-address alias analysis* – addresses known & a load can be moved before a store provided addresses not equal;
   1&4 eliminates all but RAW

Also: perfect caches; 1 cycle latency for all instructions (FP *,/);
unlimited instructions issued/clock cycle
## Limits to ILP HW Model comparison

<table>
<thead>
<tr>
<th>Model</th>
<th>Power 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instructions Issued per clock</td>
<td>Infinite</td>
</tr>
<tr>
<td>Instruction Window Size</td>
<td>Infinite</td>
</tr>
<tr>
<td>Renaming Registers</td>
<td>Infinite</td>
</tr>
<tr>
<td>Branch Prediction</td>
<td>Perfect</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>Cache</td>
<td>Perfect</td>
</tr>
<tr>
<td>Memory Alias Analysis</td>
<td>Perfect</td>
</tr>
</tbody>
</table>
Upper Limit to ILP: Ideal Machine

- **Integer**: 18 - 60
- **FP**: 75 - 150

- Programs:
  - gcc: 54.8
  - espresso: 62.6
  - li: 17.9
  - fpppp: 75.2
  - doducd: 118.7
  - tomcatv: 150.1
## Limits to ILP HW Model comparison

<table>
<thead>
<tr>
<th></th>
<th>New Model</th>
<th>Model</th>
<th>Power 5</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Instructions Issued per</strong></td>
<td><strong>Infinite</strong></td>
<td><strong>Infinite</strong></td>
<td>4</td>
</tr>
<tr>
<td><strong>clock</strong></td>
<td>Inf,**2K, 512,**128,<strong>32</strong></td>
<td><strong>Infinite</strong></td>
<td>200</td>
</tr>
<tr>
<td><strong>Instruction Window Size</strong></td>
<td>**Infinite, 2K, 512,**128,<strong>32</strong></td>
<td><strong>Infinite</strong></td>
<td>48 integer + 40 Fl. Pt.</td>
</tr>
<tr>
<td><strong>Renaming Registers</strong></td>
<td><strong>Infinite</strong></td>
<td><strong>Infinite</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Branch Prediction</strong></td>
<td><strong>Perfect</strong></td>
<td><strong>Perfect</strong></td>
<td>2% to 6% misprediction (Tournament Branch Predictor)</td>
</tr>
<tr>
<td><strong>Cache</strong></td>
<td><strong>Perfect</strong></td>
<td><strong>Perfect</strong></td>
<td>64KI, 32KD, 1.92MB L2, 36 MB L3</td>
</tr>
<tr>
<td><strong>Memory Alias</strong></td>
<td><strong>Perfect</strong></td>
<td><strong>Perfect</strong></td>
<td>??</td>
</tr>
</tbody>
</table>
More Realistic HW: Window Impact

Change from Infinite window
2048, 512, 128, 32

FP: 9 - 150

Integer: 8 - 63
## Limits to ILP HW Model comparison

<table>
<thead>
<tr>
<th></th>
<th>New Model</th>
<th>Model</th>
<th>Power 5</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Instructions Issued per clock</strong></td>
<td><strong>64</strong></td>
<td>Infinite</td>
<td>4</td>
</tr>
<tr>
<td><strong>Instruction Window Size</strong></td>
<td><strong>2048</strong></td>
<td>Infinite</td>
<td>200</td>
</tr>
<tr>
<td><strong>Renaming Registers</strong></td>
<td>Infinite</td>
<td>Infinite</td>
<td>48 integer + 40 Fl. Pt.</td>
</tr>
<tr>
<td><strong>Branch Prediction</strong></td>
<td><strong>Perfect vs. 8K Tournament vs. 512 2-bit vs. profile vs. none</strong></td>
<td>Perfect</td>
<td>2% to 6% misprediction (Tournament Branch Predictor)</td>
</tr>
<tr>
<td><strong>Cache</strong></td>
<td>Perfect</td>
<td>Perfect</td>
<td>64KI, 32KD, 1.92MB L2, 36 MB L3</td>
</tr>
<tr>
<td><strong>Memory Alias</strong></td>
<td>Perfect</td>
<td>Perfect</td>
<td>??</td>
</tr>
</tbody>
</table>
More Realistic HW: Branch Impact

Change from Infinite window to 2048, and maximum issue of 64 instructions per clock cycle

**Integer: 6 - 12**

**FP: 15 - 45**

<table>
<thead>
<tr>
<th>Program</th>
<th>Perfect</th>
<th>Tournament</th>
<th>BHT (512)</th>
<th>Profile</th>
<th>No prediction</th>
</tr>
</thead>
<tbody>
<tr>
<td>gcc</td>
<td>9</td>
<td>6</td>
<td>6</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>espresso</td>
<td>41</td>
<td>12</td>
<td>7</td>
<td>6</td>
<td>2</td>
</tr>
<tr>
<td>li</td>
<td>16</td>
<td>10</td>
<td>6</td>
<td>7</td>
<td>2</td>
</tr>
<tr>
<td>fpppp</td>
<td>48</td>
<td>46</td>
<td>45</td>
<td>45</td>
<td>58</td>
</tr>
<tr>
<td>doducd</td>
<td>15</td>
<td>13</td>
<td>14</td>
<td>4</td>
<td>46</td>
</tr>
<tr>
<td>tomcatv</td>
<td>46</td>
<td>45</td>
<td>45</td>
<td>19</td>
<td>46</td>
</tr>
</tbody>
</table>

Program  | Instruction issues per cycle
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>gcc</td>
<td>35</td>
</tr>
<tr>
<td>espresso</td>
<td>41</td>
</tr>
<tr>
<td>li</td>
<td>16</td>
</tr>
<tr>
<td>fpppp</td>
<td>48</td>
</tr>
<tr>
<td>doducd</td>
<td>15</td>
</tr>
<tr>
<td>tomcatv</td>
<td>46</td>
</tr>
</tbody>
</table>
Misprediction Rates

![Misprediction Rate Graph]

- Profile-based
- 2-bit counter
- Tournament

Table of Misprediction Rates:

<table>
<thead>
<tr>
<th></th>
<th>Profile-based</th>
<th>2-bit counter</th>
<th>Tournament</th>
</tr>
</thead>
<tbody>
<tr>
<td>tomcatv</td>
<td>1%</td>
<td>0%</td>
<td>0%</td>
</tr>
<tr>
<td>doduc</td>
<td>3%</td>
<td>16%</td>
<td>5%</td>
</tr>
<tr>
<td>fpppp</td>
<td>18%</td>
<td>14%</td>
<td>2%</td>
</tr>
<tr>
<td>li</td>
<td>2%</td>
<td>12%</td>
<td>2%</td>
</tr>
<tr>
<td>espresso</td>
<td>14%</td>
<td>18%</td>
<td>4%</td>
</tr>
<tr>
<td>gcc</td>
<td>12%</td>
<td>30%</td>
<td>6%</td>
</tr>
</tbody>
</table>
## Limits to ILP HW Model comparison

<table>
<thead>
<tr>
<th></th>
<th>New Model</th>
<th>Model</th>
<th>Power 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instructions Issued per clock</td>
<td><strong>64</strong></td>
<td>Infinite</td>
<td>4</td>
</tr>
<tr>
<td>Instruction Window Size</td>
<td><strong>2048</strong></td>
<td>Infinite</td>
<td>200</td>
</tr>
<tr>
<td>Renaming Registers</td>
<td>Infinite v. 256, 128, 64, 32, none</td>
<td>Infinite</td>
<td>48 integer + 40 Fl. Pt.</td>
</tr>
<tr>
<td>Branch Prediction</td>
<td><strong>8K 2-bit</strong></td>
<td>Perfect</td>
<td>Tournament Branch Predictor</td>
</tr>
<tr>
<td>Cache</td>
<td>Perfect</td>
<td>Perfect</td>
<td>64KI, 32KD, 1.92MB L2, 36 MB L3</td>
</tr>
<tr>
<td>Memory Alias</td>
<td>Perfect</td>
<td>Perfect</td>
<td>Perfect</td>
</tr>
</tbody>
</table>
More Realistic HW: Renaming Register Impact

Change to 2048 instr window, 64 instr issue, 8K 2 level Prediction

FP: 11 - 45
Integer: 5 - 15

Program

gcc espresso li fpopp doducd tomcatv

Infinite 256 128 64 32 None
## Limits to ILP HW Model comparison

<table>
<thead>
<tr>
<th></th>
<th>New Model</th>
<th>Model</th>
<th>Power 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instructions Issued per clock</td>
<td>64</td>
<td>Infinite</td>
<td>4</td>
</tr>
<tr>
<td>Instruction Window Size</td>
<td>2048</td>
<td>Infinite</td>
<td>200</td>
</tr>
<tr>
<td>Renaming Registers</td>
<td>256 Int + 256 FP</td>
<td>Infinite</td>
<td>48 integer + 40 Fl. Pt.</td>
</tr>
<tr>
<td>Branch Prediction</td>
<td>8K 2-bit</td>
<td>Perfect</td>
<td>Tournament</td>
</tr>
<tr>
<td>Cache</td>
<td>Perfect</td>
<td>Perfect</td>
<td>64KI, 32KD, 1.92MB L2, 36 MB L3</td>
</tr>
<tr>
<td>Memory Alias</td>
<td>Perfect v. Stack v. Inspect v. none</td>
<td>Perfect</td>
<td>Perfect</td>
</tr>
</tbody>
</table>
More Realistic HW: Memory Address Alias Impact

Change 2048 instr window, 64 instr issue, 8K 2 level Prediction, 256 renaming registers

FP: 4 - 45 (Fortran, no heap)

Perfect Global/stack perf; Compiler None

heap conflicts Inspection

IPC

Program

Perfect Global/stack Perfect Inspection None

gcc espresso li fpppp doducd tomcatv

Integer: 4 - 9

IPC

10 7 4 3
15 7 5 5
12 9 4 3
49 49 4 3
16 16 6 4
45 45 5 4
## Limits to ILP HW Model comparison

<table>
<thead>
<tr>
<th></th>
<th>New Model</th>
<th>Model</th>
<th>Power 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instructions Issued per clock</td>
<td>64 (no restrictions)</td>
<td>Infinite</td>
<td>4</td>
</tr>
<tr>
<td>Instruction Window Size</td>
<td>Infinite vs. 256, 128, 64, 32</td>
<td>Infinite</td>
<td>200</td>
</tr>
<tr>
<td>Renaming Registers</td>
<td>64 Int + 64 FP</td>
<td>Infinite</td>
<td>48 integer + 40 Fl. Pt.</td>
</tr>
<tr>
<td>Branch Prediction</td>
<td>1K 2-bit</td>
<td>Perfect</td>
<td>Tournament</td>
</tr>
<tr>
<td>Cache</td>
<td>Perfect</td>
<td>Perfect</td>
<td>64KI, 32KD, 1.92MB L2, 36 MB L3</td>
</tr>
<tr>
<td>Memory Alias</td>
<td>HW disambiguation</td>
<td>Perfect</td>
<td>Perfect</td>
</tr>
</tbody>
</table>
Realistic HW: Window Impact

Perfect disambiguation (HW), 1K
Selective Prediction, 16 entry return, 64
registers, issue as many as window

FP: 8 - 45

Integer: 6 - 12

Program

<table>
<thead>
<tr>
<th>Program</th>
<th>gcc</th>
<th>expresso</th>
<th>li</th>
<th>fpppp</th>
<th>doducd</th>
<th>tomcatv</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Infinite</td>
<td>256</td>
<td>128</td>
<td>64</td>
<td>32</td>
<td>16</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>15</td>
<td>15</td>
<td>13</td>
</tr>
<tr>
<td></td>
<td>9</td>
<td>8</td>
<td>6</td>
<td>10</td>
<td>6</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>4</td>
<td>3</td>
<td>12</td>
<td>11</td>
<td>11</td>
</tr>
</tbody>
</table>

Infinite 256 128 64 32 16 8 4