Design and Implementation of the TRIPS EDGE Architecture

TRIPS Tutorial
Originally Presented at ISCA-32
June 4, 2005

Department of Computer Sciences
The University of Texas at Austin
Tutorial Outline

• Part I: Overview
  – Introduction *(Doug Burger/Steve Keckler)*
  – EDGE architectures and the TRIPS ISA *(Doug Burger)*
  – TRIPS microarchitecture and prototype overview *(Steve Keckler)*
  – TRIPS code examples *(Robert McDonald)*

• Part II: TRIPS Front End
  – Global control *(Ramdas Nagarajan)*
  – Control-flow prediction *(Nitya Ranganathan)*
  – Instruction fetch *(Haiming Liu)*
  – Register accesses and operand routing *(Karu Sankaralingam)*

• Part III: TRIPS Execution Core & Memory System
  – Issue logic and execution *(Premkishore Shivakumar)*
  – Primary memory system *(Simha Sethumadhavan)*
  – Secondary memory system *(Changkyu Kim)*
  – Chip- and system-level networks and I/O *(Paul Gratz)*

• Part IV: TRIPS Compiler
  – Compiler overview *(Kathryn McKinley)*
  – Forming TRIPS Blocks *(Aaron Smith)*
  – Code optimization *(Kathryn McKinley)*

• Part V: Conclusions

*Names in italics refer to both the presenters at ISCA-32 and the main developers of each section’s slide deck.*
Copyright Information

- The material in this tutorial is Copyright © 2005-2006 by The University of Texas at Austin. All rights reserved.

- This material was developed as a part of the TRIPS project in the Computer Architecture and Technology Laboratory, Department of Computer Sciences, at The University of Texas at Austin. Please contact Doug Burger (dburger@cs.utexas.edu) or Steve Keckler (skeckler@cs.utexas.edu) for information about redistribution or usage of these materials in other presentations.

- Portions of the TRIPS technology described in this tutorial are patent pending. Commercial parties interested in licensing or using TRIPS technology for profit should contact the project principal investigators listed above.
Acknowledgment of TRIPS Sponsors

• DARPA
    • Contracts F33615-01-C-1892 and F44615-03-C-4106
  – Special thanks to Bob Graybill

• Air Force Research Laboratory

• Intel Research Council

• IBM

• Sun Microsystems

• National Science Foundation
  – CAREER and Research Infrastructure Grants (1999-2008)

• Peter O’Donnell Foundation

• UT-Austin College of Natural Sciences
  – Matching funds for industrial fellows, infrastructure grants (1999-2008)
## Other Members of the TRIPS Team

### Research Scientists
- **Bill Yoder**  
  - Chief developer of the TRIPS toolchain
- **Jim Burrill**  
  - Chief engineer of the Scale compiler
- **Nick Nethercote**  
  - TRIPS compiler performance leader

### Graduate Students
- **Xia Chen**  
  - PowerPC to TRIPS translation and emulation
- **Raj Desikan**  
  - Initial data tile design
- **Saurabh Drolia**  
  - Prototype DMA controller, system simulator, parallel software
- **Madhu Sibi Govindan**  
  - Prototype external bus and clock controller
- **Divya Gulati**  
  - Execution tile design, processor core verification
- **Heather Hanson**  
  - Operand router design
- **Sundeep Kushwaha**  
  - Back-end instruction placement
- **Bert Maher**  
  - Hyperblock construction and formation
- **Suriya Narayanan**  
  - Compiler/memory system optimization
- **Sadia Sharif**  
  - TRIPS system software
Relevant TRIPS/EDGE Publications

- “The Design and Implementation of the TRIPS Prototype Chip”
  - HotChips 17, 2005.
  - Contains details about the prototype ASIC.
- “Dynamic Placement, Static Issue (SPDI) Scheduling for EDGE Architectures”
  - Specifies compiler algorithm for scheduling instructions on the TRIPS architecture.
- “Scaling to the End of Silicon with EDGE Architectures”
  - IEEE Computer, July, 2004
  - Provides an overview of EDGE architectures using the TRIPS architecture as a case study.
- “Exploiting ILP, TLP, and DLP with the Polymorphous TRIPS Architecture”
  - Details the flexibility of the TRIPS architecture for exploiting different types of parallelism.
- “An Adaptive, Non-Uniform Cache Structure for Wire-Dominated On-Chip Caches”
  - The first paper describing NUCA caches and their design tradeoffs.
- “A Design Space Evaluation of Grid Processor Architectures”
  - An early study that formed the basis for the TRIPS architecture.
Principal Related Work

• Transport-Triggered Architectures [Supercomputing 1991]
  – Hans Mulder and Henk Corporaal
  – MOVE Architecture had direct instruction-to-instruction communication bypassing registers

• WaveScalar [Micro 2003]
  – Mark Oskin, Steve Swanson, Ken Michelson, and Andrew Schwerin
  – Imperative/dataflow EDGE ISA
  – 3 most significant differences with TRIPS:
    • Dynamic instruction placement
    • All-path execution
    • Two-level microarchitectural execution hierarchy

• ASH/CASH/Pegasus IR [CGO-03, ASPLOS-04, ISPASS-05]
  – Mihai Budiu and Seth Goldstein
  – Similar goals and related compiler techniques
  – More of a focus on compiling to ASICs or reconfigurable substrates

• RAW [IEEE Computer-97, ISCA-04]
  – Pioneering tiled architecture tackling concurrency and wire delays
  – Leading work on Scalar Operand Networks
  – Direct instruction-to-instruction communication through network switch I-stream
Key Trends

- **Power**
  - Now a budget, just like area
  - What performance can you get for a given power and area budget?
  - Transferring work from the HW to the SW (compiler, programmer, etc.) is power efficient
  - Leakage trends must be addressed (not underlying goal of TRIPS)

- **Slowing (stopping?) frequency increases**
  - Pipelining has reached depth limits
  - Device speed scaling may stop tracking feature size reduction
  - May result in decreasing main memory latencies
  - Must exploit more concurrency
  - Key issue: how to expose this concurrency? Force the programmer?

- **Wire Delays**
  - Will force growing partitioning of future chips
  - Works against reduced power and improved concurrency

- **Reliability**
  - Do everything twice (or thrice) at some level of abstraction
  - Works against power limits and exploitation of concurrency
Performance Scaling and Technology Challenges

Single-processor Performance Scaling

- **Log2 Speedup**
  - New programming models needed?
  - Concurrency only solution for higher performance
  - Conventional architectures cannot improve performance
  - Industry shifts to frequency dominated strategy

- **55%/year improvement**
- **RISC/CISC CPI**
- **Pipelining**
- **Device speed**
- **Architectural frequency wall**
- **Frequency wall**
- **EDGE CPI**

**Timeline:**
- 1984
- 1986
- 1988
- 1990
- 1992
- 1994
- 1996
- 1998
- 2000
- 2002
- 2004
- 2006
- 2008
- 2010
- 2012
- 2014
- 2016
- 2018
- 2020

**Technology Nodes:**
- 90 nm
- 65 nm
- 45 nm
- 35 nm
- 25 nm

June 4, 2005 UT-Austin TRIPS Tutorial 9
EDGE Architectures: Can New ISAs Help?

'60s, '70s
- Complex, few instructions
- Discrete components
- Minimize storage
- Small numbers in flight
- Pipelining difficult

'CISC

'80s, '90s, early '00s
- More numerous, simple instructions
- Reliance on compiler ordering
- Optimized for pipelining
- Tens of instructions in flight
- Wide issue inefficient

'RISC¹

late '00s, '10s
- Blocks amortize per-inst overhead
- Hundreds to thousands in flight
- Inter-inst. communication explicit
- Exploits more concurrency
- Leakage not yet addressed
- Reliability not yet addressed

'EDGE?

¹RISC concepts implemented in both ISA and H/W
The TRIPS EDGE ISA

ISCA-32 Tutorial

Doug Burger

Department of Computer Sciences
The University of Texas at Austin
What is an EDGE Architecture?

- **Explicit Data Graph Execution**
  - Defined by two key features

  1. Program graph is broken into sequences of *blocks*
     - Basic blocks, hyperblocks, or something else
     - Blocks commit atomically or not - a block never partially executes
  2. Within a block, ISA support for direct producer-to-consumer communication
     - No shared named registers within a block (point-to-point dataflow edges only)
     - Caveat: memory is still a shared namespace
     - The block’s dataflow graph (DFG) is explicit in the architecture

- What are design alternatives for different architectures?
  - Single path through program block graph (TRIPS)
  - All paths through program block graph (WaveScalar)
  - Mechanism for specifying instruction-to-instruction links in the ISA
  - Block constraints (composition, fixed size vs. variable size, etc.)
Architectural Structure of a TRIPS Block

Block characteristics:
- **Fixed size:**
  - 128 instructions max
  - L1 and core expands empty 32-inst chunks to NOPs
- **Load/store IDs:**
  - Maximum of 32 loads+stores may be emitted, but blocks can hold more than 32
- **Registers:**
  - 8 read insts. max to reg. bank (4 banks = max of 32)
  - 8 write insts. max to reg bank (4 banks = max of 32)
- **Control flow:**
  - Exactly one branch emitted
  - Blocks may hold more
Blocks encoded in object file include:
• Read/Write instructions
• Computation instructions
• 32-bit store mask for tracking store completion

Two major considerations
• Every possible execution of a given block must emit a consistent number of outputs (register writes, stores, PC)
  – Outputs may be actual or null
  – Different blocks may have different #s of outputs
• No block state may be written until block is committed
  – Bulk commit is logically atomic
  – Similar to an architectural checkpoint in some respects
### TRIPS Block Example

<table>
<thead>
<tr>
<th>RISC code</th>
<th>TIL (operand format)</th>
<th>TASL (target format)</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{ld} ) ( R3, 4(R2) )</td>
<td>.\texttt{bbegin} block1</td>
<td>([R1] \ $g1 \ [2])</td>
</tr>
<tr>
<td>( \text{add} ) ( R4, R1, R3 )</td>
<td>read ( $t1, \ $g1 )</td>
<td>([R2] \ $g2 \ [1] \ [4])</td>
</tr>
<tr>
<td>( \text{st} ) ( R4, 4(R2) )</td>
<td>read ( $t2, \ $g2 )</td>
<td>([1] \ \text{ld} \ \text{L}[1] 4 \ [2])</td>
</tr>
<tr>
<td>( \text{addi} ) ( R5, R4, #2 )</td>
<td>( \text{ld} ) ( $t3, 4($t2) )</td>
<td>([2] \ \text{add} \ [3] \ [4])</td>
</tr>
<tr>
<td>( \text{beqz} ) ( R4, \text{Block3} )</td>
<td>( \text{add} ) ( $t4, \ $t1, \ $t3 )</td>
<td>([3] \ \text{mov} \ [5] \ [6])</td>
</tr>
<tr>
<td>( )</td>
<td>( \text{st} ) ( $t4, 4($t2) )</td>
<td>([4] \ \text{st} \ S[2] 4)</td>
</tr>
<tr>
<td>( )</td>
<td>( \text{addi} ) ( $t5, $t4, #2 )</td>
<td>([5] \ \text{addi} 2 \ [W1])</td>
</tr>
<tr>
<td>( )</td>
<td>( \text{teqz} ) ( $t6, $t4 )</td>
<td>([6] \ \text{teqz} \ [7] \ [8])</td>
</tr>
<tr>
<td>( )</td>
<td>( \text{b}_t&lt;$t6&gt; \text{block3} )</td>
<td>([7] \ \text{b}_t \ \text{block3})</td>
</tr>
<tr>
<td>( )</td>
<td>( \text{b}_f&lt;$t6&gt; \text{block2} )</td>
<td>([8] \ \text{b}_f \ \text{block2})</td>
</tr>
<tr>
<td>( )</td>
<td>( \text{write} ) ( $g5, \ $t5 )</td>
<td>([W1] \ $g5)</td>
</tr>
</tbody>
</table>
| \( \) | .\texttt{bend} block1 | .\texttt{bbegin} block2 ...

- **Read target format**
- **Predicated instructions**
- **LD/ST sequence numbers**
- **Target fanout with movs**
- **Block outputs fixed**
  (3 in this example)
Key Features of TRIPS ISA

- Fixed instruction lengths - all instructions 32 bits
- Read and write instructions
  - Contained in block header for managing DFG/register file communication
- Target format \((T0, T1)\)
  - Output destinations specified with 9-bit targets
- Predication \((PR)\)
  - Nearly every instruction has a predicate field, encoded for efficient dataflow predication
- Load/store IDs \((LSID)\)
  - Used to maintain sequential memory semantics despite EDGE ISA
- Exit bits \((EXIT)\)
  - Used to improve block exit control prediction
### TRIPS Instruction Formats

#### General Instruction Formats
```
<table>
<thead>
<tr>
<th>31</th>
<th>25 24 23 22</th>
<th>18 17</th>
<th>9 8</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>OPCODE</td>
<td>PR</td>
<td>XOP</td>
<td>T1</td>
<td>T0</td>
</tr>
<tr>
<td>OPCODE</td>
<td>PR</td>
<td>XOP</td>
<td>IMM</td>
<td>T0</td>
</tr>
</tbody>
</table>
```

#### Load and Store Instruction Formats
```
<table>
<thead>
<tr>
<th>31</th>
<th>25 24 23 22</th>
<th>18 17</th>
<th>9 8</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>OPCODE</td>
<td>PR</td>
<td>LSID</td>
<td>IMM</td>
<td>T0</td>
</tr>
<tr>
<td>OPCODE</td>
<td>PR</td>
<td>LSID</td>
<td>IMM</td>
<td>0</td>
</tr>
</tbody>
</table>
```

#### Branch Instruction Format
```
<table>
<thead>
<tr>
<th>31</th>
<th>25 24 23 22</th>
<th>20 19</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>OPCODE</td>
<td>PR</td>
<td>EXIT</td>
<td>OFFSET</td>
</tr>
</tbody>
</table>
```

#### Constant Instruction Format
```
<table>
<thead>
<tr>
<th>31</th>
<th>25 24</th>
<th>9 8</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>OPCODE</td>
<td></td>
<td>CONST</td>
<td>T0</td>
</tr>
</tbody>
</table>
```

#### Read Instruction Format
```
<table>
<thead>
<tr>
<th>21 20</th>
<th>16 15</th>
<th>8 7</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>V</td>
<td>GR</td>
<td>RT0</td>
<td>RT1</td>
</tr>
</tbody>
</table>
```

#### Write Instruction Format
```
<table>
<thead>
<tr>
<th>5 4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>GR</td>
<td></td>
</tr>
</tbody>
</table>
```

### INSTRUCTION FIELDS
- **OPCODE** = Primary Opcode
- **XOP** = Extended Opcode
- **PR** = Predicate Field
- **IMM** = Signed Immediate
- **T0** = Target 0 Specifier
- **T1** = Target 1 Specifier
- **LSID** = Load/Store ID
- **EXIT** = Exit Number
- **OFFSET** = Branch Offset
- **CONST** = 16-bit Constant
- **V** = Valid Bit
- **GR** = General Register Index
- **RT0** = Read Target 0 Specifier
- **RT1** = Read Target 1 Specifier

Not shown: M3, M4 formats
TRIPS G and L formats

G:

<table>
<thead>
<tr>
<th>7</th>
<th>2</th>
<th>5</th>
<th>9</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>opcode</td>
<td>Pred</td>
<td>xop</td>
<td>T1</td>
<td>Target field</td>
</tr>
</tbody>
</table>

- 00: unpredicated
- 01: reserved
- 10: predicated on false
- 11: predicated on true

L:

<table>
<thead>
<tr>
<th>7</th>
<th>2</th>
<th>5</th>
<th>9</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>opcode</td>
<td>pr</td>
<td>LSID</td>
<td>IMM</td>
<td>T0</td>
</tr>
</tbody>
</table>

- 5-bit sequence number for ordering loads/stores
- 9-bit address index for calculating effective address
Target (Operand) Formats

- 00: No target (or write inst.)
- 01: Targets predicate field
- 10: Targets left operand
- 11: Targets right operand

- Names one of 128 target instructions
- Location of instruction is microarchitecture dependent

- Names one of 32 write instructions
Object Code Example

TASL (target format)

[R1] $g1  [2]
[R2] $g2  [1] [4]
[5] addi 2 [W1]
[7] b_t  block3
[8] b_f  block2
[W1] $g5

Object Code

<table>
<thead>
<tr>
<th>v</th>
<th>reg</th>
<th>T1</th>
<th>T0</th>
</tr>
</thead>
<tbody>
<tr>
<td>v</td>
<td>reg</td>
<td>T1</td>
<td>T0</td>
</tr>
<tr>
<td>v</td>
<td>reg</td>
<td>T1</td>
<td>T0</td>
</tr>
<tr>
<td>v</td>
<td>reg</td>
<td>T1</td>
<td>T0</td>
</tr>
<tr>
<td>v</td>
<td>reg</td>
<td>T1</td>
<td>T0</td>
</tr>
<tr>
<td>v</td>
<td>reg</td>
<td>T1</td>
<td>T0</td>
</tr>
</tbody>
</table>

opcode  pr  LSID  imm  T0
opcode  pr  xop  T1   T0
opcode  pr  xop  T1   T0
opcode  pr  LSID  imm  0
opcode  pr  xop  imm  T0
opcode  pr  xop  T1   T0
opcode  pr  exit offset
opcode  pr  exit offset

June 4, 2005  UT-Austin TRIPS Tutorial
### Object Code Example

**TASL (target format)**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Arguments</th>
</tr>
</thead>
<tbody>
<tr>
<td>[R1] $g1</td>
<td>[2]</td>
</tr>
<tr>
<td>[R2] $g2</td>
<td>[1] [4]</td>
</tr>
<tr>
<td>[5] addi</td>
<td>2 [W1]</td>
</tr>
<tr>
<td>[7] b_t</td>
<td>block3</td>
</tr>
<tr>
<td>[8] b_f</td>
<td>block2</td>
</tr>
<tr>
<td>[W1] $g5</td>
<td></td>
</tr>
</tbody>
</table>

**Object Code**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Address</th>
<th>Data</th>
<th>Immediate</th>
</tr>
</thead>
<tbody>
<tr>
<td>ld</td>
<td>00</td>
<td>00001</td>
<td>4</td>
</tr>
<tr>
<td>add</td>
<td>00</td>
<td>---</td>
<td>[3] [4]</td>
</tr>
<tr>
<td>mov</td>
<td>00</td>
<td>---</td>
<td>[5] [6]</td>
</tr>
<tr>
<td>st</td>
<td>00</td>
<td>00010</td>
<td>4</td>
</tr>
<tr>
<td>addi</td>
<td>00</td>
<td>---</td>
<td>2</td>
</tr>
<tr>
<td>teqz</td>
<td>00</td>
<td>---</td>
<td>[7] [8]</td>
</tr>
<tr>
<td>b_t</td>
<td>11</td>
<td>(t)</td>
<td>E0</td>
</tr>
<tr>
<td>b_f</td>
<td>10</td>
<td>(f)</td>
<td>E1</td>
</tr>
</tbody>
</table>

Store mask: 000000000000000000000000000000100
TRIPS Block Format

Each block is formed from two to five 128-byte program “chunks”

Blocks with fewer than five chunks are expanded to five chunks in the L1 I-cache

The header chunk includes a block header (execution flags plus a store mask) and register read/write instructions

Each instruction chunk holds 32 4-byte instructions (including NOPs)

A maximally sized block contains 128 regular instructions, 32 read instructions, and 32 write instructions
Example of Predicate Operation

- Test instructions output low-order bit of 1 or 0 (T or F)
- Predicated instructions either match the arriving predicate or not
  - An ADD with PR=10 is predicated on false, an arriving 0 predicate will match and enable the ADD
  - Predicated instructions must receive all operands and a matching predicate before firing
  - Instructions may receive multiple predicate operands but at most one matching predicate
  - Despite predicates, blocks must produce fixed # of outputs.

```
if (p==0)  z = a * 2 + 3;
else  z = b * 3 + 4;
```
Predicate Optimization and Fanout Reduction

- **Predicate dependence heads**
  - Reduce instructions executed
  - Reduces dynamic energy by using less speculation

- **Predicate dependence tails**
  - Tolerate predicate latencies with speculatively executed insts.
  - Cannot speculate on block outputs

![Image of predicate optimization and fanout reduction](image-url)
Instruction (Store) Merging

- **Old**: two copies of stores down distinct predicate paths
  - Old: two copies of stores down distinct predicate paths
  - Mult: multi
  - Add: addi
  - Store: st(1)

- **New**: merge redundant insts.
  - Speculative copies unneeded
  - Only one data source will fire

\[ P \rightarrow \text{teqz} \rightarrow \text{muli} \rightarrow \text{addi} \rightarrow \text{st}(1) \]

\[ P \rightarrow \text{teqz} \rightarrow \text{muli} \rightarrow \text{addi} \rightarrow \text{st}(1) \]
Nullified Outputs

• Each transmitted operand tagged with “null” and “exception” tokens.
  – Nullified operands arriving at insts. produce nullified outputs
  – Nulls used to indicate that a block won’t produce a certain output (writes and stores)

Block with store down one conditional path:
...
if (p==0)
  z = a * 2 + 3;
...

```
null
```
Exception Handling

• TRIPS exception types:
  – fetch, breakpoint, timeout, execute, store, system call, reset, interrupt

• TRIPS exception model demands
  – Must not raise exceptions for speculative instructions

• Key concept: propagate exception tokens within dataflow graph
  – Instructions with exception-set operands do not execute, but propagate exception token to their targets
  – Exceptions are detected at commit time if one or more block output (write, store, branch) has exception bit set
  – Exceptions serviced between blocks
    • “Block-precise” exception model
Other Architecturally Supported Features

- **Execution mode via Thread Control Register**
  - Types of speculation (load, branch)
  - Number of blocks in flight
  - Breakpoints before/after blocks execute
- **T-Morph via Processor Control Register**
  - Can place processors into single or 4-threaded SMT modes
- **Per-bank cache/scratchpad configurations**
- **TLBs managed by software**
- **On-chip network routing tables programmed by software**
- **40-bit global system memory address space**
  - Divided into SRF, cached memory, configuration quadrants
TRIPS Microarchitecture and Implementation

ISCA-32 Tutorial

Steve Keckler

Department of Computer Sciences
The University of Texas at Austin
TRIPS Microarchitecture Principles

- Limit wire lengths
  - Architecture is partitioned and distributed
    - Microarchitecture composed of different types of tiles
  - No centralized resources
  - Local wires are short
    - Tiles are small (2-5 mm$^2$ per tile is typical)
  - No global wires
  - Networks connect only nearest neighbors

- Design for scalability
  - Design productivity by replicating tiles (design reuse)
  - Tiles interact through well-defined control and data networks
    - Networks are extensible, even late in the design cycle
    - Opportunity for asynchronous interfaces
  - Prototype: employs communication latencies of 35nm technology, even though prototype is implemented in 130nm
TRIPS Tile-level Microarchitecture

TRIPS Tiles

G: Processor control - TLB w/ variable size pages, dispatch, next block predict, commit

R: Register file - 32 registers x 4 threads, register forwarding

I: Instruction cache - 16KB storage per tile

D: Data cache - 8KB per tile, 256-entry load/store queue, TLB

E: Execution unit - Int/FP ALUs, 64 reservation stations

M: Memory - 64KB, configurable as L2 cache or scratchpad

N: OCN network interface - router, translation tables

DMA: Direct memory access controller

SDC: DDR SDRAM controller

EBC: External bus controller - interface to external PowerPC

C2C: Chip-to-chip network controller - 4 links to XY neighbors
Grid Processor Tiles and Interfaces

- **GDN**: global dispatch network
- **OPN**: operand network
- **GSN**: global status network
- **GCN**: global control network
Mapping TRIPS Blocks to the Microarchitecture

- **Frame**
  - **D-tile[3]**
    - LSID
      - 0
      - 1
      - 2
      - 3
      - 4
      - 5
      - 6
      - 7
    - Frame
      - 0
      - 1
      - 2
      - 3
      - 4
      - 5
      - 6
      - 7

- **R-tile[3]**
  - Architecture
  - Register Files
  - Read/Write Queues
  - Thread

- **E-tile[3,3]**
  - Instruction reservation stations
  - Frame
  - Slot
    - 0
    - 1
    - 2
    - 3
    - 4
    - 5
    - 6
    - 7
  - OP1
  - OP2
Mapping TRIPS Blocks to the Microarchitecture

Block i mapped into Frame 0

Header
Chunk

Inst. Chunk 0

Inst. Chunk 3

D-tile[3]

LSID

E-tile[3,3]

Instruction reservation stations

Slot

Frame

Thread R-tile[3]

Architecture

Register Files

Read/Write Queues
# Mapping TRIPS Blocks to the Microarchitecture

## Header Chunk

- Block \(i+1\) mapped into Frame 1

## Instruction Chunk 0

- Instruction reservation stations

## Instruction Chunk 3

- Frame 0 1 2 3 4 5 6 7

## D-tile[3]

- Frame 0 1 2 3 4 5 6 7

## LSID

- Frame 0 1 2 3 4 5 6 7

## E-tile[3,3]

- Instruction reservation stations

- Slot 0 1 2 3 4 5 6 7

- Frame 0 1 2 3 4 5 6 7
Mapping Target Identifiers to Reservation Stations

Frame: 100
Slot: 101
OP: 10 = OP1

Target = 87, OP1

Type (2 bits)
Y (2 bits)
X (2 bits)
Frame (assigned by GT at runtime)
ISA Target Identifier

<table>
<thead>
<tr>
<th>Frame (3 bits)</th>
<th>Type (2 bits)</th>
<th>Y (2 bits)</th>
<th>Slot (3 bits)</th>
<th>X (2 bits)</th>
</tr>
</thead>
<tbody>
<tr>
<td>100 10 10 101 11</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

E-tile Instruction reservation stations
Processor Core Tiles and Interfaces

- Fetch
  - G-tile examines I$ tags
  - Sends dispatch command to I-tiles
  - I-tiles fetch instructions, distribute to R-tiles, E-tiles
Processor Core Tiles and Interfaces

- Execute
  - R-tiles inject register values
  - E-tiles execute block instructions
    - compute, load/store
  - E-tiles delivers outputs
    - Results to R-tiles/D-tiles
    - Branch to G-tile
Processor Core Tiles and Interfaces

- Commit
  - R-tiles/D-tiles accumulate outputs
    - Send completion messages to G-tile
  - G-tile sends commit message
  - R-tiles/D-tiles respond with commit acknowledge
    - R-tiles commit state to register files
    - D-tiles commit state to data cache
Execute/commit overlapped across multiple blocks
- G-tile manages frames as a circular buffer
  - D-morph: 1 thread, 8 frames
  - T-morph: up to 4 threads, 2 frames each
- **Flexible/fast on-chip network fabric**
  - Programmable translation tables
  - Configurable 1 MB memory system
  - 128-bit wide, 533 MHz
  - 6.8GB/sec per link

- **M-tile (on-chip memory)**
  - 64 KB SRAM capacity
  - Configurable tag array (on/off)

- **N-tile with translation tables**
  - System address $\Rightarrow$ OCN address

- **EBC - external bus controller**
  - Connection to master control processor

- **SDC - DDR1 SDRAM controller**
  - 1.1 GB/sec per controller

- **DMA controller**
  - Linear, scatter-gather, chaining
Chip-to-Chip (C2C) Network

- Glueless interchip network
  - Extension of OCN
  - 2D mesh router
  - 64-bits wide
  - 266MHz, 1.7GB/sec per link
Chip Implementation

- **Chip Specs**
  - 130 nm 7LM IBM ASIC process
  - 18.3mm x 18.3mm die
  - 853 signal I/Os
    - 568 for C2C
    - 216 for SDCs
    - ~70 misc I/Os
- **Schedule**
  - 5/99: Seed of TRIPS ideas
  - 12/01: Publish TRIPS concept architecture
  - 10/02: Publish NUCA concept architecture
  - 6/03: Start high-level prototype design
  - 10/03: Start RTL design
  - 2/05: RTL 90% complete
  - 7/05: Design 100% complete
  - 9/05: Tapeout
  - 12/05: Start system evaluation in lab
Prototype Simplifications

• Architecture simplifications
  – No hardware cache coherence mechanisms
  – No floating-point divider
  – No native exception handling
  – Simplified exception reporting
  – Limited support for variable-sized blocks

• Microarchitecture simplifications
  – No I-cache prefetching or block predict for I-cache refill
  – No variable resource allocation (for smaller blocks)
  – One cycle per hop on every network
  – No clock-gating
  – Direct-mapped and replicated LSQ (non-associative, oversized)
  – No selective re-execution
  – No fast block refetch
• TRIPS chip network
  – 4 TRIPS chips (8 processors) per board
  – Connected using a 2x2 chip-to-chip network
  – Each chip provides local and remote access to 2GB SDRAM

• PPC440GP
  – Configures and controls the TRIPS chips
  – Connects to a Host PC for user interface (via serial port or ethernet)
  – Runs an embedded Linux OS
  – Includes a variety of integrated peripherals

• SDRAM DIMMs
  – 8 slots for up to 8 GB TRIPS memory
  – 2 slots for up to 1 GB PPC memory

• FPGA - usage TBD but may include
  – Protocol translator for high-speed I/O devices
  – Programmable acceleration engine
Maximum Size TRIPS System
TRIPS Prototype System Capabilities

- Clock speeds
  - 533MHz internal clock
  - 266MHz C2C clock
  - 133/266 SDRAM clock

- Single chip
  - 2 processors per chip each with
    - 64 KB L1-I$, 32KB L1-D$
    - 8.5 Gops or GFlops/processor
  - 17 Gops or Gflops/chip peak
  - Up to 8 threads
  - 1 MB on-chip memory
    - L2 cache or scratchpad
  - 2 GB SDRAM
  - 2.2 GB/sec DRAM bandwidth

- Full system (8 boards)
  - 32 chips, 64 processors
  - 1024 64-bit FPUs
  - 545 Gops/Gflops
  - Up to 256 threads
  - 64GB SDRAM
  - 70 GB/s aggregate DRAM bandwidth
  - 6.8 GB/sec C2C bisection bandwidth
Code Examples

ISCA-32 Tutorial

Robert McDonald

Department of Computer Sciences
The University of Texas at Austin
Two Examples

• Vector Add Example
  – A simple for loop
  – Basic unrolling
  – TRIPS Intermediate Language (TIL)
  – Block Dataflow Graph
  – Placed Instructions
  – TRIPS Assembly Language (TASL)
  – Sample Execution

• Linked List Example
  – A while loop with pointers
  – TRIPS predication
  – Predicated loop unrolling
Vector Add – C Code

- Consider this simple C routine
- The routine does an add and accumulate for fixed-size vectors
- An initial control flow graph is shown

```c
#define VSIZE 1024

void vadd(double* A, double* B, double* C)
{
    int i;
    for (i = 0; i < VSIZE; i++)
    {
        C[i] += (A[i] + B[i]);
    }
}
```
Vector Add – Unrolled

- This loop wants to be unrolled
- Unrolling reduces the overhead per loop iteration
- It reduces the number of conditional branches that must be executed
Vector Add – TIL Code

- The compiler produces TRIPS Intermediate Language (TIL) files
- Each file defines one or more program blocks
- Each block includes instructions, normally listed in program order
- Instructions are defined using a familiar syntax (name, target, sources)
- Read and write instructions access registers
- All other instructions deal with temporaries
- Notice the Load/Store IDs
- Notice the predicated branches

```assembly
.bbegin vadd$1
  read $t0, $g70
  read $t1, $g71
  read $t2, $g72
  read $t3, $g73
  ld $t4, ($t1) L[0]
  ld $t5, ($t2) L[1]
  ld $t6, ($t3) L[2]
  fadd $t7, $t5, $t6
  fadd $t8, $t4, $t7
  sd ($t1), $t8 S[3]
  ld $t9, 8($t1) L[4]
  ld $t10, 8($t2) L[5]
  ld $t11, 8($t3) L[6]
  fadd $t12, $t10, $t11
  fadd $t13, $t9, $t12
  sd 8($t1), $t13 S[7]
  ld $t14, 16($t1) L[8]
  ld $t15, 16($t2) L[9]
  ld $t16, 16($t3) L[10]
  fadd $t17, $t15, $t16
  fadd $t18, $t14, $t17
  sd 16($t1), $t18 S[11]
  ld $t19, 24($t1) L[12]
  ld $t20, 24($t2) L[13]
  ld $t21, 24($t3) L[14]
  fadd $t22, $t20, $t21
  fadd $t23, $t19, $t22
  sd 24($t1), $t23 S[15]
  ld $t24, 32($t1) L[16]
  ld $t25, 32($t2) L[17]
  ld $t26, 32($t3) L[18]
  fadd $t27, $t25, $t26
  fadd $t28, $t24, $t27
  sd 32($t1), $t28 S[19]
(continued)
```

```assembly
  ld $t29, 40($t1) L[20]
  ld $t30, 40($t2) L[21]
  ld $t31, 40($t3) L[22]
  fadd $t32, $t30, $t31
  fadd $t33, $t29, $t32
  sd 40($t1), $t33 S[23]
  ld $t34, 48($t1) L[24]
  ld $t35, 48($t2) L[25]
  ld $t36, 48($t3) L[26]
  fadd $t37, $t35, $t36
  fadd $t38, $t34, $t37
  sd 48($t1), $t38 S[27]
  ld $t39, 56($t1) L[28]
  ld $t40, 56($t2) L[29]
  ld $t41, 56($t3) L[30]
  fadd $t42, $t40, $t41
  fadd $t43, $t39, $t42
  sd 56($t1), $t43 S[31]
  addi $t45, $t0, 8
  addi $t47, $t1, 64
  addi $t49, $t2, 64
  addi $t51, $t3, 64
  genu $t52, 1024
  tlt $t54, $t45, $t52
  bro_t<$t54> vadd$1
  bro_f<$t54> vadd$2
  write $g70, $t45
  write $g71, $t47
  write $g72, $t49
  write $g73, $t51
  .bend
```
Vector Add – Block Dataflow Graph

- Read and load instructions gather block inputs
- Write, store, and branch instructions produce block outputs
- Address fanout can present an interesting challenge

(abbreviated graph)
Vector Add – Placed Instructions

- The scheduler analyzes each block dataflow graph.
- It inserts fanout instructions, as needed.
- It places the instructions within the block (they don’t have to be in program order).
- It produces assembly language files.
- An instruction fires when its operands become available (regardless of position).

<table>
<thead>
<tr>
<th>ET0:</th>
<th>ET1:</th>
<th>ET2:</th>
<th>ET3:</th>
</tr>
</thead>
<tbody>
<tr>
<td>ld, mov3,</td>
<td>mov3, ld,</td>
<td>mov, mov3,</td>
<td>mov, mov,</td>
</tr>
<tr>
<td>mov3, mov,</td>
<td>mov, mov,</td>
<td>ld, addi,</td>
<td>mov, mov3,</td>
</tr>
<tr>
<td>mov, mov</td>
<td>mov</td>
<td>addi</td>
<td>ld</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ET4:</th>
<th>ET5:</th>
<th>ET6:</th>
<th>ET7:</th>
</tr>
</thead>
<tbody>
<tr>
<td>ld, ld, ld,</td>
<td>ld, mov3,</td>
<td>ld, fadd,</td>
<td>ld, genu,</td>
</tr>
<tr>
<td>mov3, addi</td>
<td>mov3, fadd,</td>
<td>ld, ld, ld,</td>
<td>addi</td>
</tr>
<tr>
<td></td>
<td>sd</td>
<td>mov</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ET8:</th>
<th>ET9:</th>
<th>ET10:</th>
<th>ET11:</th>
</tr>
</thead>
<tbody>
<tr>
<td>ld, ld, ld,</td>
<td>ld, ld, ld,</td>
<td>ld, fadd,</td>
<td>fadd, fadd,</td>
</tr>
<tr>
<td>mov3, fadd,</td>
<td>mov3, fadd,</td>
<td>fadd, fadd,</td>
<td>sd, tlt,</td>
</tr>
<tr>
<td>sd</td>
<td>sd</td>
<td>sd</td>
<td>bro_t</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ET12:</th>
<th>ET13:</th>
<th>ET14:</th>
<th>ET15:</th>
</tr>
</thead>
<tbody>
<tr>
<td>ld, fadd,</td>
<td>ld, fadd,</td>
<td>fadd, fadd,</td>
<td>fadd, fadd,</td>
</tr>
<tr>
<td>fadd, sd</td>
<td>fadd, fadd,</td>
<td>sd, ld</td>
<td>sd, bro_f</td>
</tr>
</tbody>
</table>
Vector Add – TASL Code

TRIPS Assembly Language (TASL) files are fed to the assembler

Instructions are described using a dataflow notation
Vector Add – Block-Level Execution

- The TRIPS processor can execute up to 8 blocks concurrently
- This diagram shows block latency and throughput for an optimal vector add loop
- It achieves ~7.4 IPC, ~3.2 loads & stores per cycle
Linked List – C Code

```
// type info
struct foo {
    long tag;
    long value;
    struct foo * next;
};
struct foo * head;

void list_add( long key ) {
    struct foo * p = head;

    // search list and increment node while (p)
    while (p) {
        if (p->tag == key)
            {  
                p->value++;
                break;
            }
        p = p->next;
    }

    // ...
}
```

- For a second example, let’s examine something more complex
- We’ll be focusing upon the code in red
- A linked list operation is performed using a while loop and search pointer
- The number of while loop iterations can vary
Linked List – CFG

- The while loop’s control flow graph is shown here
- It consists of four rather small basic blocks
- It’s hard to get anywhere fast with all of those branches
- We’d like to use predication to form a larger program block (or hyperblock)
Here is a TIL representation of the hyperblock:

- Predicate p0 represents the test to determine whether p is null
- Predicate p1 represents the test to determine whether the tag matches the key
- The test instruction that produces p1 is itself predicated upon p0 being false
- That means that there are only three possible predicate outcomes:
  - p0 true, p1 unresolved
  - p0 false, p1 false
  - p0 false, p1 true

This example demonstrates “predicate-oring”, which allows multiple exclusive predicates to be OR’d at the target instruction.

This example demonstrates write and store nullification, which allows block outputs to be cancelled.

Loads are allowed to execute speculatively because exceptions will be predicated downstream.
Linked List – Predicated Loop Unrolling

- With predication, it’s also possible (and helpful) to unroll this loop
- Many variations are possible
- This diagram shows just two iterations within the hyperblock
- The number of block exits and block outputs remains the same as before
- The compiler can use static heuristics or profile data to choose an unrolling factor
- Can sometimes use instruction merging to combine multiple statements into one (in yellow)
Linked List – Hyperblock #2

- Here is some TIL for the unrolled while loop
- The changes from the first version are highlighted (in red)
- Notice the use of predicate-ORing, which allows the block to finish executing early in some cases
- There are now four predicates and five possible outcomes

```
.bbegin block1
  read $t0, $g70           ; key
  read $t1, $g71           ; p
  teqi $p0, $t1, 0
  ld $t3, ($t1)            ; p->tag
  teq_f <$p0> $p1, $t3, $t0
  ld $t4, 16($t1)           ; p->next
  teqi_f <$p1> $p2, $t4, 0
  ld $t5, ($t4)            ; p->next->tag
  teq_f <$p2> $p3, $t5, $t0
  ld $t6, 16($t4)           ; p->next->next
  ; conditional p update
  null_t <$p0,$p1> $t99
  mov_t <$p2,$p3> $t99, $t4
  mov_f <$p3> $t99, $t6
  write $g71, $t99           ; p
  ; conditional p->data update
  ld_t <$p1> $t10, 8($t1)
  ld_f <$p1> $t10, 8($t4)
  null_t <$p0,$p2> $t11
  addi_t <$p1,$p3> $t11, $t10, 1
  null_f <$p3> $t11
  sd 8($t1), $t11            ; p->data
  ; branches
  bro_t <$p0,$p1,$p2,$p3> block2
  bro_f <$p3> block1
.bend
```

<table>
<thead>
<tr>
<th>p0</th>
<th>p1</th>
<th>p2</th>
<th>p3</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td></td>
<td></td>
</tr>
<tr>
<td>F</td>
<td>F</td>
<td>T</td>
<td></td>
</tr>
<tr>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
</tbody>
</table>

June 4, 2005

UT-Austin TRIPS Tutorial 62
Conclusion

- The EDGE ISA allows instructions to be both *fetched* and *executed* out-of-order with minimal book-keeping.
- It’s possible to use loop unrolling and predication to form large blocks.
- The TRIPS predication model allows control flow to be converted into an efficient dataflow representation.
- Operand fanout is more explicit in an EDGE ISA and costs extra instructions.
G-Tile Location and Connections

In the diagram:
- **G-Tile**: Central processor with connections to other tiles.
- **I-Tile**: Input tile connected to **G-Tile**.
- **R-Tile**: Retention tile connected to **G-Tile**.
- **D-Tile**: Data tile with connections to **G-Tile**.

Connections include:
- **HALT_GP** (to EBC)
- **GP_HALT** (from EBC)
- **GDN**, **GRN**, **OCN**, **GSN**, **GCN**, **GSN**, **ESN**, **OPN**

**On-Chip Network (OCN)**
- **SDRAM 0**, **SDRAM 1**, **IRQ**, **EBI**

**Processor Layout**
- **Processor 0** and **Processor 1** with various memory and interface connections.
Major G-Tile Responsibilities

- Orchestrates global control for the core
  - Maps blocks to execution resources

- Instruction fetch control
  - Determines I-cache hits/misses, ITLB hits/misses
  - Initiates I-cache refills
  - Initiates block fetches

- Next-block prediction
  - Generates speculative block addresses

- Block completion detection and commit initiation

- Flush detection and initiation
  - Branch mispredictions, Ld/St dependence violations, exceptions

- Exception reporting
Networks Used by G-Tile

- **OPN** - Operand Network
  - Receive and send branch addresses from/to E-tiles
- **OCN** - On-Chip Network
  - Receive block header from adjacent I-tile
- **GDN** - Global Dispatch Network
  - Send inst. fetch commands to I-tiles and read/write instructions to R-tiles
- **GCN** - Global Control Network
  - Send commit and flush commands to D-tiles, R-tiles and E-tiles
- **GRN** - Global Refill Network
  - Send I-cache refill commands to I-tile slave cache banks
- **GSN** - Global Status Network
  - Receive refill, store, and register write completion status and commit acknowledgements from I-tiles, D-tiles, and R-tiles
- **ESN** - External Store Network
  - Receive external store pending and error status from D-tiles
Global Dispatch Network (GDN)

- Delivers instruction fetch commands to I-tiles
- Dispatches instructions from I-Tiles to R-Tiles/E-Tiles

**Interface Ports**

- **CMD:** None/Fetch/Dispatch
- **ADDR:** Address of block
- **SMASK:** Store mask in block
- **FLAGS:** Block execution flags
- **ID:** Frame identifier
- **INST:** Instruction bits
Global Refill Network (GRN)

Delivers refill commands to I-Tiles

Interface Ports

CMD: None/Refill
RID: Refill buffer identifier
ADDR: Address of block to refill
Global Control Network (GCN)

Interface Ports

CMD: None/Commit/Flush
MASK: Mask of frames for commit/flush

Delivers commit and flush commands to R-Tiles, D-Tiles and E-Tiles
Global Status Network (GSN)

I-tiles: Completion status for pending refills
R-tiles: Completion status register writes, acknowledgement of register commits
D-tiles: Completion status for stores, acknowledgement of store commits

CMD: None/Completed/Committed
ID: Frame/Refill buffer identifier
STATUS: Exception Status
External Store Network (ESN)

Interface Ports

**Pending:** Pending status of external store requests and spills in each thread

**Error:** Errors encountered for external store requests and spills in each thread

Delivers external store pending and error status from D-tiles
Overview of Block Execution

Different states in the execution of a block:

- Start
  - Ready in I-cache
  - Allocate
    - Fetch
      - Dispatch
        - Execute
          - Completed
            - Commit
              - GCN
            - Flushed
              - GCN
            - Deallocated
              - GSN
          - Acks
            - GSN
          - Refill
            - GRN

Major G-Tile Structures

- Control Registers
- Fetch Unit
  - ITLB
  - I-cache dir.
- Refill Unit
  - I-cache
  - MSHRs
- Exit Predictor
- Retire Unit
  - Commit/Flush Ctrl

- GDN/GRN
- GSN
- OCN
- OPN
- GCN
- ESN
Frame Management

Frames allocated from a circular queue

Frames in use

Youngest

Oldest

Non-speculative block
Speculative blocks

Control Speculation
Frame Management

Frames allocated from a circular queue

After a new fetch

Control Speculation
- Non-speculative block
- Speculative blocks
Frame Management

Frames allocated from a circular queue

After a commit

Oldest

Youngest

Oldest

Control Speculation

Non-speculative block

Speculative blocks
Fetch Unit

- Instruction TLB
  - 16 entries
  - 64KB - 1TB supported page size

- I-cache directory
  - 128 entries
    - Each entry corresponds to one cached block
  - Entry Fields
    - Physical tag for the block
    - Store mask for the block
    - Execution flags for the block
  - 2-way set associative
  - LRU replacement
  - Virtually indexed, physically tagged
## Fetch Pipeline

<table>
<thead>
<tr>
<th>Block A</th>
<th>Cycle 0</th>
<th>Cycle 1</th>
<th>Cycle 2</th>
<th>Cycle 3</th>
<th>Cycle 4</th>
<th>Cycle 5</th>
<th>Cycle 6</th>
<th>Cycle 7</th>
<th>Cycle 8</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Predict</td>
<td>Predict</td>
<td>Predict</td>
<td>TLB lookup</td>
<td>Hit/Miss detection</td>
<td>Fetch SLOT 0</td>
<td>Fetch SLOT 1</td>
<td>Fetch SLOT 2</td>
<td>Fetch SLOT 3</td>
</tr>
<tr>
<td></td>
<td>(Stage 0)</td>
<td>(Stage 1)</td>
<td>(Stage 2)</td>
<td>l-cache</td>
<td>Frame allocation</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Address</td>
<td>Select</td>
<td></td>
<td>lookup</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Block B</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Predict</td>
<td>Predict</td>
<td>Predict</td>
<td>Fetch SLOT 0</td>
<td>Fetch SLOT 1</td>
<td>Fetch SLOT 2</td>
<td>Fetch SLOT 3</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>(Stage 0)</td>
<td>(Stage 1)</td>
<td>(Stage 2)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cycle 9</td>
<td>Fetch SLOT 4</td>
<td>Fetch SLOT 5</td>
<td>Fetch SLOT 6</td>
<td>Fetch SLOT 7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cycle 10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cycle 11</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cycle 12</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cycle 13</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cycle 14</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cycle 15</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cycle 16</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cycle 17</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **Predict cycle**
- **Control cycle**
- **Fetch cycle**

June 4, 2005  UT-Austin TRIPS Tutorial
**Refill Pipeline**

<table>
<thead>
<tr>
<th>Block A</th>
<th>Cycle 0</th>
<th>Cycle 1</th>
<th>Cycle 2</th>
<th>Cycle 3</th>
<th>Cycle 4</th>
<th>Cycle 5</th>
<th>Cycle X</th>
<th>Cycle X+1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Predict (Stage 0)</td>
<td>Predict (Stage 1)</td>
<td>Predict (Stage 2)</td>
<td>TLB lookup</td>
<td>l-cache lookup</td>
<td>Hit/Miss detection</td>
<td>Refill</td>
<td>Refill Complete Frame allocation</td>
</tr>
</tbody>
</table>

- Refills stall fetches for the same thread
- Support for up to four outstanding refills – one per thread
## Per-Frame State

### Frame status

<table>
<thead>
<tr>
<th>V</th>
<th>Valid</th>
</tr>
</thead>
<tbody>
<tr>
<td>O</td>
<td>Oldest</td>
</tr>
<tr>
<td>Y</td>
<td>Youngest</td>
</tr>
</tbody>
</table>

### Block execution status

<table>
<thead>
<tr>
<th>RC</th>
<th>Registers completed</th>
</tr>
</thead>
<tbody>
<tr>
<td>SC</td>
<td>Stores completed</td>
</tr>
<tr>
<td>BC</td>
<td>Branch completed</td>
</tr>
<tr>
<td>E</td>
<td>Exception reported</td>
</tr>
<tr>
<td>L</td>
<td>Load violation reported</td>
</tr>
</tbody>
</table>

### Block commit status

<table>
<thead>
<tr>
<th>CS</th>
<th>Commit sent</th>
</tr>
</thead>
<tbody>
<tr>
<td>RCT</td>
<td>Registers committed</td>
</tr>
<tr>
<td>SCT</td>
<td>Stores committed</td>
</tr>
</tbody>
</table>

### Branch prediction status

<table>
<thead>
<tr>
<th>BADDR</th>
<th>Block address</th>
</tr>
</thead>
<tbody>
<tr>
<td>NADDR</td>
<td>Next block address</td>
</tr>
<tr>
<td>NSTATE</td>
<td>Next block address state (predicted or resolved)</td>
</tr>
<tr>
<td>M</td>
<td>Branch misprediction detected</td>
</tr>
<tr>
<td>PC</td>
<td>Prediction completed</td>
</tr>
<tr>
<td>RC</td>
<td>Repair completed</td>
</tr>
<tr>
<td>UC</td>
<td>Update completed</td>
</tr>
</tbody>
</table>

### Commit

\[
\text{Commit} \leftarrow V \& O \& RC \& SC \& BC \\
& \sim E \& \sim L \& \sim CS
\]

### Dealloc

\[
\text{Dealloc} \leftarrow V \& O \& CS \\
& \sim RCT \& \sim SCT \& \sim UC
\]
Block Completion Detection

• Block completed execution if:
  – All register writes received
  – All stores received
  – Branch target received

• Register completion tracked locally at each R-Tile
  – Expected writes at each R-Tile known statically
  – Completion status forwarded from farthest R-Tile to the G-Tile on the GSN

• Store completion reported by D-tile 0
  – All D-Tiles communicate with each other to determine completion
  – Completion status delivered on the GSN by D-tile 0
Commit Pipeline

- Block ready for commit if:
  - Oldest in the thread
  - Completed with no exceptions
  - No load/store dependence violations detected

- Commit operation
  - G-Tile sends commit message for a block on the GCN
  - R-Tiles and D-Tiles perform local commit operations
  - R-Tiles and D-Tiles acknowledge commit using GSN
  - G-Tile updates predictor history for the block
T-Morph Support

- Up to four threads supported in T-morph mode
  - Threads mapped to fixed set of frames

- Per-thread architected registers in the G-tile
  - Thread Control and Status Registers
  - Program Counter (PC)

- Other T-morph support state/logic
  - Per-thread frame management
  - Round-robin thread prioritization for block fetch/commit/flush
Exception Reporting

- Exceptions reported to the G-tile
  - GSN: refill exceptions, execute exceptions delivered to stores, register outputs
  - OPN: execute exceptions delivered to branch targets
  - ESN: errors in external stores

- G-tile halts processor whenever exception occurs
  - Speculative blocks in all threads flushed
  - Oldest blocks in all threads completed and committed
  - All outstanding refills, stores and spills completed
  - Signals EBC to cause external interrupt

- Exception type reported in PSR and TSRs
  - Multiple exceptions may be reported
Control Flow Prediction

ISCA-32 Tutorial

Nitya Ranganathan

Department of Computer Sciences
The University of Texas at Austin
Predictor Responsibilities

• Predict next block address for fetch
  – Predict block exit and exit branch type (branches not seen by predictor)
  – Predict target of exit based on predicted branch type
  – Predictions amortized over each large block
    • Maximum necessary prediction rate is one every 8 cycles

• Speculative update for most-recent histories
  – Keep safe copies around to recover from mispredictions

• Repair predictor state upon a misprediction

• Update the predictor upon block commit
  – Update exits, targets and branch types with retired block state
Each hyperblock may have several predicated exit branches:
- Exit e0 → block H2
- Exit e1 → block H1

Only one branch will fire.

Predictor uses exit IDs to make prediction:
- 8 exit IDs supported
  - e0 → “000”
  - e1 → “001”

Predictor predicts the ID of the exit that will fire:
- implicitly predict several predicates in a path

Exit history formed by concatenating sequence of exit IDs.
High-level Predictor Design

- Index exit predictor with current block address
- Use exit history to predict 1 of 8 exits
- Predict branch type (call/return/branch) for chosen exit
- Index multiple target predictors with block address and predicted exit
- Choose target address based on branch type

```
block address
```

```
Exit Predictor -> predicted exit
Predicted exit -> Branch Type Predictor -> predicted branch type
Predicted branch type -> Target Predictor -> predicted next-block address
```
Predictor Components

- **Exit Predictor (37 Kbits)**
  - 2-level Local Predictor (9 K)
  - 2-level Global Predictor (16 K)
  - 2-level Choice Predictor (12 K)

- **Target Predictor (45 Kbits)**
  - Sequential predictor
  - Branch Target Buffer (20 K)
  - Call Target Buffer (6 K)
  - Return Address Stack (7 K)
  - Branch Type Predictor (12 K)

**Block address**

**predicted exit**

**predicted next-block address**
Prediction using Exit History (linked list example)

- Dynamic block sequence: H1 H1 H1 H2 H1 H1 H1 H2
- Local exit history for block H1: e1 e1 e1 e0 e1 e1 e1 e0
- Local exit history encoding (2 bits): 01 01 01 00 01 01 01 00
  - Use only 2 bits from each 3-bit exit to permit longer histories
- History table entry holds 5 exit IDs
- Prediction table contains exit IDs and hysteresis bits

Exit history register | Prediction table entry | Predictor action
--- | --- | ---
01 01 00 01 01 01 00 | 001 | Predict “001”
01 01 00 01 01 01 00 | 000 | Update history
01 01 00 01 01 01 00 | 000 | Predict “000”
01 01 01 00 01 01 01 00 | 001 | Update history
Prediction Timing

<table>
<thead>
<tr>
<th>Cycle 1</th>
<th>Cycle 2</th>
<th>Cycle 3</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Local</strong></td>
<td>Read local history</td>
<td>Read local exit predictor</td>
</tr>
<tr>
<td><strong>Global</strong></td>
<td>Read global exit predictor (contd.)</td>
<td>Choose final exit</td>
</tr>
<tr>
<td><strong>Choice</strong></td>
<td>Read choice predictor (contd.)</td>
<td></td>
</tr>
<tr>
<td><strong>Btype</strong></td>
<td>Read branch types for all exits (contd.)</td>
<td>Get chosen exit's type, targets</td>
</tr>
<tr>
<td><strong>BTB</strong></td>
<td>Read branch targets for all exits (contd.)</td>
<td>Choose final target</td>
</tr>
<tr>
<td><strong>CTB</strong></td>
<td>Read call, return targets for all exits</td>
<td></td>
</tr>
<tr>
<td><strong>RAS</strong></td>
<td>Read return address from stack (contd.)</td>
<td>Push return address</td>
</tr>
</tbody>
</table>
More Predictor Features

- Speculative state buffering
  - Latest global histories in global and choice history registers
    - Old snapshots of histories stored in history file for repair
  - Latest local history maintained in future file
    - Local exit history table holds committed block history
  - Return address stack state buffering
    - Save top of stack pointer and value before every prediction
    - Repair stack on address and branch type mispredictions

- Call-return
  - Call/return instructions enforce proper control flow
  - Predictor learns to predict return addresses using call-return link stack
    - Update stores return address in link stack corresponding to call stored in CTB
Learning Call-Return Relationships

When a call commits:
- Update Call Target Address in CTB
- Push CTB index onto Link Stack

When a return commits:
- Pop CTB index from Link Stack
- Update Return Address in CTB

During prediction:
- Call ➔ push return address from CTB onto Address Stack
- Return ➔ pop address from Address Stack
Prediction and Speculative Update Timing

Cycle 1

**Local**
- Read local history
  - Read local future file

**Global**
- Read global exit predictor (contd.)
  - Backup global history

**Choice**
- Read choice predictor (contd.)
  - Backup choice history

**Btype**
- Read branch types for all exits (contd.)

**BTB**
- Read branch targets for all exits (contd.)

**CTB**
- Read call, return targets for all exits

**RAS**
- Read return address from stack (contd.)

Cycle 2

**Local**
- Read local exit predictor
  - Update local future file history

**Global**
- Choose final exit
  - Update speculative global history

**Choice**
- Update speculative choice history

**Btype**
- Get chosen exit's type, targets

**BTB**
- Choose final target

**CTB**
- Push return address
  - Backup top of stack

Cycle 3
Optimizations for Better Prediction

- Exit prediction more difficult than branch prediction
  - 1 of 8 vs. 1 of 2
- Fewer exits in block → better predictability

Exit merging

Predicating control flow merges (B8)

Hyperblock
Instruction Fetch

ISCA-32 Tutorial

Haiming Liu

Department of Computer Sciences
The University of Texas at Austin
I-Tile Location and Connections

I-Tile

G-Tile

N-Tile

GDN  GRN  GSN

I-Tile

GDN  GRN  GSN
I-Tile Major Responsibilities

• Instruction Fetch
  – Cache instructions for one row of R-Tiles/E-Tiles
  – Dispatch instructions to R-Tiles/E-Tiles through GDN

• Instruction Refill
  – Receive refill commands from G-Tile on GRN
  – Submit memory read requests to the N-Tile on OCN
  – Keep track of completion status of each ongoing refill
  – Report completion status to G-Tile on the GSN when refill completes

• OCN Access Control
  – Share N-Tile connection between I-Tile and D-Tile/G-Tile
  – Forward OCN transactions between GT/NT or DT/NT
Major I-Tile Structures

Refill buffer
(128bit x 32, 1 read port, 1 write port)

OCN control

CTRL logic

I-$ array
(128bit x 1024 SRAM, 1 R/W port)
Instruction Layout in a Maximal Block

- Header Chunk
- Instruction Chunk0
- Instruction Chunk1
- Instruction Chunk2
- Instruction Chunk3

PC

<table>
<thead>
<tr>
<th>Word 0</th>
<th>Word 1</th>
<th>Word 30</th>
<th>Word 31</th>
</tr>
</thead>
<tbody>
<tr>
<td>H0</td>
<td>Read 0</td>
<td>Write 0</td>
<td></td>
</tr>
<tr>
<td>H1</td>
<td>Read 1</td>
<td>Write 1</td>
<td></td>
</tr>
<tr>
<td>....</td>
<td>....</td>
<td>....</td>
<td></td>
</tr>
<tr>
<td>H30</td>
<td>Read 30</td>
<td>Write 30</td>
<td></td>
</tr>
<tr>
<td>H31</td>
<td>Read 31</td>
<td>Write 31</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Column 0</th>
<th>Column 1</th>
<th>Column 2</th>
<th>Column 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inst. 96</td>
<td>Inst. 97</td>
<td>Inst. 98</td>
<td>Inst. 99</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>Inst. 124</td>
<td>Inst. 125</td>
<td>Inst. 126</td>
<td>Inst. 127</td>
</tr>
</tbody>
</table>
### Instruction Fetch Pipeline

<table>
<thead>
<tr>
<th>Cycle 0</th>
<th>Cycle 1</th>
<th>Cycle 2</th>
<th>Cycle 3</th>
<th>Cycle 4</th>
<th>Cycle 9</th>
</tr>
</thead>
<tbody>
<tr>
<td>ITLB lookup</td>
<td>Hit/Miss detection Frame allocation</td>
<td>Fetch slot 0</td>
<td>Fetch slot 1</td>
<td>Fetch slot 2</td>
<td>......</td>
</tr>
<tr>
<td>I-$ Tag lookup</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Fetch slot 7</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Cycle 3</th>
<th>Cycle 4</th>
<th>Cycle 9</th>
<th>Cycle 10</th>
</tr>
</thead>
<tbody>
<tr>
<td>R/W slot 0 Fetch slot 0</td>
<td>R/W slot 1 Fetch slot 1 Dispatch slot 0</td>
<td>......</td>
<td>R/W slot 6 Fetch slot 6 Dispatch slot 5</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>R/W slot 7 Fetch slot 7 Dispatch slot 6</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Cycle 4</th>
<th>Cycle 9</th>
</tr>
</thead>
<tbody>
<tr>
<td>R/W slot 0 Fetch slot 0</td>
<td>......</td>
</tr>
<tr>
<td></td>
<td>R/W slot 5 Fetch slot 5 Dispatch slot 4</td>
</tr>
</tbody>
</table>
Instruction Dispatch Timing Diagram

- **Issue fetch on cycle x**
  - IT0
  - Receives inst 1 cycle x+4
- RT0
- RT1
- RT2
- RT3

- IT0
  - Receives inst 1 cycle x+5
- DT0
  - Receives inst 1 cycle x+6
- ET0
- ET1
- ET2
- ET3

- IT0
  - Receives inst 1 cycle x+6
- DT1
  - Receives inst 1 cycle x+7
- ET4
- ET5
- ET6
- ET7

- IT0
  - Receives inst 1 cycle x+7
- DT2
  - Receives inst 1 cycle x+8
- ET8
- ET9
- ET10
- ET11

- IT0
  - Receives inst 1 cycle x+8
- DT3
  - Receives inst 1 cycle x+9
- ET12
- ET13
- ET14
- ET15
Instruction Refill

- Receive refill id and block physical address from GRN input
- Request 128 bytes of inst. from memory hierarchy
  - Send out 2 OCN read requests
- Buffer received inst. in refill buffer
- Report refill completion on GSN output when
  - 128 bytes of inst. have been received
  - Completion has been received on GSN input
- Write inst. into SRAM when dispatched from refill buffer
Register Accesses and Operand Routing

ISCA-32 Tutorial

Karu Sankaralingam

Department of Computer Sciences
The University of Texas at Austin
R-Tile and OPN Location and Connections

G-Tile

R-Tile

E-Tile

OPN Router

OPN

OPN Router

OPN

OPN Router

OPN

OPN Router

OPN

OPN Router
Major R-Tile Responsibilities

- Maintain architecture state, commit register writes
  - 512 registers total
  - 128 registers per thread, interleaved across 4 tiles
  - 32 registers/thread * 4 threads = 128 registers per tile
- Process read/write instructions for each block
  - 64 entry Read Queue (RQ), 8 blocks, 8 reads in each
  - 64 entry Write Queue (WQ), 8 blocks, 8 writes in each
- Forward inter-block register values
- Detect per-block and per-tile register write completion
- Detect exceptions that flow to register writes
Major Structures in the R-tile

- Write Queue
- Read Queue
- Committed Register File
- OPN Router
- Commit
- Decode
- GSN
- GCN, GDN
Pipeline diagrams: Register Read and Write

**Cycle 0**
- Read dispatch
- Read Queue

**Cycle 1**
- Read issue

**Cycle 2**
- Read wakeup (WQ search)
- Write Queue
- Write issue
- Write forward (RQ search)

**Cycle 3**
- OPN inject
Register Forwarding: RQ and WQ contents

Read Queue

Write Queue

<table>
<thead>
<tr>
<th>Valid</th>
<th>Status</th>
<th>Reg #</th>
<th>Target 1</th>
<th>Target 2</th>
<th>WQ wait</th>
<th>WQ pointer</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3</td>
<td>5</td>
<td>8</td>
<td>8</td>
<td>1</td>
<td>6</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Valid</th>
<th>Status</th>
<th>Reg #</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>8</td>
<td>5</td>
<td>64</td>
</tr>
</tbody>
</table>
Register Forwarding Example

**B0:**
R[0], W[0]: dispatch
R[0]: search WQ, read from CRF

W[0] receives value at WQ
Wakes up R[0] in B1
Forwards 45 to N[0]

**B1:**
R[0], R[4], W[0]: dispatch
R[0]: search WQ, wait
R[4]: search WQ, read from CRF

N[0]…N[3] execute
W[0] receives value at WQ
### Block completion: Commit unit

<table>
<thead>
<tr>
<th>Block Status</th>
<th>Valid</th>
<th>All writes received</th>
<th>Right neighbor completed</th>
<th>Completion sent</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Cycle Status

<table>
<thead>
<tr>
<th>Cycle 0</th>
<th>RT0</th>
<th>RT1</th>
<th>RT2</th>
<th>RT3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 0 0</td>
<td></td>
<td>1 0 0 0 0</td>
<td></td>
<td>1 1 1 1 0</td>
</tr>
<tr>
<td>Cycle 1</td>
<td></td>
<td>1 0 1 0</td>
<td></td>
<td>1 1 1 1 1</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cycle 5</td>
<td>1 1 1 0 0</td>
<td></td>
<td>1 1 1 1 0</td>
<td></td>
</tr>
<tr>
<td>Cycle 6</td>
<td>1 1 1 1 0</td>
<td>GSN</td>
<td>1 1 1 1 1</td>
<td></td>
</tr>
<tr>
<td>Cycle 7</td>
<td>1 1 1 1 1</td>
<td></td>
<td>1 1 1 1 1</td>
<td></td>
</tr>
</tbody>
</table>

- **GSN** indicates a state change.
Major OPN Responsibilities and Attributes

- Wormhole-routed network connecting 25 tiles
- 4 entry FIFO flit buffers in each direction
- Fixed packet length; 2 flits
- On-off flow control: 1-bit hold signal
- Dimension order routing (Y-X)
**OPN Packet Format**

- One control packet followed by one data packet in next cycle
- Dedicated wires for control/data
- Control packet
  - Contains routing information
  - Enables early wakeup and fast bypass in ET

### Control Packet

<table>
<thead>
<tr>
<th>Field</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
</tr>
</tbody>
</table>

- **Valid**
- **Type**
- **Frame ID**
- **Destination Node**
- **Destination Index**
- **Source Node**
- **Source Index**

### Control Packet Types

- 0 - Generic transfer or reply
- 1 - Load
- 2 - Store
- 4 - PC read
- 5 - PC write
- 14 - Special register read
- 15 - Special register write

### Data Packet

<table>
<thead>
<tr>
<th>Field</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>64</td>
<td>40</td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
</tbody>
</table>

- **Valid**
- **Type**
- **Value**
- **Address**
- **Operation**

### Data Packet Operations

- lb, lh, lw, ld
- sb, sh, sw, sd

### Data Packet Types

- Normal, Null, Exception
Operand Network Properties

• Directly integrated into processor pipeline
  – Hold signal stalls tile pipelines (back pressure flow control)
  – FIFOs flushed with processor flush wave

• Deadlock avoidance
  – On-off flow control
  – Dimension order routing
  – Guaranteed buffer location for every packet

• Tailored for operands
  – [Taylor et al. 2003, Sankaralingam et al. 2003]
Issue Logic and Execution

ISCA-32 Tutorial

Premkishore Shivakumar

Department of Computer Sciences
The University of Texas at Austin
E-Tile Location and Connections

- E-Tile
- E/D-Tile
- GDN
- GCN
- OPN

Diagram showing connections and locations of E-Tiles, E/D-Tiles, and GDNs-GCNs-OPNs.
Networks Used by E-Tile

- **OPN** - Operand Network
  - Transmit register values from/to R-tiles
  - Transmit operands from/to other E-tiles
  - Transmit load/store packets from/to D-tiles

- **GDN** - Global Dispatch Network
  - Receive instructions from I-tile, and the dispatch commands from G-tile
    - Through a D-tile or a neighboring E-tile

- **GCN** - Global Control Network
  - Receive commit, flush commands originating at G-tile
    - From a D-tile or a neighboring E-tile
Major E-Tile Responsibilities

- **Buffer Instructions and Operands**
  - Local reservation stations for instruction and operand management

- **Instruction Wakeup and Select**

- **Operand Bypass**
  - Local operand bypass to enable back-to-back local dependent instruction execution
  - Remote operand bypass with one extra cycle of OPN forwarding latency per hop

- **Instruction Target Delivery**
  - Instructions with multiple local and/or remote targets
  - Successive local or remote targets in consecutive cycles
  - Concurrently deliver a remote and a local target

- **Predication Support**
  - Predicate-based conditional execution
  - Hardware predicate OR-ing

- **Exception and Null**
  - Exception generation: divide by zero exception, NULL instruction
  - Exception and null tokens propagated in dataflow manner to register writes, stores

- **Flush and Commit**
  - Clear block state (pipeline registers, status buffer etc.)
E-Tile High Level Block Diagram

- **GDN**
- **Status Buffer**
- **Instruction Buffer**
- **Operand Buffer** (64 entries, 8 frames, 8 slots per frame)
- **EXECUTION UNITS**
  - INT ALU
  - INT MULT
  - INT DIV
  - FP ADD/SUB
  - FP MULT
  - FP CMP
  - FP CVT ($D\to S$, $S\to D$, $I\to D$, $D\to I$)

**Routing Diagram**:
- Router
- OPN
- OP1
- OP2
- ALU
- INST RUCT ION OP1
- OP2
- Router
- INST RUCT ION OP2

June 4, 2005

UT-Austin TRIPS Tutorial
## Functional Unit Latencies in the E-Tile

<table>
<thead>
<tr>
<th>Functional Unit</th>
<th>Latency</th>
<th>Implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer ALU</td>
<td>1</td>
<td>Synopsys IP</td>
</tr>
<tr>
<td>Integer Multiplier</td>
<td>3 (Pipelined)</td>
<td>Synopsys IP</td>
</tr>
<tr>
<td>Integer Divider</td>
<td>24 (Iterative)</td>
<td>Synopsys IP</td>
</tr>
<tr>
<td>FP Add / Sub</td>
<td>4 (Pipelined)</td>
<td>Synopsys IP</td>
</tr>
<tr>
<td>FP Multiplier</td>
<td>4 (Pipelined)</td>
<td>Synopsys IP</td>
</tr>
<tr>
<td>FP Comparator</td>
<td>2 (Pipelined)</td>
<td>Synopsys IP</td>
</tr>
<tr>
<td>FPDtoI Convert</td>
<td>2 (Pipelined)</td>
<td>Synopsys IP</td>
</tr>
<tr>
<td>FP ItoD Convert</td>
<td>3 (Pipelined)</td>
<td>Synopsys IP</td>
</tr>
<tr>
<td>FP StoD Convert</td>
<td>2 (Pipelined)</td>
<td>UT TRIPS IP</td>
</tr>
<tr>
<td>FP DtoS Convert</td>
<td>3 (Pipelined)</td>
<td>UT TRIPS IP</td>
</tr>
</tbody>
</table>
Reservation stations at one E-Tile

- Instructions can execute out-of-order
- Issue prioritization among ready instructions
- Scheduler assigns slots for block instructions
- Slot based instruction priority within a block
Issue Prioritization Among Ready Instructions

Reservation stations at one E-Tile

D-MORPH

B2
B1
B0
B7
B3

Older Block First
B1 before B2

Smaller Instruction
Slot First
B3.I0 before B3.I2

T-MORPH

T2
T1
T0
T3

Three-level Select Priority:

- Round robin priority across threads
- Older block first priority in a thread
- Smaller instruction slot first within a block
Two Phase Instruction Selection

- Priority Encoder uses *Issue Prioritization Policy*
- Wakeup: Back-to-back operand delivery, and two bypassed operands in the same cycle
Basic Operand Local & Remote Bypass

Sample Dependence Graph

Instruction Physical Placement

Local Bypass

- Execute `mov`
- Send `mov` Local Bypass Data Pkt
- Wakeup, Select `sub`

Remote Bypass

- Execute `sub`
- Send `sub` OPN Ctrl Pkt from ET0 to ET1
- Wakeup, Select `add`

Local Bypass

- Receive `mov` Local Bypass Data Pkt

Remote Bypass (extra OPN cycle)

- Receive `sub` OPN Data Pkt
- Execute `add`
Bypass: Sources of Pipeline Bubbles

- **OPN stall due to network congestion**
  - Local Bypass:
    - `mov` to `sub`
  - Remote Bypass:
    - `sub` to `add`
  - Cycle 1:
    - Execute `mov`
    - Send `mov` Local Bypass Ctrl Pkt
    - Wakeup, Select `sub`
  - Cycle 2:
    - Receive `mov` Local Bypass Data Pkt
    - Execute `sub`
    - Send `sub` OPN Ctrl Pkt from ET0 to ET1
  - Cycle X:
    - Send `sub` OPN Data Pkt from ET0 to ET1
    - Wakeup, Select `add`
  - Cycle X+1:
    - Receive `sub` OPN Data Pkt
    - Execute `add`
  - **OPN Stall**

- **Bypassed operand data is a non-enabling predicate**
  - Detected in execute stage after receiving bypassed data => Issue slot wasted
  - Similar pipeline bubble due to invalid bypassed OPN data packet

---

June 4, 2005  UT-Austin TRIPS Tutorial
Target Delivery: Multiple Local Targets

• Successive local targets delivered in consecutive cycles
• Successive remote targets also delivered in consecutive cycles (extra OPN forward cycle)
Concurrent Local & Remote Target Delivery

- Instruction local and remote target can be sent concurrently
- \textit{mov3}, \textit{mov4} instructions have 3 and 4 targets respectively, used to fanout operands to multiple children

\begin{table}
\begin{tabular}{|c|c|}
\hline
\textbf{# cycles to emit targets (Local/Remote)} & \\
\hline
LL & 2 \\
LR & 1 \\
RR & 2 \\
LLL & 3 \\
LLR & 2 \\
LRR & 2 \\
RRR & 3 \\
LLLL & 4 \\
LLL & 3 \\
LLRR & 2 \\
LRRR & 3 \\
RRRR & 4 \\
\hline
\end{tabular}
\end{table}
Primary Memory System

ISCA-32 Tutorial

Simha Sethumadhavan

Department of Computer Sciences
The University of Texas at Austin
D-Tile Location and Connections

I-Tile

G-Tile

E-Tile

D-Tile

On-Chip Network (OCN)

SDRAM 0

IRQ

EBI

Processor 0

DMA

SDC

ERC

D M M N I D E E E E

D M M N I D E E E E

D M M N I D E E E E

D M M N I D E E E E

D M M N I D E E E E

D M M N I D E E E E

D M M N I D E E E E

D M M N I D E E E E

Processor 1

SDRAM 1

C2C (x4)
Major D-Tile Responsibilities

- Provide D-cache access for arriving loads and stores
- Perform address translation with DTLB
- Handle cache misses with MSHRs
- Perform dynamic memory disambiguation with load/store queues
- Perform dependence prediction for aggressive load/store issue
- Detect per-block store completion
- Write stores to caches/memory upon commit
- Store merging on L1 cache misses
Primary Memory Latencies

- Data cache is interleaved on a cache line basis (64 bytes)

- Instruction placement affects load latency
  - Best case load-to-use latency: 5 cycles
  - Worst case load-to-use latency: 17 cycles
  - Loads favored in E-tile column adjacent to the D-tiles
  - Deciding row placement is difficult

- Memory access latency: Three components
  - From E-Tile (load) to D-Tile
  - D-Tile access (this talk)
  - From D-Tile back to E-Tile (load target)
Major Structures and Sizes in the D-Tile

- **Caches**
  - 8KB, 2-way, 1R and 1W port
  - 64 byte lines, LRU, Virtually indexed, physically tagged,
  - Write-back, Write-no-allocate

- **DTLB**
  - 16 entries, 64KB to 1TB pages

- **Dependence Predictor**
  - 1024 bits, PC based hash function

- **Load Store Queues**
  - 256 entries, 32 loads/stores per block, single ported

- **Miss Handling**
  - 16 outstanding load misses to 4 cache lines
  - Support for merging stores up to one full cache line
## Load Execution Scenarios

<table>
<thead>
<tr>
<th>TLB</th>
<th>Dep. Pr</th>
<th>LSQ</th>
<th>Cache</th>
<th>Response</th>
</tr>
</thead>
<tbody>
<tr>
<td>Miss</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>Report TLB Exception</td>
</tr>
<tr>
<td>Hit</td>
<td>Wait (Hit)</td>
<td>-</td>
<td>-</td>
<td>Defer load until all prior stores are received. Non deterministic latency</td>
</tr>
<tr>
<td>Hit</td>
<td>Miss</td>
<td>Miss</td>
<td>Hit</td>
<td>Forward data from L1 Cache</td>
</tr>
<tr>
<td>Hit</td>
<td>Miss</td>
<td>Miss</td>
<td>Miss</td>
<td>Forward data from L2 Cache (or memory) Issue cache fill request (if cacheable)</td>
</tr>
<tr>
<td>Hit</td>
<td>Miss</td>
<td>Hit</td>
<td>Hit</td>
<td>Forward data from LSQ</td>
</tr>
<tr>
<td>Hit</td>
<td>Miss</td>
<td>Hit</td>
<td>Miss</td>
<td>Forward data from LSQ Issue cache fill request (if cacheable)</td>
</tr>
</tbody>
</table>
Load Pipeline

- Common case: 2 cycle load hit latency

- For LSQ hits, load latency varies between 4 to $n+3$ cycles, where $n$ is size of the load in bytes
  - Stalls all pipelines except forwarding stages (cycles 2 and 3)
Dependence Prediction

• Prediction at memory side (not at issue)
  – New invention because of distributed execution

• Properties
  – PC-based, updated on violation
  – Only loads access the predictor, loads predicted dependent are deferred
  – Deferred loads are woken up when all previous stores have arrived
  – Reset periodically (flash clear) based on number of blocks committed
  – Dependence speculation is configurable: Override, Serial, Regular

![Diagram of Dependence Prediction](image-url)
Deferred Load Pipeline

<table>
<thead>
<tr>
<th>Cycle 1</th>
<th>Cycle 2</th>
<th>Cycle X</th>
<th>Cycle X+1</th>
<th>Cycle X+2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Access Cache, TLB, DP, and LSQ</td>
<td>Mark Load Waiting</td>
<td>All Prior Stores Arrived</td>
<td>Prepare Load for Replay</td>
<td>Re-execute as if new load</td>
</tr>
</tbody>
</table>

- **Dependence Predictor Hit**
- **Deferred Load Processing (a.k.a. Replay pipe)**

- Deferred loads are woken up when all prior stores have arrived at D-tiles
  - Loads between two consecutive stores can be woken up out-of-order
- Deferred loads are re-injected into the main pipeline, as if they were new load
  - During this phase, loads get the value from dependent stores, if any
- Pipeline stall
  - When deferred load in cycle X+2 cannot be re-injected into the main pipeline
  - Load cannot be prepared in cycle X+1 because of resource conflicts in the LSQ
At block dispatch, each DT receives and records information on stores in the dispatched block.

Each DT records store counts and store mask (in the LSQ). Store mask (32 bits) identifies the store instructions in a block.

On store arrival, the received store count is updated at local tile and store arrival is passed on to other tiles (next slide).

June 4, 2005 UT-Austin TRIPS Tutorial
Block Store Completion Detection

- Need to share store arrival information between D-tiles
  - Block completion
  - Waking up deferred loads
- Data Status Network (DSN) is used to communicate store arrival information
  - Multi-hop broadcast network
  - Each link carries store FrameID and store LSID
- DSN Operation
  - Store arrival initiates transfer on DSN
  - All tiles know about the store arrival after 3 cycles
- GSN Operation
  - Completion can be detected at any D-tile but special processing is done in D-tile0 before sending it to G-tile
  - Messages sent on GSN
Store Commit Pipeline

- Stores are committed in LSID and block order at each D-tile
  - Written into L1 cache or bypassed on a cache miss
  - One store committed per D-tile per cycle
    - Up to four stores total can be committed every cycle, possibly out-of-order
    - T-morph: Store commits from different threads are not interleaved
- On a cache miss, stores are inserted into the merge buffer
- Commit pipe stalls when cache ports are busy
  - Fills from missed loads take up cache write ports
Load and Store Miss Handling

- **Load Miss Operation**
  - Cycle 1: Miss determination
  - Cycle 2: MSHR allocation and merging
  - Cycle 3: Requests sent out on OCN, if available

- **Load Miss Return**
  - Cycle 1: Wake up all loads waiting on incoming data
  - Cycle 2: Select one load; data for the load miss starts arriving
  - Cycle 3: Prepare load and re-inject into main pipeline, if unblocked

- **Store Merging**
  - Attempts to merge consecutive writes to same line
  - Snoops on all incoming load misses for coherency

- **Resources are shared across threads**
Coherence checks
- Load and store misses to same cache line

D-Tile Pipelines Block Diagram

- Load input from E-Tile
  - Replay load pipeline
  - Missed load pipeline
  - Line fill pipeline
  - Store commit pipeline

- Cache + LSQ Load Port
  - Arb
  - LD X
  - BYPASSES
  - ST X
  - Store hit
  - Store miss pipe

- Load output to E-Tile
  - Miss handling
  - Spill Mgmt
  - Arb
  - CC
  - L2 Req. Channel

- Costly resources are shared e.g. cache and LSQ ports, L2 bandwidth

- Unfinished store bypassing to load in previous stage. Bypassing => fewer stalls => higher throughput

June 4, 2005
UT-Austin TRIPS Tutorial
Memory System Design Challenges

- Optimizing and reducing the size of the LSQ
  - Prototype LSQ size exceeds data cache array size

- Further optimization of memory-side dependence prediction
  - Prior state of the art is done at instruction dispatch/issue time

- Scaling and decentralizing the Data Status Network (DSN)
D-Tile Roadmap 😊
Secondary Memory System

ISCA-32 Tutorial

Changkyu Kim

Department of Computer Sciences
The University of Texas at Austin
M-Tile Network Attributes

- Total 1MB on-chip memory capacity
  - Array of 16 64KB M-Tiles
  - Connected by specialized on-chip network (OCN)
- High bandwidth
  - A peak of 16 cache accesses per cycle
  - 68 GB/sec to the two processor cores
- Low latency for future technologies
  - Non-uniform (NUCA) level-2 cache
- Configuration
  - Scratchpad / Level-2 cache modes on a per-bank basis
  - Configurable address mapping
    - Shared L2 cache / Private L2 cache
Non-Uniform Level-2 Cache (NUCA)

- Non-uniform access latency
  - Closer bank can be accessed faster
- Designed for wire-dominated future technologies
  - No global interconnect
- Connected by switched network
  - Can fit large number of banks with small area overhead
- Allows thousands of in-flight transactions
  - Network capacity = 2100 flits
  - Maximize throughput
## Multiple Modes in TRIPS Secondary Memory

<table>
<thead>
<tr>
<th>Mode Description</th>
<th>Diagram</th>
</tr>
</thead>
<tbody>
<tr>
<td>All shared level-2 caches</td>
<td>![Diagram 1]</td>
</tr>
<tr>
<td>Per-processor level-2 caches</td>
<td>![Diagram 2]</td>
</tr>
<tr>
<td>All scratchpad memories</td>
<td>![Diagram 3]</td>
</tr>
<tr>
<td>Half level-2 caches, half scratchpad memories</td>
<td>![Diagram 4]</td>
</tr>
</tbody>
</table>

- **L2**: M-Tile configured as a level-2 Cache
- **S**: M-Tile configured as a scratchpad memory

---

**On-Chip Network (OCN)**

- **DMA**: Direct Memory Access
- **SDC**: Shared Data Cache
- **EBC**: External Buffer Cache
- **N**: Not Connected

---

June 4, 2005

UT-Austin TRIPS Tutorial
Major Structures in the M-Tile

- 8-way set associative cache
- 64B Line size
- 1 MSHR entry (Total 16 entries in the whole cache)
- Configurability (tag on/off)
- 1B ~ 64B Read / Write / Swap transactions support
- Error checking and handling

OCN ROUTER

OCN Incoming Interface

MT Decoder

Tag Array

MSHR

Data Array

OCN Outgoing Interface

MT Encoder

• 8-way set associative cache
• 64B Line size
• 1 MSHR entry (Total 16 entries in the whole cache)
• Configurability (tag on/off)
• 1B ~ 64B Read / Write / Swap transactions support
• Error checking and handling
Back-to-Back Read Pipeline

7-cycle latency for transferring all 64B data

4-cycle latency for sending a reply header

<table>
<thead>
<tr>
<th>Cycle 0</th>
<th>Cycle 1</th>
<th>Cycle 2</th>
<th>Cycle 3</th>
<th>Cycle 4</th>
<th>Cycle 5</th>
<th>Cycle 6</th>
<th>Cycle 7</th>
<th>Cycle 8</th>
<th>Cycle 9</th>
</tr>
</thead>
<tbody>
<tr>
<td>Request Arrives at Input FIFO</td>
<td>Tag Access Data Access 1</td>
<td>Data Access 2</td>
<td>Send reply header</td>
<td>Send 1&lt;sup&gt;st&lt;/sup&gt; 16B of reply data</td>
<td>Send 2&lt;sup&gt;nd&lt;/sup&gt; 16B of reply data</td>
<td>Send 3&lt;sup&gt;rd&lt;/sup&gt; 16B of reply data</td>
<td>Send 4&lt;sup&gt;th&lt;/sup&gt; 16B of reply data</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Seamless Data Transfer (6.8GB/sec)
Read Miss Pipeline

4-cycle latency for sending a fill request to SDC

<table>
<thead>
<tr>
<th>Cycle 0</th>
<th>Cycle 1</th>
<th>Cycle 2</th>
<th>Cycle 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Request Arries at Input FIFO</td>
<td>Tag Access</td>
<td>MSHR Allocation</td>
<td>Send outgoing read miss request</td>
</tr>
</tbody>
</table>

Access the main memory

9-cycle latency for filling 64B data and sending a reply header

<table>
<thead>
<tr>
<th>Cycle 0</th>
<th>Cycle 1</th>
<th>Cycle 2</th>
<th>Cycle 3</th>
<th>Cycle 4</th>
<th>Cycle 5</th>
<th>Cycle 6</th>
<th>Cycle 7</th>
<th>Cycle 8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reply Arrives at Input FIFO</td>
<td>Tag Access</td>
<td>Fill 1st 16B of reply data</td>
<td>Fill 1st 16B of reply data</td>
<td>Fill 1st 16B of reply data</td>
<td>Fill 1st 16B of reply data</td>
<td>MSHR Access</td>
<td>Generate reply header</td>
<td>Send 1st reply header</td>
</tr>
</tbody>
</table>

June 4, 2005 UT-Austin TRIPS Tutorial
Chip- and System-Level Networks and I/O

ISCA-32 Tutorial

Paul Gratz

Department of Computer Sciences
The University of Texas at Austin
OCN Location and Connections

---

[Diagram showing OCN connections and locations]
Goals and Requirements of the OCN

• Fabric for interconnection of processors, memory and I/O
• Handle multiple types of traffic:
  – Routing of L1 and L2 cache misses
  – Memory requests
  – Configuration and data register reads/writes by external host
  – Inter-chip communication
• OCN Network Attributes
  – Flexible mapping of resources
  – Deadlock free operation
  – Performance (16 Client Links – 6.8GB/sec per link)
  – Guaranteed delivery of request and reply
OCN Protocol

- Flexible mapping of resources
  - System address mapping tables are runtime re-configurable to allow for runtime re-mapping between L2 cache banks and scratchpad memory banks

- Deadlock Avoidance
  - Dimension order routed network (Y-X)
  - Four virtual channels used to avoid deadlock

- Performance
  - Links are 128 bit wide in each direction
  - Credit-based flow control
  - One cycle per hop plus cycle at network insertion
  - 68 GB/sec at processor interfaces
**N-Tile Design**

- **Goals**
  - Router for OCN packets
  - System address to network address translation
    - Configurable L2 network addresses
  - One cycle per hop latency
- **Attributes**
  - Composed of OCN router with translation tables on local input
  - Portions of the system address index into translation tables
  - Translation tables configured through memory mapped registers
  - Error back on invalid addresses
I/O Tiles

- **EBC Tile**
  - Provides interface between board-level PowerPC 440 GP chip and the TRIPS chip
  - Forwards interrupts to the 440 GP’s service routines

- **SDC Tile**
  - Contains IBM SDRAM Memory Controller
  - Error detection, swap support and address remapping added

- **DMA Tile**
  - Supports multi-block and scatter/gather transfers
  - Buffered to optimize traffic

- **C2C Tile**
C2C Tile Design Considerations

• C2C Goals
  – Extend OCN fabric across multiple chips
  – Provide flexible extensibility

• C2C Attributes
  – C2C network protocol similar to OCN except:
    • 64 bit wide
    • 2 virtual channels
  – Operates at up to 266MHz
  – Clock speed configurable at boot
  – Source synchronous off-chip links
  – Multiple clock domains with resynchronization
C2C Network

- **Latencies:**
  - L2 Hit Latency: 9 – 27 cycles
  - L2 Miss Latency: 26 – 44 cycles (+SDRAM Access time)
  - L2 Hit Latency (adjacent chip): 33 – 63 cycles
  - L2 Miss Latency (adjacent chip): 50 – 78 cycles (+SDRAM)
  - Each additional C2C hop adds 8 cycles in each direction

- **Bandwidth:**
  - OCN Link: 6.8 GB/sec
  - One processor to OCN: 34 GB/sec
  - Both processors to OCN: 68 GB/sec
  - Both SDRAMs: 2.2 GB/sec
  - C2C Link: 1.7 GB/sec
Compiler Overview

ISCA-32 Tutorial

Kathryn McKinley

Department of Computer Sciences
The University of Texas at Austin
Compiler Challenges

“Don’t put off to runtime what you can do at compile time.” Norm Jouppi, May 2005.

• Block constraints & compiler scheduling
  – Makes the hardware simpler
  – Shifts work to compile time
Generating Correct, High Performance Code

• TRIPS specific compiler challenges
  – EDGE execution model
  – What’s a TRIPS block?

• The compiler’s job
  – Correct code
    • Satisfying TRIPS block constraints
  – Optimized code
    • Filling up TRIPS blocks with useful instructions
Block Execution

1 - 128 instruction DFG

Reg. banks

32 read instructions

Reg. banks

32 write instructions

Address+targets sent to memory, data returned to target

32 loads

PC read

32 stores

terminating branch

PC

Memory

Memory

June 4, 2005

UT-Austin TRIPS Tutorial
TRIPS Block Constraints

• **Fixed size**: 128 instructions
  – Padded with no-ops if needed
  – Simpler hardware: instruction size, I-cache design, global control

• **Registers**: 8 reads and writes to each of 4 banks (in addition to 128 instructions)
  – Reduced buffering
  – Read/write instruction overhead, register file size

• **Block Termination**: all stores and writes execute, one branch
  – Simplifies hardware logic for detecting block completion

• **Fixed output**: 32 load or store queue IDs, one branch
  – LSQ size, instruction size
Outline

• Overview of TRIPS compiler components
• How hard is it to meet TRIPS block constraints?
  – Compiler algorithms to enforce them
  – Evaluation
    • Question: How often does the compiler under-fill a block to meet a constraint?
    • Answer: Almost never
• Optimization
  – High quality TRIPS blocks
  – Scheduling for TRIPS
  – Cache bank optimization
• Preliminary evaluation
Compiler Phases (2)

- Code Generation
  - Legal Block Analysis
  - Block Splitting
  - Reverse If-conversion
  - Register Allocation
  - Spill?

- Hyperblock flow graph after code generation

- Backend Optimizations
  - LSQ ID Assignment
  - Store Nullification
  - Peephole
  - Instruction Promotion

- Hyperblock flow graph after splitting and optimization

- Scheduled and assembled code

- Scheduler
  - Insert Moves
  - Places Instructions
  - Generates Assembly
High Quality TRIPS Blocks

• Full
• High ratio of committed to total instructions
  – Dependence height is not an issue
• Minimize branch misprediction
• Meet correctness constraints
TRIPS Block Constraints

- **Fixed size**: at most, 128 instructions
  - Easy to satisfy by the compiler, but….
  - Good performance requires blocks full of useful instructions
- Registers: 8 reads and writes to each of 4 banks (plus 128 instructions)
- Block Termination
- Fixed output: at most, 32 load or store queue IDs
03: Basic Blocks as TRIPS Blocks

- O3: every basic block is a TRIPS block
- Simple, but not high performance
O4: Hammock Hyperblocks as TRIPS Blocks

Two possible hammock hyperblocks for this graph
## Static TRIPS Block Instruction Counts

### Static Averages

<table>
<thead>
<tr>
<th></th>
<th>03</th>
<th>04</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>min</strong></td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td><strong>max</strong></td>
<td>42</td>
<td>42</td>
</tr>
<tr>
<td><strong>mean</strong></td>
<td>16</td>
<td>18</td>
</tr>
</tbody>
</table>

- **SPEC2000**
  - min: 7
  - max: 42
  - mean: 16
  - min: 8
  - max: 42
  - mean: 18

- **EEMBC**
  - min: 7
  - max: 31
  - mean: 16
  - min: 9
  - max: 67
  - mean: 22
• 15% average reduction in number of TRIPS blocks between O3 and O4
• Results missing sixtrack, perl, and gcc
Enforcing Block Size

Too Big?
1. Unpredicated
   - cut
2. Predicated
   - Reverse if-conversion

Initial Hammock Hyperblock  After Reverse If-Conversion
SPEC > 128 Instructions

- At O4 an average of 0.9% TRIPS blocks > 128 insts. (max 6.6%)
In Progress: Tail Duplication

- Tail duplicate A
- Increases block size
- Hot long paths

After Reverse If-Conversion

After Tail Duplication
while (*pp != '\0')
{
    if (*pp == '.') {
        break;
    }
    pp++;
}

• Add support for complex control flow (i.e. breaks/continues)
• An example: the number of TRIPS blocks in ammp_1 microbenchmark
  • 38 / 27 / 12 with O3 / 04 (current) / new
While Loop Unrolling

```
while (*pp != '\0')
{
    if (*pp == '.') {
        break;
    }
    pp++;
}
```

- Supported on architectures with predication
- Implemented, but needs improved hyperblock generation
TRIPS Block Constraints

- Fixed size: at most, 128 instructions
- **Registers**: 8 reads and writes to each of 4 banks (plus 128 instructions)
  - Simple linear scan priority order allocator
- Block Termination
- Fixed output: at most, 32 load or store queue IDs
Linear Scan Register Allocation

- 128 registers 32 per 4 banks
- Hyperblocks as large instructions
- Computes liveness over hyperblocks
- Memory savings vs. instructions
  - Fewer hyperblocks than instructions
  - Allocator ignores local variables
- Spills only 5 SPEC2000 vs 18 Alpha
  - gcc: 4290 Alpha - 211 TRIPS
  - sixtrack: 3510 Alpha - 165 TRIPS
  - mesa: 2048 Alpha - 207 TRIPS
  - apsi: 920 Alpha - 7 TRIPS
  - applu: 14 Alpha - 19 TRIPS
- Average spill: 1 store, 2-7 loads
- Spills are less than 0.5% of all dynamic loads/stores
SPEC > 32 (8x4 banks) Registers

- At O4 an average of 0.3% TRIPS blocks have > 32 register reads
- At O4 an average of 0.2% TRIPS blocks have > 32 register writes
SPEC > 8 Registers per Bank

- At O4 an average of 0.12% TRIPS blocks have > 8 read banks
- At O4 an average of 0.07% TRIPS blocks have > 8 write banks
TRIPS Block Constraints

- Fixed size: at most, 128 instructions
- Control flow: 8 branches (soft)
- Registers: 8 reads and writes to each of 4 banks (plus 128 instructions)

**Block Termination**

- Fixed output: at most, 32 load or store queue IDs
Block Termination

• One branch fires
• All writes complete
  – Write nullification
• All stores complete
  – Store nullification
Register Writes

Before SSA

write 4

def 4

write 4

After SSA

read 4

phi 4,6

write 4

def 6

• How do we guarantee all paths produce a value?
  – Insert corresponding read instructions for all write instructions
  – Transform into SSA form
  – Out of SSA to add moves
• Could nullify writes
  – Increases block size
Store Nullification

How do you insert nulls for this? Two possibilities. Which is better?

- Dummy stores
  - Wastes space
  - Easier to schedule?

- Code motion
  - Better resource utilization
SPEC Store Nullification

- Nullification is 1.2% of instructions for 177.mesa (0.8% dummy stores)
- Is it significant?
TRIPS Block Constraints

- Fixed size: at most, 128 instructions
- Control flow: 8 branches (soft)
- Registers: 8 reads and writes to each of 4 banks (plus 128 instructions)
- Block Termination: each store issues once
- **Fixed output**: at most, 32 load or store queue IDs
Load/Store ID Assignment

- Ensures Memory Consistency
- Simplifies Hardware
- Overlapping LSQ IDs
  - Increases number of load/store instructions in a TRIPS block
  - But a load and store cannot share the same ID

Before Assignment

LD
SD

After Assignment

LD [0]
LD [0]
SD [1]
SPEC > 32 LSQ IDs

- At O4 an average of 0.3% TRIPS blocks have > 32 LSQ IDs
Correct Compilation of TRIPS Blocks

- Some new compiler challenges
- New, but straightforward algorithms
- Does not limit compiler effectiveness (so far!)
Code Optimization

ISCA-32 Tutorial

Kathryn McKinley

Department of Computer Sciences
The University of Texas at Austin
Outline

• Overview of TRIPS Compiler components
• Are TRIPS block constraints onerous?
  – Compiler algorithms that enforce them
  – Evaluation
    • Question: How often does the compiler under-fill a block to meet a constraint?
    • Answer: almost never
• Towards optimized code
  – TRIPS specific optimization policies
  – Scheduling for TRIPS
  – Cache bank optimization
• Preliminary evaluation
Towards Optimized Code

• Filling TRIPS blocks with useful instructions
  – Better hyperblock formation
  – Loop unrolling for TRIPS blocks constraints
  – Aggressive inlining
  – Driving block formation with path and/or edge profiles

• Shortening the critical path in a block
  – Tree-height reduction
  – Redundant computation

• 64 bit machine optimizations

• etc.
TRIPS Scheduling Problem

• Place instructions on 4x4x8 grid
• Encode placement in target form
Contrast with Conventional Approaches

- **VLIW**
  - Relies completely on compiler to schedule code
  + Eliminates need for dynamic dependence check hardware
  + Good match for partitioning
  + Can minimize communication latencies on critical paths
  - Poor tolerance to unpredictable dynamic latencies
    - These latencies continue to grow

- **Superscalar approach**
  - Hardware dynamically schedules code
  + Can tolerate dynamic latencies
  - Quadratic complexity of dependence check hardware
  - Not a good match for partitioning
    - Difficult to make good placement decisions
    - ISA does not allow software to help with instruction placement
Dissecting the Problem

• Scheduling is a two-part problem
  – Placement: *Where an instruction executes*
  – Issue: *When an instruction executes*

• VLIW represents one extreme
  – Static Placement and Static Issue (SPSI)
    + Static Placement works well for partitioned architectures
    – Static Issue causes problems with unknown latencies

• Superscalars represent another extreme
  – Dynamic Placement and Dynamic Issue (DPDI)
    + Dynamic Issue tolerates unknown latencies
    – Dynamic Placement is difficult in the face of partitioning
EDGE Architecture Solution

- EDGE: Explicit Dataflow Graph Execution
  - Static Placement and Dynamic Issue (SPDI)
  - Renegotiates the compiler/hardware binary interface
- An EDGE ISA explicitly encodes the dataflow graph specifying targets

**RISC**

1. movi r1, #10
2. movi r2, #20
3. add r3, r2, r1

**EDGE**

- ALU-1: movi #10, ALU-3
- ALU-2: movi #20, ALU-3
- ALU-3: add ALU-4

- Static Placement
  - Explicit DFG simplifies hardware → no HW dependency analysis!
  - Results are forwarded directly through point-to-point network → no associative issue queues!
  - → no global bypass network!

- Dynamic Instruction Issue
  - Instructions execute in original dataflow-order
Scheduling Algorithms (1 & 2)

• CRBLO: list scheduler [PACT 2004]
  – greedy top down
  – C - prioritize critical path
  – R - reprioritize
  – B - load balance for issue contention
  – L - data cache locality
  – O - register bank locality

• BSP: Block Specific Policy
  – Characterizes blocks’ ILP and load/stores
  – if l/s > 50%, pre-place them and schedule bottom up
  – elseif flt pt > 50% or int + flt pt > 90% use CRBLO
  – else preplace l/s and schedule top down greedy
Physical path scheduling
  – Top down, critical path focused
  – Models network contention (N), in addition to issue contention based on instruction latencies (L)
  – Cost function selects an instruction placement that minimizes maximum contention and latency
    • min of max(N, L)
    • PP - load/store pre-placement
    • tried average, difference, etc.
Improvements on Hand Coded Microbenchmarks

Speedup (Normalized)

CRBLO  BSP  min max(L,N)  min max(L,N) + PP

art_1_hand  art_2_hand  art_3_hand  bzip2_3_hand  doppler_GMTI_hand  equake_1_hand  ft2_GMTI_hand  ft4_GMTI_hand  gzip_2_hand  matrix_1_hand  parser_1_hand  sieve_hand  transpose_GMTI_hand  vadd_hand  average

June 4, 2005

UT-Austin TRIPS Tutorial
Spatial L1 Bank D-Cache Load/Store Alignment

• Problem
  – Load/store mapping to data L1 cache bank unknown to compiler

• Example

```c
for (i=0; i<1024; i++) {
  // A[i], L[i], B[i] to all banks
  A[i] += L[i] + B[i];
}
```
Scheduling for L1 Bank D-cache Communication

L2 cache

I G R R R R

I D E A E E E

I D E E E E E

I D E E E E E

I D L E E E E
Spatial Load/Store Alignment

- Solution:
  - Align arrays on bank alignment boundary
    - function of cache line size and number of banks
    - alignment declaration &/or specialized malloc
  - *Jump-and-fill* new loop transformation
    - similar to unroll-and-jam

```c
for (j=0; j<1024; j+=32) {
    for (i=j; i<j+8; i++) {
        C[i   ] += (A[i   ] + B[i   ]); // Bank 0
        C[i+ 8] += (A[i+ 8] + B[i+ 8]); // Bank 1
    }
}
```
Spatial L1 D-cache Communication

L2 cache

I G R R R R
I D S S E E
I D S A E E
I D S L E E
I D L L E E
Unrolling for Larger Hyperblocks

- **Vector Add Unrolled Continuously (tcc 03)**

  ```c
  for (i=0; i<1024; j+=32) {
    C[i  ] += (A[i  ] + B[i  ]); // Bank 0
    C[i+ 1] += (A[i+ 1] + B[i+ 1]); // Bank 0
    C[i+ 2] += (A[i+ 2] + B[i+ 2]); // Bank 0
    ...  
  }
  ```

- **Vector Add Unrolled with Jump-and-Fill (JF Unroll)**

  ```c
  for (i=0; i<1024; j+=32) {
    C[i  ] += (A[i  ] + B[i  ]); // Bank 0
    C[i+ 8] += (A[i+ 8] + B[i+ 8]); // Bank 1
    C[i+ 1] += (A[i+ 1] + B[i+ 1]); // Bank 0
    ...  
  }
  ```
Spatial Load/Store Alignment Results

![Bar Chart]

- Speedup
- Unroll 32C
tcc -03
- Jump&Fill
- JF Unroll

- No Hints
- Hints

June 4, 2005 UT-Austin TRIPS Tutorial
Performance Goals

- Technology comparison points
  - Alpha
    - Fairest comparison we can do
    - High ILP, high frequency, low cycle count machine
    - gcc
    - Alpha compiler
  - Scale on Alpha
  - Scale on TRIPS
- Performance goal: 4x speed up over Alpha
- Benchmarks
  - Hand coded microbenchmarks
    - Metric for measuring compiler success
  - Industry standards: EEMBC, SPEC
Microbenchmark Speedup

- 1.2 avg. speedup O3 to Alpha
- 3.1 avg. speedup hand to Alpha
- 1.5 avg. speedup O4 to Alpha
Progress Towards Correct & Optimized Code

• SPEC 2000 C/Fortran-77 -- 03: basic blocks
  - March 2004: 0/21
  - November 2004: 6/21
  - March 2005: 15/21
  - May 2005: 19/21
• SPEC 2000 C/Fortran-77 -- 04: hammock hyperblocks
  - May 2005: 8/21
• EEMBC -- 04
  - May 2005: 30/30
• GCC torture tests and NIST -- 04
  - May 2005: 30/30
• Plenty of work left to do!
Wrap-up and Discussion

ISCA-32 Tutorial

TRIPS Team and Guests

Department of Computer Sciences
The University of Texas at Austin
Project Plans

• Tools release
  – Hope to build academic/industrial collaborators
  – Plan to release compilers, toolchain, and simulators sometime in the (hopefully early) fall
• Prototype system
  – Plan to have it working December 2005/January 2006
  – Software development effort increasing after tape-out
• Commercialization
  – Actively looking for interested commercial partners
  – Discussing best business models for transfer to industry
  – Happy to listen to ideas

• Will EDGE architectures become widespread?
  – Many questions remain unanswered (power, performance)
  – Promising initial results, hope to have more concrete answers by next spring
  – A key question is: how successful will universal parallel programming become?
    • Perhaps room for parallel programming & EDGE architectures
And finally ...

THANK YOU for your interest, questions and taking all of this time!

Subset of complete manuals and specifications available at:
http://www.cs.utexas.edu/users/cart/trips
Appendix A - Supplemental Materials

- ISA Specification
- Bandwidth summary
- Additional Tile Details
### ISA: Header Chunk Format

- Includes 32 General Register “read” instructions
- Includes 32 General Register “write” instructions
- Up to 128 bits of header information is encoded in the upper nibbles
  - Block Type
  - Block Size
  - Block Flags
  - Store Mask

---

<table>
<thead>
<tr>
<th>Bit Offsets</th>
<th>31</th>
<th>27</th>
<th>6</th>
<th>5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>H0</td>
<td>Read 0.0</td>
<td></td>
<td></td>
<td></td>
<td>Write 0.0</td>
</tr>
<tr>
<td>H1</td>
<td>Read 1.0</td>
<td></td>
<td></td>
<td></td>
<td>Write 1.0</td>
</tr>
<tr>
<td>H2</td>
<td>Read 2.0</td>
<td></td>
<td></td>
<td></td>
<td>Write 2.0</td>
</tr>
<tr>
<td>H3</td>
<td>Read 3.0</td>
<td></td>
<td></td>
<td></td>
<td>Write 3.0</td>
</tr>
<tr>
<td>H4</td>
<td>Read 0.1</td>
<td></td>
<td></td>
<td></td>
<td>Write 0.1</td>
</tr>
<tr>
<td>H5</td>
<td>Read 1.1</td>
<td></td>
<td></td>
<td></td>
<td>Write 1.1</td>
</tr>
<tr>
<td>H6</td>
<td>Read 2.1</td>
<td></td>
<td></td>
<td></td>
<td>Write 2.1</td>
</tr>
<tr>
<td>H7</td>
<td>Read 3.1</td>
<td></td>
<td></td>
<td></td>
<td>Write 3.1</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td>...</td>
</tr>
<tr>
<td>H24</td>
<td>Read 0.6</td>
<td></td>
<td></td>
<td></td>
<td>Write 0.6</td>
</tr>
<tr>
<td>H25</td>
<td>Read 1.6</td>
<td></td>
<td></td>
<td></td>
<td>Write 1.6</td>
</tr>
<tr>
<td>H26</td>
<td>Read 2.6</td>
<td></td>
<td></td>
<td></td>
<td>Write 2.6</td>
</tr>
<tr>
<td>H27</td>
<td>Read 3.6</td>
<td></td>
<td></td>
<td></td>
<td>Write 3.6</td>
</tr>
<tr>
<td>H28</td>
<td>Read 0.7</td>
<td></td>
<td></td>
<td></td>
<td>Write 0.7</td>
</tr>
<tr>
<td>H29</td>
<td>Read 1.7</td>
<td></td>
<td></td>
<td></td>
<td>Write 1.7</td>
</tr>
<tr>
<td>H30</td>
<td>Read 2.7</td>
<td></td>
<td></td>
<td></td>
<td>Write 2.7</td>
</tr>
<tr>
<td>H31</td>
<td>Read 3.7</td>
<td></td>
<td></td>
<td></td>
<td>Write 3.7</td>
</tr>
</tbody>
</table>
**ISA: Read and Write Formats**

**Register Read Instructions**

```
 21 20 16 15 8 7 0
 V GR RT1 RT2 RQ
```

**Register Write Instructions**

```
 5 4 0
 V GR WQ
```

- General Register access is controlled by special read and write instructions
- Read instructions will be executed from the Read Queue
- Write instructions will be executed from the Write Queue
- Read instructions may target the 1\textsuperscript{st} or 2\textsuperscript{nd} operand slot of any instruction
- General registers are divided into 4 architectural banks
  - 32 registers per bank
  - Bank $i$ holds GRs $i$, $i+4$, $i+8$, etc
  - Up to 8 reads and 8 writes may be performed at each bank
  - Bank position is implicit based on read and write instructions in block header
  - Implicit two bits added to 5-bit general register fields for 128 registers total
## ISA: Microarchitecture-Specific Target Formats

<table>
<thead>
<tr>
<th>Bit Position</th>
<th>Target Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>8 7 6 5 4 3 2 1 0</td>
<td>No Target</td>
</tr>
<tr>
<td>00 00 00 000</td>
<td>Write Slot (WQ)</td>
</tr>
<tr>
<td>00 01</td>
<td>WID</td>
</tr>
<tr>
<td>01</td>
<td>Inst. ID</td>
</tr>
<tr>
<td>10</td>
<td>Inst. ID</td>
</tr>
<tr>
<td>11</td>
<td>Inst. ID</td>
</tr>
</tbody>
</table>

- A 9-bit general target specifier is used by most instructions.
- Read target specifiers are truncated to 8 bits, with the implicit 9<sup>th</sup> bit always set to 1 (they cannot target a write slot or a predicate slot).
- The all-zero encoding is used to represent no target.
- Write Queue entry 0.0 is implicitly targeted by branch instructions and cannot be explicitly targeted.
- Most instructions are allowed to target register outputs, predicates, 1<sup>st</sup> operands, and 2<sup>nd</sup> operands.
ISA: More Details on Predication

- Any instruction using the G, I, L, S, or B format may be predicated
- A 2-bit predicate field specified whether an instruction is predicated and, if so, upon what condition
- Predicated instructions must receive all of their normal operands plus a matching predicate before they are allowed to execute
- Instructions that meet these criteria are said to be enabled
- Instructions that don’t meet these criteria are said to be inhibited
- Inhibited instructions may cause other dependent instructions to be indirectly inhibited
- A block completes executing when all of its register and memory outputs are produced (it is not necessary for all of its instructions to execute)

<table>
<thead>
<tr>
<th>PR</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>Not Predicated</td>
</tr>
<tr>
<td>01</td>
<td>(Reserved)</td>
</tr>
<tr>
<td>10</td>
<td>Predicated Upon False</td>
</tr>
<tr>
<td>11</td>
<td>Predicated Upon True</td>
</tr>
</tbody>
</table>
# ISA: Loads and Stores

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Name</th>
<th>Format</th>
<th>Sources</th>
<th>Targets</th>
</tr>
</thead>
<tbody>
<tr>
<td>LB</td>
<td>Load Byte</td>
<td>L</td>
<td>OP0, IMM</td>
<td>T0</td>
</tr>
<tr>
<td>LBS</td>
<td>Load Byte Signed</td>
<td>L</td>
<td>OP0, IMM</td>
<td>T0</td>
</tr>
<tr>
<td>LH</td>
<td>Load Halfword</td>
<td>L</td>
<td>OP0, IMM</td>
<td>T0</td>
</tr>
<tr>
<td>LHS</td>
<td>Load Halfword Signed</td>
<td>L</td>
<td>OP0, IMM</td>
<td>T0</td>
</tr>
<tr>
<td>LW</td>
<td>Load Word</td>
<td>L</td>
<td>OP0, IMM</td>
<td>T0</td>
</tr>
<tr>
<td>LWS</td>
<td>Load Word Signed</td>
<td>L</td>
<td>OP0, IMM</td>
<td>T0</td>
</tr>
<tr>
<td>LD</td>
<td>Load Doubleword</td>
<td>L</td>
<td>OP0, IMM</td>
<td>T0</td>
</tr>
<tr>
<td>SB</td>
<td>Store Byte</td>
<td>S</td>
<td>OP0, OP1, IMM</td>
<td>None</td>
</tr>
<tr>
<td>SH</td>
<td>Store Halfword</td>
<td>S</td>
<td>OP0, OP1, IMM</td>
<td>None</td>
</tr>
<tr>
<td>SW</td>
<td>Store Word</td>
<td>S</td>
<td>OP0, OP1, IMM</td>
<td>None</td>
</tr>
<tr>
<td>SD</td>
<td>Store Doubleword</td>
<td>S</td>
<td>OP0, OP1, IMM</td>
<td>None</td>
</tr>
</tbody>
</table>

- \( \text{LB: } T_0 = \text{ZEXT} (\text{MEM}_B[\text{OP0} + \text{SEXT}(\text{IMM})]) \)
- \( \text{LBS: } T_0 = \text{SEXT} (\text{MEM}_B[\text{OP0} + \text{SEXT}(\text{IMM})]) \)
- \( \text{LH: } T_0 = \text{ZEXT} (\text{MEM}_H[\text{OP0} + \text{SEXT}(\text{IMM})]) \)
- \( \text{LHS: } T_0 = \text{SEXT} (\text{MEM}_H[\text{OP0} + \text{SEXT}(\text{IMM})]) \)
- \( \text{LW: } T_0 = \text{ZEXT} (\text{MEM}_W[\text{OP0} + \text{SEXT}(\text{IMM})]) \)
- \( \text{LWS: } T_0 = \text{SEXT} (\text{MEM}_W[\text{OP0} + \text{SEXT}(\text{IMM})]) \)
- \( \text{LD: } T_0 = \text{ZEXT} (\text{MEM}_D[\text{OP0} + \text{SEXT}(\text{IMM})]) \)
- \( \text{SH: } \text{MEM}_H[\text{OP0} + \text{SEXT}(\text{IMM})] = \text{OP1}[15:0] \)
- \( \text{SW: } \text{MEM}_W[\text{OP0} + \text{SEXT}(\text{IMM})] = \text{OP1}[31:0] \)
- \( \text{SD: } \text{MEM}_D[\text{OP0} + \text{SEXT}(\text{IMM})] = \text{OP1}[63:0] \)
### ISA: Integer Arithmetic

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Name</th>
<th>Format</th>
<th>Sources</th>
<th>Targets</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD</td>
<td>Add</td>
<td>G</td>
<td>OP0, OP1</td>
<td>T0, T1</td>
</tr>
<tr>
<td>ADDI</td>
<td>Add Immediate</td>
<td>I</td>
<td>OP0, IMM</td>
<td>T0</td>
</tr>
<tr>
<td>SUB</td>
<td>Subtract</td>
<td>G</td>
<td>OP0, OP1</td>
<td>T0, T1</td>
</tr>
<tr>
<td>SUBI</td>
<td>Subtract Immediate</td>
<td>I</td>
<td>OP0, IMM</td>
<td>T0</td>
</tr>
<tr>
<td>MUL</td>
<td>Multiply</td>
<td>G</td>
<td>OP0, OP1</td>
<td>T0, T1</td>
</tr>
<tr>
<td>MULI</td>
<td>Multiply Immediate</td>
<td>I</td>
<td>OP0, IMM</td>
<td>T0</td>
</tr>
<tr>
<td>DIVS</td>
<td>Divide Signed</td>
<td>G</td>
<td>OP0, OP1</td>
<td>T0, T1</td>
</tr>
<tr>
<td>DIVSI</td>
<td>Divide Signed Immediate</td>
<td>I</td>
<td>OP0, IMM</td>
<td>T0</td>
</tr>
<tr>
<td>DIVU</td>
<td>Divide Unsigned</td>
<td>G</td>
<td>OP0, OP1</td>
<td>T0, T1</td>
</tr>
<tr>
<td>DIVUI</td>
<td>Divide Unsigned Immediate</td>
<td>I</td>
<td>OP0, IMM</td>
<td>T0</td>
</tr>
</tbody>
</table>

- There is no support for overflow detection or extended arithmetic
- The multiply instruction may be split into signed and unsigned versions
# ISA: Integer Logical

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Name</th>
<th>Format</th>
<th>Sources</th>
<th>Targets</th>
</tr>
</thead>
<tbody>
<tr>
<td>AND</td>
<td>Bitwise AND</td>
<td>G</td>
<td>OP0, OP1</td>
<td>T0, T1</td>
</tr>
<tr>
<td>ANDI</td>
<td>Bitwise AND Immediate</td>
<td>I</td>
<td>OP0, IMM</td>
<td>T0</td>
</tr>
<tr>
<td>OR</td>
<td>Bitwise OR</td>
<td>G</td>
<td>OP0, OP1</td>
<td>T0, T1</td>
</tr>
<tr>
<td>ORI</td>
<td>Bitwise OR Immediate</td>
<td>I</td>
<td>OP0, IMM</td>
<td>T0</td>
</tr>
<tr>
<td>XOR</td>
<td>Bitwise XOR</td>
<td>G</td>
<td>OP0, OP1</td>
<td>T0, T1</td>
</tr>
<tr>
<td>XORI</td>
<td>Bitwise XOR Immediate</td>
<td>I</td>
<td>OP0, IMM</td>
<td>T0</td>
</tr>
</tbody>
</table>

- XORI may be used to perform a bitwise complement – `XORI( A, -1 )`
## ISA: Integer Shift

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Name</th>
<th>Format</th>
<th>Sources</th>
<th>Targets</th>
</tr>
</thead>
<tbody>
<tr>
<td>SLL</td>
<td>Shift Left Logical</td>
<td>G</td>
<td>OP0, OP1</td>
<td>T0, T1</td>
</tr>
<tr>
<td>SLLI</td>
<td>Shift Left Logical Immediate</td>
<td>I</td>
<td>OP0, IMM</td>
<td>T0</td>
</tr>
<tr>
<td>SRL</td>
<td>Shift Right Logical</td>
<td>G</td>
<td>OP0, OP1</td>
<td>T0, T1</td>
</tr>
<tr>
<td>SRLI</td>
<td>Shift Right Logical Immediate</td>
<td>I</td>
<td>OP0, IMM</td>
<td>T0</td>
</tr>
<tr>
<td>SRA</td>
<td>Shift Right Arithmetic</td>
<td>G</td>
<td>OP0, OP1</td>
<td>T0, T1</td>
</tr>
<tr>
<td>SRAI</td>
<td>Shift Right Arithmetic Immediate</td>
<td>I</td>
<td>OP0, IMM</td>
<td>T0</td>
</tr>
</tbody>
</table>

- Operand 1 provides the data to be shifted
- Operand 2 or IMM provides the shift count
- The lower 6 bits of the shift count are interpreted as an unsigned quantity
- The higher bits of the shift count are ignored
### ISA: Integer Extend

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Name</th>
<th>Format</th>
<th>Sources</th>
<th>Targets</th>
</tr>
</thead>
<tbody>
<tr>
<td>EXTSB</td>
<td>Extend Signed Byte</td>
<td>G</td>
<td>OP0</td>
<td>T0, T1</td>
</tr>
<tr>
<td>EXTSH</td>
<td>Extend Signed Halfword</td>
<td>G</td>
<td>OP0</td>
<td>T0, T1</td>
</tr>
<tr>
<td>EXTSW</td>
<td>Extend Signed Word</td>
<td>G</td>
<td>OP0</td>
<td>T0, T1</td>
</tr>
<tr>
<td>EXTUB</td>
<td>Extend Unsigned Byte</td>
<td>G</td>
<td>OP0</td>
<td>T0, T1</td>
</tr>
<tr>
<td>EXTUH</td>
<td>Extend Unsigned Halfword</td>
<td>G</td>
<td>OP0</td>
<td>T0, T1</td>
</tr>
<tr>
<td>EXTUW</td>
<td>Extend Unsigned Word</td>
<td>G</td>
<td>OP0</td>
<td>T0, T1</td>
</tr>
</tbody>
</table>

- These are better than doing a left shift followed by a right shift
## ISA: Integer Test

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Name</th>
<th>Format</th>
<th>Sources</th>
<th>Targets</th>
</tr>
</thead>
<tbody>
<tr>
<td>TEQ</td>
<td>Test EQ</td>
<td>G</td>
<td>OP0, OP1</td>
<td>T0, T1</td>
</tr>
<tr>
<td>TNE</td>
<td>Test NE</td>
<td>G</td>
<td>OP0, OP1</td>
<td>T0, T1</td>
</tr>
<tr>
<td>TLE</td>
<td>Test LE</td>
<td>G</td>
<td>OP0, OP1</td>
<td>T0, T1</td>
</tr>
<tr>
<td>TLEU</td>
<td>Test LE Unsigned</td>
<td>G</td>
<td>OP0, OP1</td>
<td>T0, T1</td>
</tr>
<tr>
<td>TLT</td>
<td>Test LT</td>
<td>G</td>
<td>OP0, OP1</td>
<td>T0, T1</td>
</tr>
<tr>
<td>TLTU</td>
<td>Test LT Unsigned</td>
<td>G</td>
<td>OP0, OP1</td>
<td>T0, T1</td>
</tr>
<tr>
<td>TGE</td>
<td>Test GE</td>
<td>G</td>
<td>OP0, OP1</td>
<td>T0, T1</td>
</tr>
<tr>
<td>TGEU</td>
<td>Test GE Unsigned</td>
<td>G</td>
<td>OP0, OP1</td>
<td>T0, T1</td>
</tr>
<tr>
<td>TGT</td>
<td>Test GT</td>
<td>G</td>
<td>OP0, OP1</td>
<td>T0, T1</td>
</tr>
<tr>
<td>TGTU</td>
<td>Test GT Unsigned</td>
<td>G</td>
<td>OP0, OP1</td>
<td>T0, T1</td>
</tr>
</tbody>
</table>

- Test instructions produce true (1) or false (0)
## ISA: Integer Test #2

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Name</th>
<th>Format</th>
<th>Sources</th>
<th>Targets</th>
</tr>
</thead>
<tbody>
<tr>
<td>TEQI</td>
<td>Test EQ Immediate</td>
<td>I</td>
<td>OP0, IMM</td>
<td>T1</td>
</tr>
<tr>
<td>TNEI</td>
<td>Test NE Immediate</td>
<td>I</td>
<td>OP0, IMM</td>
<td>T1</td>
</tr>
<tr>
<td>TLEI</td>
<td>Test LE Immediate</td>
<td>I</td>
<td>OP0, IMM</td>
<td>T1</td>
</tr>
<tr>
<td>TLEUI</td>
<td>Test LE Unsigned Immediate</td>
<td>I</td>
<td>OP0, IMM</td>
<td>T1</td>
</tr>
<tr>
<td>TLTI</td>
<td>Test LT Immediate</td>
<td>I</td>
<td>OP0, IMM</td>
<td>T1</td>
</tr>
<tr>
<td>TLTUI</td>
<td>Test LT Unsigned Immediate</td>
<td>I</td>
<td>OP0, IMM</td>
<td>T1</td>
</tr>
<tr>
<td>TGEI</td>
<td>Test GE Immediate</td>
<td>I</td>
<td>OP0, IMM</td>
<td>T1</td>
</tr>
<tr>
<td>TGEUI</td>
<td>Test GE Unsigned Immediate</td>
<td>I</td>
<td>OP0, IMM</td>
<td>T1</td>
</tr>
<tr>
<td>TGTI</td>
<td>Test GT Immediate</td>
<td>I</td>
<td>OP0, IMM</td>
<td>T1</td>
</tr>
<tr>
<td>TGTTUI</td>
<td>Test GT Unsigned Immediate</td>
<td>I</td>
<td>OP0, IMM</td>
<td>T1</td>
</tr>
</tbody>
</table>

- Test instructions produce true (1) or false (0)
ISA: Floating Point Arithmetic

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Name</th>
<th>Format</th>
<th>Sources</th>
<th>Targets</th>
</tr>
</thead>
<tbody>
<tr>
<td>FADD</td>
<td>FP Add</td>
<td>G</td>
<td>OP0, OP1</td>
<td>T0, T1</td>
</tr>
<tr>
<td>FSUB</td>
<td>FP Subtract</td>
<td>G</td>
<td>OP0, OP1</td>
<td>T0, T1</td>
</tr>
<tr>
<td>FMUL</td>
<td>FP Multiply</td>
<td>G</td>
<td>OP0, OP1</td>
<td>T0, T1</td>
</tr>
<tr>
<td>FDIV</td>
<td>FP Divide (simulator only!)</td>
<td>G</td>
<td>OP0, OP1</td>
<td>T0, T1</td>
</tr>
</tbody>
</table>

- These operations are performed assuming double-precision values
- All results are rounded as necessary using the default IEEE rounding mode (round-to-nearest)
- No exceptions are reported
- There is no support for gradual underflow or denormal representations
ISA: Floating-Point Test

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Name</th>
<th>Format</th>
<th>Sources</th>
<th>Targets</th>
</tr>
</thead>
<tbody>
<tr>
<td>FEQ</td>
<td>FP Test EQ</td>
<td>G</td>
<td>OP0, OP1</td>
<td>T0, T1</td>
</tr>
<tr>
<td>FNE</td>
<td>FP Test NE</td>
<td>G</td>
<td>OP0, OP1</td>
<td>T0, T1</td>
</tr>
<tr>
<td>FLE</td>
<td>FP Test LE</td>
<td>G</td>
<td>OP0, OP1</td>
<td>T0, T1</td>
</tr>
<tr>
<td>FLT</td>
<td>FP Test LT</td>
<td>G</td>
<td>OP0, OP1</td>
<td>T0, T1</td>
</tr>
<tr>
<td>FGE</td>
<td>FP Test GE</td>
<td>G</td>
<td>OP0, OP1</td>
<td>T0, T1</td>
</tr>
<tr>
<td>FGT</td>
<td>FP Test GT</td>
<td>G</td>
<td>OP0, OP1</td>
<td>T0, T1</td>
</tr>
</tbody>
</table>

- These operations are performed assuming double-precision values
- Test instructions produce true (1) or false (0)
### ISA: Floating Point Conversion

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Name</th>
<th>Format</th>
<th>Sources</th>
<th>Targets</th>
</tr>
</thead>
<tbody>
<tr>
<td>FITOD</td>
<td>Convert Integer to Double FP</td>
<td>G</td>
<td>OP0</td>
<td>T0, T1</td>
</tr>
<tr>
<td>FDTOI</td>
<td>Convert Double FP to Integer</td>
<td>G</td>
<td>OP0</td>
<td>T0, T1</td>
</tr>
<tr>
<td>FSTOD</td>
<td>Convert Single FP to Double FP</td>
<td>G</td>
<td>OP0</td>
<td>T0, T1</td>
</tr>
<tr>
<td>FDTOS</td>
<td>Convert Double FP to Single FP</td>
<td>G</td>
<td>OP0</td>
<td>T0, T1</td>
</tr>
</tbody>
</table>

- FITOD: Converts a 64-bit signed integer to a double-precision float
- FDTOI: Converts a double-precision float to a 64-bit signed integer
- FSTOD: Converts a single-precision float to a double-precision float
- FDTOS: Converts a double-precision float to a single-precision float
- FSTOD is needed to load of a single-precision float value
- FDTOS is needed to store of a single-precision float value
Predication should be used for conditional branches
Special branch and call instructions may produce an address offset (in chunks) rather than a full address
Call and return behave like normal branches (but may be treated in a special way by a hardware branch predictor)
The System Call instruction will trigger a System Call Exception
### ISA: Miscellaneous

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Name</th>
<th>Format</th>
<th>Sources</th>
<th>Targets</th>
</tr>
</thead>
<tbody>
<tr>
<td>NULL</td>
<td>Nullify Output</td>
<td>G</td>
<td>None</td>
<td>T0, T1</td>
</tr>
<tr>
<td>MOV</td>
<td>Move</td>
<td>G</td>
<td>OP0</td>
<td>T0, T1</td>
</tr>
<tr>
<td>MOVI</td>
<td>Move Immediate</td>
<td>I</td>
<td>IMM</td>
<td>T0</td>
</tr>
<tr>
<td>MOV3</td>
<td>Move to three targets</td>
<td>M3</td>
<td>OP0</td>
<td>T0, T1, T2</td>
</tr>
<tr>
<td>MOV4</td>
<td>Move to four targets</td>
<td>M4</td>
<td>OP0</td>
<td>T0, T1, T2, T3</td>
</tr>
<tr>
<td>MFPC</td>
<td>Move from PC</td>
<td>I</td>
<td>None</td>
<td>T0</td>
</tr>
<tr>
<td>GENS</td>
<td>Generate Signed Constant</td>
<td>C</td>
<td>CONST</td>
<td>T0</td>
</tr>
<tr>
<td>GENU</td>
<td>Generate Unsigned Constant</td>
<td>C</td>
<td>CONST</td>
<td>T0</td>
</tr>
<tr>
<td>APP</td>
<td>Append Constant</td>
<td>C</td>
<td>OP0, CONST</td>
<td>T0</td>
</tr>
<tr>
<td>NOP</td>
<td>No Operations</td>
<td>C</td>
<td>None</td>
<td>None</td>
</tr>
<tr>
<td>Lock</td>
<td>Load and Lock</td>
<td>L</td>
<td>OP0</td>
<td>T0</td>
</tr>
</tbody>
</table>

- NULL is a special instruction for cancelling one or more block outputs
- MOV is the same as RPT (but with a conventional name)
- The generate and append instructions may be used to form large constants
- Three instructions are required to form a 48-bit constant
- Four instructions are required to form a 64-bit constant
## Bandwidth Summary

<table>
<thead>
<tr>
<th>Interface</th>
<th>Width</th>
<th>Freq</th>
<th>Multiplier</th>
<th>Bytes/Sec</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processor Data Cache (Fill + Spill)</td>
<td>128 bits</td>
<td>533 MHz</td>
<td>4 * 2</td>
<td>68 GB/s</td>
</tr>
<tr>
<td>Processor Data Cache (Load + Store)</td>
<td>64 bits</td>
<td>533 MHz</td>
<td>4 * 2</td>
<td>34 GB/s</td>
</tr>
<tr>
<td>Processor Inst Cache (Fetch or Fill)</td>
<td>128 bits</td>
<td>533 MHz</td>
<td>4</td>
<td>34 GB/s</td>
</tr>
<tr>
<td>Processor to OCN Interface</td>
<td>128 bits</td>
<td>533 MHz</td>
<td>5 * 2 * 0.8</td>
<td>68 GB/s</td>
</tr>
<tr>
<td>On-Chip Network Link (64B Read)</td>
<td>128 bits</td>
<td>533 MHz</td>
<td>0.8</td>
<td>6.8 GB/s</td>
</tr>
<tr>
<td>On-Chip Memory Bank (64B Read)</td>
<td>128 bits</td>
<td>533 MHz</td>
<td>0.8</td>
<td>6.8 GB/s</td>
</tr>
<tr>
<td>Chip-to-Chip Network Link (64B Read)</td>
<td>64 bits</td>
<td>(266 MHz)</td>
<td>0.8</td>
<td>(1.7 GB/s)</td>
</tr>
<tr>
<td>DDR 266 SDRAM (64B Read)</td>
<td>64 bits</td>
<td>266 MHz</td>
<td>0.5</td>
<td>1.1 GB/s</td>
</tr>
<tr>
<td>External Bus Interface</td>
<td>32 bits</td>
<td>66 MHz</td>
<td>0.33</td>
<td>88 MB/s</td>
</tr>
</tbody>
</table>

(Maximum bandwidth calculations)
G-tile: Access to Architected Registers

Access to architected registers allowed when processor is halted

Only one read/write request handled at a time
OPN: Detailed Router Schematic
E-Tile: List of Pipelines

- **Instruction Dispatch Pipeline (GDN)**
  - Update Instruction Register File and Status Buffer

- **Operand Delivery Pipeline (OPN)**
  - Update Operand Register File and Status Buffer

- **Instruction Wakeup and Select Pipeline**
  - Two stage instruction selection among three potential candidates
    - *Definite ready* instruction
    - *Instruction Late Selection*: instruction with remote/local bypassed operand
  - Guarantee absence of retire conflict

- **Instruction Execute and Writeback Pipeline**
  - Pipelined execution
  - Retire local targets through local bypass
  - Retire remote targets through OPN

- **Instruction Flush / Commit Pipeline (GCN)**
  - Clear block state (pipeline registers, status buffer etc.)
E-tile: Two Stage Select-Execute Pipeline

1. Instruction Wakeup
2. Issue Prioritization
3. Scoreboard Access
4. Instruction, Operand Buffer Read

Previous Remote Bypass Data
Previous Local Bypass Data
Register Data
Immediate Data
Local Bypass Data
OPN Remote Bypass Data

EXECUTE
Control Packet

Previous Remote Bypass Data
Previous Local Bypass Data
Register Data
Immediate Data
Local Bypass Data
OPN Remote Bypass Data

DEFINITE INSTRUCTION SELECTION CYCLE
INSTRUCTION LATE SELECTION CYCLE
INSTRUCTION EXECUTION CYCLE
M-Tile: Internal Structure

OCN dataIn [127:0]
OCN typeIn [1:0]
OCN validIn [3:0]

OCN dataOut [127:0]
OCN typeOut [1:0]
OCN validOut [3:0]

OCN Incoming Interface

OCN Outgoing Interface

Tag Array

Data Array

MT decoder

MT encoder

MSHR

OCN incoming tag array

OCN outgoing tag array

OCN incoming type array

OCN incoming valid array

OCN outgoing tag array

OCN outgoing type array

OCN outgoing valid array
EBC: External Bus Controller Datapath

- Provides an interface between the 440 GP and TRIPS chips
- GPIOs for general purpose I/O
- Synchronizes between 66MHz 440GP bus and system clock
- Interrupt process:
  - CPU’s and DMAs notify EBC of an exception through the “intr” interface
  - Interrupt controller calls service routines in 440 GP via irq line
  - 440 GP writes to the Halt Controller’s registers to stop the processors
  - 400 GP services the interrupt
• Memory Controller provided by IBM
• SDC Interface provides:
  – Protocol conversion between OCN and Controller’s interface.
  – Synchronization between SDRAM clock and OCN clock
  – Address remapping from System address to 2GB SDRAMs
  – Error detection and reply for misaligned/out of range requests
  – Support for OCN swap operation
DMA Controller

- Facilitates transfer of data from one set of system addresses to another.
- Transfer from and to any addresses except CFG space
- Three types of transfers:
  - Single Block contiguous
  - Single Block scatter/gather
  - Multi Block linked list
- Up to 64KB transfer per block
- 512 byte buffer to optimize OCN traffic