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 Symanski, Ken Michelson, and Andrew Schroetlin
  - Improvements to the EDGE ISA
  - 3 major 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
  - New 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 reduced power and exploitation of concurrency

Performance Scaling and Technology Challenges

EDGE Architectures: Can New ISAs Help?

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

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

- late '00s, '10s
  - Blocks amortize per-inst overhead
  - Hundreds to thousands in flight
  - Inter-inst. communication explicit
  - Exploits more concurrency

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 expand empty 32-bit chunks to NOPs
- Load/store IDs:
  - Maximum of 32 loads/stores may be emitted, but blocks can hold more than 32
- Registers:
  - 8 write auto. max to reg
  - 8 write auto. max to reg
  - 8 write auto. max to reg
- Control flow:
  - Exactly one branch emitted
  - Blocks may hold more

June 4, 2005 UT-Austin TRIPS Tutorial 13

TRIPS Block Contents

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
  - Outputs may be actual or null
  - Different blocks may have different
  - No block state may be written until

June 4, 2005 UT-Austin TRIPS Tutorial 14

TRIPS Block Example

RISC code
- TIL (operand format)
- TIL (target format)

```
.id R3, 4(R2)
.add R4, R3, R5
.addi R5, R4, #2
.beg B4, Block3

[RB4] R5, S1, S2
[RB5] S1, S2, S3
[RB6] S3, S4, S5
[RB7] S4, S5, S6
[h<TB6> block6]
[wb<TB6> block6]
.write S6, S5

.bbgin block2 ...
```

June 4, 2005 UT-Austin TRIPS Tutorial 15

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

June 4, 2005 UT-Austin TRIPS Tutorial 16

TRIPS Instruction Formats

June 4, 2005 UT-Austin TRIPS Tutorial 17

TRIPS G and L formats

```
G:
7 2 5 9
<table>
<thead>
<tr>
<th>opcode</th>
<th>Reg</th>
<th>reg</th>
<th>T1</th>
<th>Target field</th>
</tr>
</thead>
</table>
00: unpredicated | 01: reserved | 10: predicted on false | 11: predicted on true |
```

June 4, 2005 UT-Austin TRIPS Tutorial 18
**Target (Operand) Formats**

- **Target field**: 9 bits
  - 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**

- **Operand field**: 2 bits
  - 00: Write (or left operand)
  - 01: Write (or right operand)

- **Names one of 32 write instructions**
- **Location of instruction is microarchitecture dependent**

---

**Object Code Example**

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

---

**Object Code**

```
```

---

**TRIPS Block Format**

- **Each block is formed from two to five 128-byte program segments**
- **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 $I$ or $T$ ($T$ 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.

---

**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
Instruction (Store) Merging

- Old: two copies of stores down distinct predicate paths
  - a \text{ teqz } b
  - null addi (1)
  - null addi (1)

- New: merge redundant insts.
  - Only one data source will fire
  - p \text{ teqz } muli addi (1)
  - null addi (1)
  - null addi (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)

### Example 1

Block with store down one conditional path:

\[
\begin{align*}
\text{if } (p==0) \\
\quad z &= a \times 2 + 3 \\
\end{align*}
\]

### Example 2

\[
\begin{align*}
\text{null } a \\
\text{null addi (1)}
\end{align*}
\]

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 Chip Block Diagram

TRIPS Proc 0
TRIPS Proc 1
DMA Ctrl 0
DMA Ctrl 1

On-Chip Network

L2 Cache (1MB)

External Bus Ctrl
C2C Ctrl

SDRAM Ctrl 0
SDRAM Ctrl 1

On-Chip Memory

External Bus Interface

Chip-to-Chip Links (266 MHz)

SDRAM Modules (DDR-266)

TRIPS Proc 0
TRIPS Proc 1
DMA Ctrl 0
DMA Ctrl 1

On-Chip Network

L2 Cache (1MB)

External Bus Ctrl
C2C Ctrl

SDRAM Ctrl 0
SDRAM Ctrl 1

On-Chip Memory

External Bus Interface

Chip-to-Chip Links (266 MHz)

TRIPS Tile-level Microarchitecture

TRIPS Tiles
G: Processor control - TLB w/ variable size pages, dispatch, instruction block predict, commit
R: Register file - 32 registers, 4 threads, register forwarding
I: Instruction cache - 16KB storage per tile
D: Data cache - 4KB per tile, 256-entry last-level queue, TLB
E: Execution unit - x32 ALUs, 64 reservation stations
M: Memory - 16KB, 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

Mapping TRIPS Blocks to the Microarchitecture

Header
Chunk
Inst.

Chunk 0
Inst.

Chunk 3

Block i mapped into Frame 0

Block i+1 mapped into Frame 1

Instruction reservation stations

Instruction reservation stations

Header
Chunk
Inst.

Chunk 0
Inst.

Chunk 3

Block i mapped into Frame 0

Block i+1 mapped into Frame 1

Instruction reservation stations

Instruction reservation stations
Mapping Target Identifiers to Reservation Stations

Processor Core Tiles and Interfaces

Mapping Target Identifiers to Reservation Stations

Processor Core Tiles and Interfaces

Block Execution Timeline

On-Chip Network
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 TLM BIM ASIC process
  - 8.3mm x 16.3mm die
  - 183 signal IOs
  - 668 for C2C
  - 216 for SDCs
  - ~70 misc IOs

- Schedule
  - 5/99: Seed of TRIPS ideas
  - 12/01: Publish TRIPS concept architecture
  - 6/02: 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 refresh

TRIPS Prototype System Board

- TRIPS chip network
  - 4 TRIPS chips (8 processors) per board
  - Communicates 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

- Clock speeds
  - 533MHz internal clock
  - 266MHz C2C clock
  - 133/266 SDRAM clock
- Single chip
  - 2 processors per chip each with
    - 64 KB L1-I, 32 KB L1-D
    - 8.5 Gops or GFlops/processor
    - 17 Gops or GFlops/chip peak
    - Up to 8 threads
    - 1 MB on-chip memory
    - 2 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/sec 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

June 4, 2005 UT-Austin TRIPS Tutorial 49

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

```
#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

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
• It achieves ~7.4 IPC, ~3.2 loads & stores per cycle

Vector Add – TASL Code

• TRIPS Assembly Language (TASL) files are fed to the assembler
• Instructions are described using a dataflow notation

Linked List – C Code

• 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

• Here is a TIL representation of the hyperblock
• Predicate p1 represents the test to determine whether the tag matches the key
• Predicate p0 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:
  – p1 true, p0 false
  – p1 false, p0 false
  – p1 false, p0 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 canceled if not properly nullified
• Loads are allowed to execute speculatively because exceptions will be predicated downstream

Linked List – Hyperblock #1

– p0 false, p1 true
– p0 true, p1 unresolved

• 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 predicate to form a larger program block (or hyperblock)
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
- There 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

```
.p == 0
(...)
F

.p->tag == key
.p = p->next
T

Hyperblock

.p == 0
F
.p->tag == key
.p->value++
0x0
.p = p->next
T
0x0
T
```

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

Global Control

ISCA-32 Tutorial
Ramadas Nagarajan
Department of Computer Sciences
The University of Texas at Austin

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 acknowledgments 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)

| Interface Ports |
|-----------------|-----------------|
| CMD: None/Fetch/Dispatch |
| ADDR: Address of block |
| SMASK: Store mask in block |
| ID: Frame execution flags |
| INST: Instruction bits |

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

Global Refill Network (GRN)

| Interface Ports |
|-----------------|-----------------|
| CMD: None/Refill |
| RID: Refill buffer identifier |
| ADDR: Address of block to refill |

- Delivers refill commands to I-Tiles

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)

| Interface Ports |
|-----------------|-----------------|
| CMD: None/Completed/Committed |
| ID: Frame/Refill buffer identifier |
| STATUS: Exception Status |

- 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

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
- Ready for Fetch
- Dispatch
- Execute
- Completed
- Commit
- Deallocated
- Flushed
- Converting

Major G-Tile Structures

- Fetch Unit
- Commit/Flush Ctrl
- I-cache dir
- ITLB
- Alloc Fetch
- Flush Commit
- Acks
- GDN/GRN
- GDN
- GSN
- GCN
- ESN
- Exit Predictor
- Retire Unit
- Control Registers

Frame Management

Frames allocated from a circular queue:

- Youngest
- Oldest

After a new fetch:

- Non-speculative block
- Speculative blocks

After a commit:

- 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**

- **Block A**: Fetch #0, Fetch #1, Fetch #2, Fetch #3
- **Block B**: Fetch #0, Fetch #1, Fetch #2, Fetch #3

<table>
<thead>
<tr>
<th>Cycle</th>
<th>Predict (Stage 0)</th>
<th>Predict (Stage 1)</th>
<th>Predict (Stage 2)</th>
<th>Fetch Slot 0</th>
<th>Fetch Slot 1</th>
<th>Fetch Slot 2</th>
<th>Fetch Slot 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td>Fetch Slot 0</td>
<td>Fetch Slot 1</td>
<td>Fetch Slot 2</td>
<td>Fetch Slot 3</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td>Fetch Slot 0</td>
<td>Fetch Slot 1</td>
<td>Fetch Slot 2</td>
<td>Fetch Slot 3</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td>Fetch Slot 0</td>
<td>Fetch Slot 1</td>
<td>Fetch Slot 2</td>
<td>Fetch Slot 3</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td>Fetch Slot 0</td>
<td>Fetch Slot 1</td>
<td>Fetch Slot 2</td>
<td>Fetch Slot 3</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td>Fetch Slot 0</td>
<td>Fetch Slot 1</td>
<td>Fetch Slot 2</td>
<td>Fetch Slot 3</td>
</tr>
</tbody>
</table>

- **Hit/Miss detection**
- **Frame allocation**
- **TLB lookup**
- **I-cache lookup**
- **Predict (Stage 0)**
- **Address Select**
- **Predict (Stage 1)**
- **Predict (Stage 2)**

**Fetch Pipeline**

- **Refill Pipeline**

- **Block A**: Fetch Slot 0, Fetch Slot 1, Fetch Slot 2, Fetch Slot 3

<table>
<thead>
<tr>
<th>Cycle</th>
<th>Predict (Stage 0)</th>
<th>Predict (Stage 1)</th>
<th>Predict (Stage 2)</th>
<th>Fetch Slot 0</th>
<th>Fetch Slot 1</th>
<th>Fetch Slot 2</th>
<th>Fetch Slot 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td>Fetch Slot 0</td>
<td>Fetch Slot 1</td>
<td>Fetch Slot 2</td>
<td>Fetch Slot 3</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td>Fetch Slot 0</td>
<td>Fetch Slot 1</td>
<td>Fetch Slot 2</td>
<td>Fetch Slot 3</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td>Fetch Slot 0</td>
<td>Fetch Slot 1</td>
<td>Fetch Slot 2</td>
<td>Fetch Slot 3</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td>Fetch Slot 0</td>
<td>Fetch Slot 1</td>
<td>Fetch Slot 2</td>
<td>Fetch Slot 3</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td>Fetch Slot 0</td>
<td>Fetch Slot 1</td>
<td>Fetch Slot 2</td>
<td>Fetch Slot 3</td>
</tr>
</tbody>
</table>

- **Refill cycle**
- **Control cycle**
- **Fetch cycle**

**Refill Pipeline**

- **Per-Frame State**

<table>
<thead>
<tr>
<th>Valid</th>
<th>Youngest</th>
<th>Oldest</th>
<th>Branch misprediction detected</th>
<th>Next block address</th>
<th>Next block address state (predicted or resolved)</th>
<th>Prediction completed</th>
<th>Register completed</th>
<th>Stores completed</th>
<th>Exception reported</th>
<th>Commit completed</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Commit Pipeline**

<table>
<thead>
<tr>
<th>Cycle 0</th>
<th>Cycle 1</th>
<th>Cycle 2</th>
<th>Cycle 3</th>
<th>Cycle X</th>
<th>Frame completion</th>
</tr>
</thead>
<tbody>
<tr>
<td>Predicted Block Address</td>
<td>Commit sent</td>
<td>Block completed execution</td>
<td>Block commit status</td>
<td>Block completion detection</td>
<td>T-Morph Support</td>
</tr>
</tbody>
</table>

**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**

- **Commit Pipeline**

<table>
<thead>
<tr>
<th>Cycle 0</th>
<th>Cycle 1</th>
<th>Cycle 2</th>
<th>Cycle X</th>
<th>Frame completion</th>
</tr>
</thead>
<tbody>
<tr>
<td>Predicted Block Address</td>
<td>Commit sent</td>
<td>Block completed execution</td>
<td>Block commit status</td>
<td>Block completion detection</td>
</tr>
</tbody>
</table>

**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

**T-Morph Support**

- **Frame Allocation**

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Thread 1</th>
<th>Thread 2</th>
<th>Thread 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Frame 0</td>
<td>Frame 1</td>
<td>Frame 2</td>
<td>Frame 3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Per-thread Registers</th>
</tr>
</thead>
<tbody>
<tr>
<td>GR0</td>
</tr>
<tr>
<td>PC</td>
</tr>
</tbody>
</table>

**Per-thread Registers**
### 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

### Example Hyperblock from Linked List Code

- 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 predictions
  - 8 exit IDs supported
    - e0 "300"
    - e1 "301"
- Predictor predicts the ID of the exit that will fire
  - Implicitly predicts 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

### Predictor Components

- Initial address
  - Exit Predictor
  - Branch Type Predictor
  - Target Predictor
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

Prediction Timing

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
  - 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

Optimizations for Better Prediction

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

ISCA-32 Tutorial

Haiming Liu

Department of Computer Sciences
The University of Texas at Austin

I-Tile Location and Connections

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

Instruction Layout in a Maximal Block

Instruction Fetch Pipeline

Major I-Tile Structures
Register Accesses and Operand Routing

ISCA-32 Tutorial

Karu Sankaralingam

Department of Computer Sciences
The University of Texas at Austin
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

June 4, 2005 UT-Austin TRIPS Tutorial 110

Major Structures in the R-tile

Register Forwarding: RQ and WQ contents

Block completion: Commit unit

Register Forwarding Example

Pipeline diagrams: Register Read and Write
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 Types
  - 0 - Generic transfer or reply
  - 1 - Load
  - 2 - Store
  - 3 - PC read
  - 4 - PC write
  - 14 - Special register read
  - 15 - Special register write

OPN Router Schematic

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]
## Networks Used by E-Tile

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

- **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
  - 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**
- **OPN**
- **Operand Buffer**
- **Status Buffer**
- **Instruction Buffer**
- **Router**

## 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 (Pipeline)</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 (Pipeline)</td>
<td>Synopsys IP</td>
</tr>
<tr>
<td>FP Multiplier</td>
<td>4 (Pipeline)</td>
<td>Synopsys IP</td>
</tr>
<tr>
<td>FP Comparator</td>
<td>2 (Pipeline)</td>
<td>Synopsys IP</td>
</tr>
<tr>
<td>FP Div/Convert</td>
<td>2 (Pipeline)</td>
<td>Synopsys IP</td>
</tr>
<tr>
<td>FP 32/32 Convert</td>
<td>3 (Pipeline)</td>
<td>Synopsys IP</td>
</tr>
<tr>
<td>FP 32/32 Convert</td>
<td>2 (Pipeline)</td>
<td>UT TRIPS IP</td>
</tr>
<tr>
<td>FP 32/32 Convert</td>
<td>3 (Pipeline)</td>
<td>UT TRIPS IP</td>
</tr>
</tbody>
</table>

## Issue Prioritization Among Ready Instructions

- 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

- Older Block First
  - B1 before B2

- Smaller Instruction Slot First
  - B3.10 before B3.12

- 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

- Definite Selection
- Late Selection

Bypass: Sources of Pipeline Bubbles

- OPN stall due to network congestion
- Bypassed operand data in a non-enabling predicate
- Similar pipeline bubble due to invalid bypassed OPN data packet

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
- movl, mov4 instructions have 3 and 4 targets respectively, used to fanout operands to multiple children

Basic Operand Local & Remote Bypass

- Sample Dependence Graph
- Instruction Physical Placement
### D-Tile Location and Connections

![Diagram of D-Tile Location and Connections]

- **G-Tile**
- **E-Tile**
- **GDN**
- **GSN**
- **GCN**
- **OPN**

### 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</td>
<td>-</td>
<td>-</td>
<td>Store 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) local 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 local 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)

- **Cycle 1**
  - Access Cache, TLB, DP, and LSQ

- **Cycle 2**
  - Load Miss

- **Cycle 3**
  - Access Cache, TLB, DP, and LSQ

- **Cycle 4**
  - Load Miss

- **Cycle 5**
  - Access Cache, TLB, DP, and LSQ

- **Cycle 6**
  - Load Miss
**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

![Dependence Prediction Diagram](Image)

**Store Counting**

At block dispatch, each DT receives and records information on stores to

![Store Counting Diagram](Image)

**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

![Store Commit Pipeline Diagram](Image)

**Deferred Load Pipeline**

- **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 dependents, 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

![Deferred Load Pipeline Diagram](Image)

**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**

![Load and Store Miss Handling Diagram](Image)
D-Tile Pipelines Block Diagram

- Coherence checks
  - e.g. Load and store misses to same cache line

D-Tile Roadmap

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)

Secondary Memory System

ISCA-32 Tutorial
Changkyu Kim

Department of Computer Sciences
The University of Texas at Austin

M-Tile Location and Connections

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

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

Read Miss Pipeline

4-cycle latency for sending a fill request to SDC

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

Multiple Modes in TRIPS Secondary Memory

- M-Tile configured as a level-2 Cache
- M-Tile configured as a scratchpad memory

Back-to-Back Read Pipeline

- 7-cycle latency for transferring all 64B data

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**

- 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

**Goals and Requirements of the OCN**

- 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
- One cycle per hop plus cycle at network insertion
- 68 GB/sec at processor interfaces

**OCN Protocol**

- 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
- 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
  - Composable OCN network addresses
  - One cycle per hop latency
- Attributes
  - Composed of OCN router with translation tables on local input
  - Portion of the system address index into translation tables
  - Translation tables configured through memory mapped registers
  - Error back on invalid addresses

**I/O Tiles**

- **EBT Tile**
  - Provides interface between board-level PowerPC 440 GP chip and the TRIPS chip
  - Forwards interrupts to the 440 GP's service routines
- **SDT Tile**
  - Contains IBM SDRAM Memory Controller
  - Error detection, swap support and address remapping added
- **DMA Tile**
  - Supports multi-block and scatter/gather transfers
  - Buffled to optimize traffic

**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

- Address target sent to memory, data received at target
- Reg banks
- 32 read instructions
- 32 write instructions

- Memory
- PC read
- 1 - 128 instruction DFG

- Register banks
- 32 loads
- 32 stores
- Terminating branch

- Memory
- PC

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

- 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

<table>
<thead>
<tr>
<th>Static Averages</th>
<th>03</th>
<th>04</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>min</td>
<td>max</td>
</tr>
<tr>
<td>SPEC2000</td>
<td>7</td>
<td>42</td>
</tr>
<tr>
<td>EEMBC</td>
<td>7</td>
<td>31</td>
</tr>
</tbody>
</table>

SPEC Total TRIPS Blocks

- 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

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

June 4, 2005
UT-Austin TRIPS Tutorial
181

In Progress: More Aggressive Hyperblocks

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 / O4 (current) / new

June 4, 2005
UT-Austin TRIPS Tutorial
182

While Loop Unrolling

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

• Supported on architectures with predication
• Implemented, but needs improved hyperblock generation

June 4, 2005
UT-Austin TRIPS Tutorial
183

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

June 4, 2005
UT-Austin TRIPS Tutorial
184

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
• mesal: 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

June 4, 2005
UT-Austin TRIPS Tutorial
185

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

June 4, 2005
UT-Austin TRIPS Tutorial
186
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

- One branch fires
- All writes complete
  - Write nullification
- All stores complete
  - Store nullification

Register Writes

- 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

Load/Store ID Assignment Example:

Before Assignment:

- LD [0]
- SD [1]

After Assignment:

- 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!)

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.

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

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
  - RISC
    - i1: movi r1, #10
    - i2: movi r2, #20
    - i3: 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
  - Results are forwarded directly through point-to-point network
  - Dynamic Instruction Issue
  - Instructions execute in original dataflow order

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

Scheduling Algorithms (1 & 2)

- CRBLO: list scheduler \textit{[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
  - else if \( \text{flt pt} > 50\% \) or int + flt pt > 90\% use CRBLO
  - else preplace l/s and schedule top down greedy
Scheduling Algorithms (3)

- 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

- 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];
  }
  ```

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];
  }
  ```

Spatial L1 D-cache Communication

- 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

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

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

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: 6/21
• 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 collaboration
  – 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 architecture

Appendix A - Supplemental Materials

• ISA Specification
• Bandwidth summary
• Additional Tile Details

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

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 subfields
  - Block Type
  - Block Size
  - Block Flags
  - Store Mask

ISA: Read and Write Formats

- 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 1st or 2nd 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**

- A 9-bit general target specifier is used by most instructions.
- Read target specifiers are truncated to 8 bits, with the implicit 9th 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, 1st operands, and 2nd operands.

**ISA: More Details on Predication**

- Any instruction using the G, U, 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 those 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>Predicated Upon False</td>
</tr>
<tr>
<td>10</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>LB</td>
<td>Load Byte Signal</td>
<td>L</td>
<td>QP0, IMM</td>
<td>T0</td>
</tr>
<tr>
<td>LH</td>
<td>Load HalfWord</td>
<td>L</td>
<td>OP0, IMM</td>
<td>T1</td>
</tr>
<tr>
<td>LH</td>
<td>Load HalfWord Signal</td>
<td>L</td>
<td>QP0, IMM</td>
<td>T1</td>
</tr>
<tr>
<td>LW</td>
<td>Load Word</td>
<td>L</td>
<td>OP0, IMM</td>
<td>T1</td>
</tr>
<tr>
<td>LW</td>
<td>Load Word Signal</td>
<td>L</td>
<td>QP0, IMM</td>
<td>T1</td>
</tr>
<tr>
<td>LD</td>
<td>Load DoubleWord</td>
<td>L</td>
<td>OP0, IMM</td>
<td>T1</td>
</tr>
<tr>
<td>LW</td>
<td>Load Byte</td>
<td>S</td>
<td>OP0, QP1, IMM</td>
<td>None</td>
</tr>
<tr>
<td>LW</td>
<td>Load HalfWord</td>
<td>S</td>
<td>OP0, QP1, IMM</td>
<td>None</td>
</tr>
<tr>
<td>LW</td>
<td>Load Word</td>
<td>S</td>
<td>OP0, QP1, IMM</td>
<td>None</td>
</tr>
<tr>
<td>LW</td>
<td>Load DoubleWord</td>
<td>S</td>
<td>OP0, QP1, IMM</td>
<td>None</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 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>ADDIM</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>SUBIM</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>MULIM</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>DIVS</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>DIVU</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>AND</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>XOR</td>
<td>Bitwise XOR Immediate</td>
<td>I</td>
<td>OP0, IMM</td>
<td>T0, T1</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>SLT</td>
<td>Shift Left Logical</td>
<td>G</td>
<td>OP0, OP1</td>
<td>T0, T1</td>
</tr>
<tr>
<td>SLLT</td>
<td>Shift Left Logical Immediate</td>
<td>I</td>
<td>OP0, IMM</td>
<td>T0</td>
</tr>
<tr>
<td>SRLT</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.
- Operands 2 or IMM provide 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>XTSHB</td>
<td>Extend Signed Byte</td>
<td>G</td>
<td>OP0</td>
<td>T0, T1</td>
</tr>
<tr>
<td>XTSHH</td>
<td>Extend Signed Halfword</td>
<td>G</td>
<td>OP0</td>
<td>T0, T1</td>
</tr>
<tr>
<td>XTSSH</td>
<td>Extend Signed Word</td>
<td>G</td>
<td>OP0</td>
<td>T0, T1</td>
</tr>
<tr>
<td>XTSHB</td>
<td>Extend Unsigned Byte</td>
<td>G</td>
<td>OP0</td>
<td>T0, T1</td>
</tr>
<tr>
<td>XTSHH</td>
<td>Extend Unsigned Halfword</td>
<td>G</td>
<td>OP0</td>
<td>T0, T1</td>
</tr>
<tr>
<td>XTSSW</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>TEQI</td>
<td>Test EQ Immediate</td>
<td>I</td>
<td>OP0, IMM</td>
<td>T0</td>
</tr>
<tr>
<td>TNEI</td>
<td>Test NE Immediate</td>
<td>I</td>
<td>OP0, IMM</td>
<td>T0</td>
</tr>
<tr>
<td>TLEH</td>
<td>Test LE Immediate</td>
<td>I</td>
<td>OP0, IMM</td>
<td>T0</td>
</tr>
<tr>
<td>TLEH</td>
<td>Test LE Unsigned Immediate</td>
<td>I</td>
<td>OP0, IMM</td>
<td>T0</td>
</tr>
<tr>
<td>TLEH</td>
<td>Test LE Unaligned Immediate</td>
<td>I</td>
<td>OP0, IMM</td>
<td>T0</td>
</tr>
<tr>
<td>TLEH</td>
<td>Test LE Unaligned Immediate</td>
<td>I</td>
<td>OP0, IMM</td>
<td>T0</td>
</tr>
<tr>
<td>TLEH</td>
<td>Test LE Unaligned Immediate</td>
<td>I</td>
<td>OP0, IMM</td>
<td>T0</td>
</tr>
<tr>
<td>TLEH</td>
<td>Test LE Unaligned Immediate</td>
<td>I</td>
<td>OP0, IMM</td>
<td>T0</td>
</tr>
<tr>
<td>TLEH</td>
<td>Test LE Unaligned Immediate</td>
<td>I</td>
<td>OP0, IMM</td>
<td>T0</td>
</tr>
<tr>
<td>TLEH</td>
<td>Test LE Unaligned Immediate</td>
<td>I</td>
<td>OP0, IMM</td>
<td>T0</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>T0EQ</td>
<td>Test EQ Immediate</td>
<td>I</td>
<td>OP0, IMM</td>
<td>T0</td>
</tr>
<tr>
<td>T0NE</td>
<td>Test NE Immediate</td>
<td>I</td>
<td>OP0, IMM</td>
<td>T0</td>
</tr>
<tr>
<td>T0LE</td>
<td>Test LE Immediate</td>
<td>I</td>
<td>OP0, IMM</td>
<td>T0</td>
</tr>
<tr>
<td>T0LE</td>
<td>Test LE Unsigned Immediate</td>
<td>I</td>
<td>OP0, IMM</td>
<td>T0</td>
</tr>
<tr>
<td>T0LE</td>
<td>Test LE Unaligned Immediate</td>
<td>I</td>
<td>OP0, IMM</td>
<td>T0</td>
</tr>
<tr>
<td>T0LE</td>
<td>Test LE Unaligned Immediate</td>
<td>I</td>
<td>OP0, IMM</td>
<td>T0</td>
</tr>
<tr>
<td>T0LE</td>
<td>Test LE Unaligned Immediate</td>
<td>I</td>
<td>OP0, IMM</td>
<td>T0</td>
</tr>
<tr>
<td>T0LE</td>
<td>Test LE Unaligned Immediate</td>
<td>I</td>
<td>OP0, IMM</td>
<td>T0</td>
</tr>
<tr>
<td>T0LE</td>
<td>Test LE Unaligned Immediate</td>
<td>I</td>
<td>OP0, IMM</td>
<td>T0</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>FSDV</td>
<td>FP Double (accumulator 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
- 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>FPEQ</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>FLE</td>
<td>FP Test LE</td>
<td>G</td>
<td>OP0, OP1</td>
<td>T0, T1</td>
</tr>
<tr>
<td>FGEO</td>
<td>FP Test GE</td>
<td>G</td>
<td>OP0, OP1</td>
<td>T0, T1</td>
</tr>
<tr>
<td>FGEO</td>
<td>FP Test GE</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>FDTOD</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>FDITS</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
- FDTOS: Converts a double-precision float to a 64-bit signed integer
- FSTOD: Converts a single-precision float to a double-precision float
- FDITS: 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
### ISA: Control Flow

<table>
<thead>
<tr>
<th>Name</th>
<th>Format</th>
<th>Sources</th>
<th>Targets</th>
</tr>
</thead>
<tbody>
<tr>
<td>BR</td>
<td>B</td>
<td>OP0</td>
<td>PC</td>
</tr>
<tr>
<td>BHO</td>
<td>B</td>
<td>OFFSET</td>
<td>PC</td>
</tr>
<tr>
<td>CALL</td>
<td>B</td>
<td>OP0</td>
<td>PC</td>
</tr>
<tr>
<td>CALLO</td>
<td>B</td>
<td>OFFSET</td>
<td>PC</td>
</tr>
<tr>
<td>RET</td>
<td>B</td>
<td>OP0</td>
<td>PC</td>
</tr>
<tr>
<td>SCALL</td>
<td>B</td>
<td>None</td>
<td>PC</td>
</tr>
</tbody>
</table>

- 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 be a hardware branch predictor)
- The System Call instruction will trigger a System Call Exception

---

### ISA: Miscellaneous

<table>
<thead>
<tr>
<th>Name</th>
<th>Format</th>
<th>Sources</th>
<th>Targets</th>
</tr>
</thead>
<tbody>
<tr>
<td>NULL</td>
<td>G</td>
<td>None</td>
<td>T0, T1</td>
</tr>
<tr>
<td>MOVE</td>
<td>G</td>
<td>OP0</td>
<td>T0, T1</td>
</tr>
<tr>
<td>MOVI</td>
<td>I</td>
<td>IMM</td>
<td>F0</td>
</tr>
<tr>
<td>MOVEV</td>
<td>M3</td>
<td>OP0</td>
<td>T0, T1, T2</td>
</tr>
<tr>
<td>MOIV4</td>
<td>M4</td>
<td>OP0</td>
<td>T0, T1, T2, T3</td>
</tr>
<tr>
<td>MFPC</td>
<td>I</td>
<td>None</td>
<td>F0</td>
</tr>
<tr>
<td>GENES</td>
<td>C</td>
<td>CONST</td>
<td>F0</td>
</tr>
<tr>
<td>GENIU</td>
<td>L</td>
<td>CONST</td>
<td>F0</td>
</tr>
<tr>
<td>APP</td>
<td>C</td>
<td>OP0, CONST</td>
<td>F0</td>
</tr>
<tr>
<td>NOP</td>
<td>C</td>
<td>None</td>
<td>None</td>
</tr>
</tbody>
</table>

- Lock: Load and Lock
- Load: Load and Lock

---

### 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 + Split)</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>

---

### G-tile: Access to Architected Registers

Access to architected registers allowed when processor is halted

Read/write request

```
OCN
```

G-tile

Read reply

```
OPN
```

Only one read/write request handled at a time

---

### 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

STATUS

BUFFER

OPN

CTRL PKT

LOCAL

CTRL PKT

1. Instruction

Wakeup

2. Issue

Prioritization

3. Scoreboard

Access

4. Instruction,

Operand

Buffer

Read

OPN

CTRL PKT

LOCAL

CTRL PKT

Previous Remote Bypass

DataPrevious Local Bypass Data

Register Data

Immediate Data

Local Bypass Data

OPN Remote Bypass Data

EXECUTE

Control

Packet

Previous Remote Bypass

DataPrevious 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

Incoming

Interface

OCN

Outgoing

Interface

O CN dataIn [127:0]

OCN typeIn [1:0]

OCN validIn [3:0]

O CN dataOut [127:0]

OCN typeOut [1:0]

OCN validOut [3:0]

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 DMAa 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

SRC: SDRAM Controller Diagram

• 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