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
  - Polymorphous Computing Architectures (2001-2006)
    - Contracts F33615-01-C-1892 and F44615-03-C-4106
  - Special thanks to Bob Graybill
- Air Force Research Laboratory
- Intel Research Council
  - 2 Research Grants (2000-2004) and equipment donations (2000-2005)
- IBM
  - Faculty Partnership Awards (1999-2004) and SUR grants (1999, 2003)
- Sun Microsystems
  - Research Grant (2003)
- 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

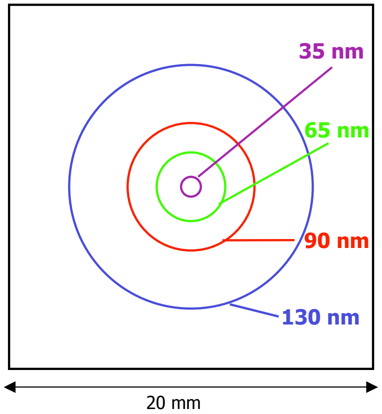
- “The Design and Implementation of the TRIPS Prototype Chip”
  - HotChips 17, 2005.
  - Contains details about the prototype ASIC.
- “Dynamic Placement, Static Issue (SPDI) Scheduling for EDGE Architectures”
  - Intl. Conference on Parallel Architectures and Compilation Techniques (PACT), 2004.
  - 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”
  - Intl. Symposium on Computer Architecture (ISCA-30), 2003.
  - 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”
  - Intl. Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-X), 2002.
  - The first paper describing NUCA caches and their design tradeoffs.
- “A Design Space Evaluation of Grid Processor Architectures”
  - Intl. Symposium on Microarchitecture (MICRO-34), 2001.
  - 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

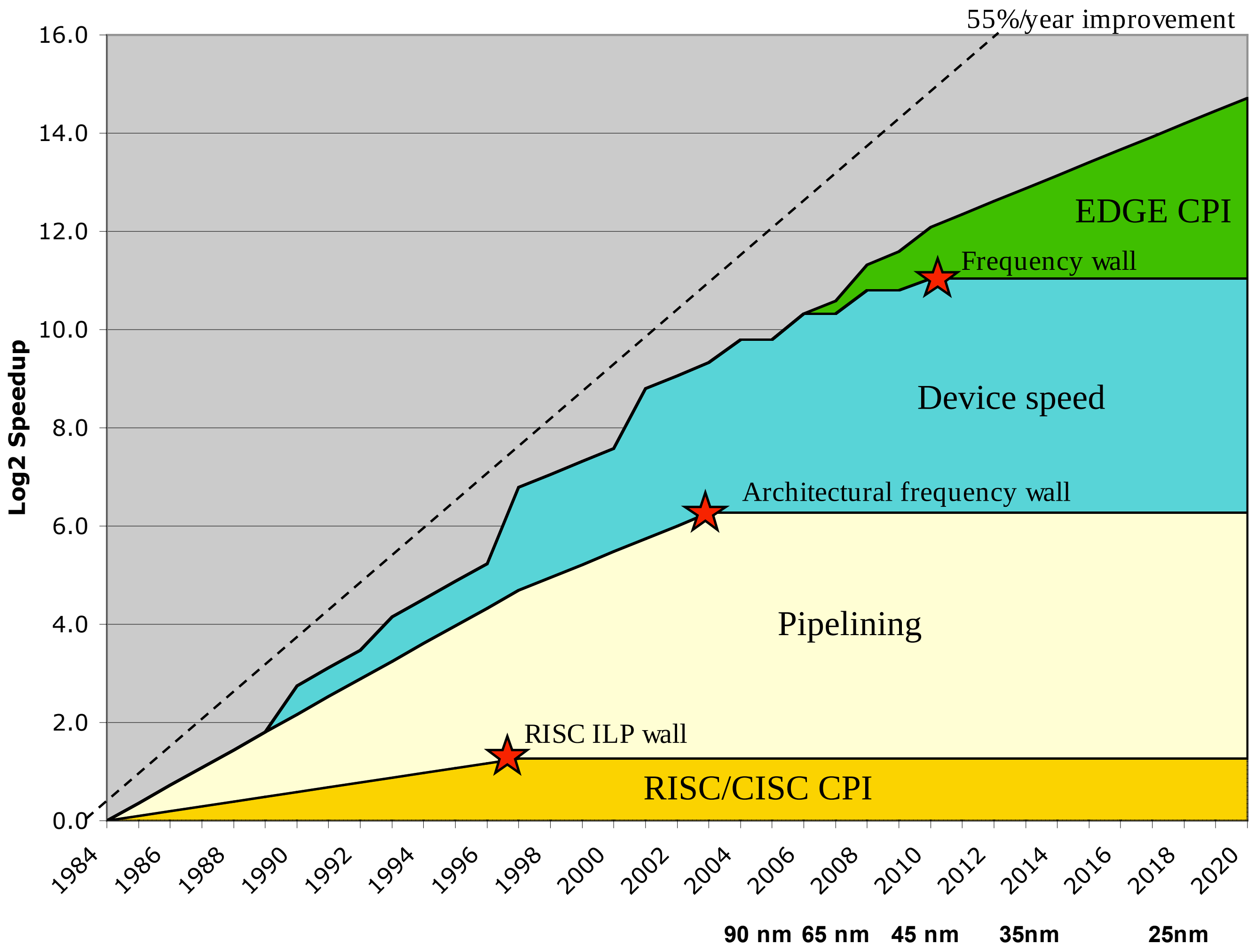
Key Trends

- 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



New programming models needed?

Concurrency only solution for higher performance

Conventional architectures cannot improve performance

Industry shifts to frequency dominated strategy

EDGE Architectures: Can New ISAs Help?

'60s, '70s

'80s, '90s, early '00s

late '00s, '10s

CISC

RISC[1]

EDGE?

Complex, few instructions
Discrete components
Minimize storage
Small numbers in flight

More numerous, simple instructions
Reliance on compiler ordering
Optimized for pipelining
Tens of instructions in flight

Blocks amortize per-inst overhead
Hundreds to thousands in flight
Inter-inst. communication explicit
Exploits more concurrency

Pipelining difficult

Wide issue inefficient

Leakage not yet addressed
Reliability not yet addressed

[1] RISC 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

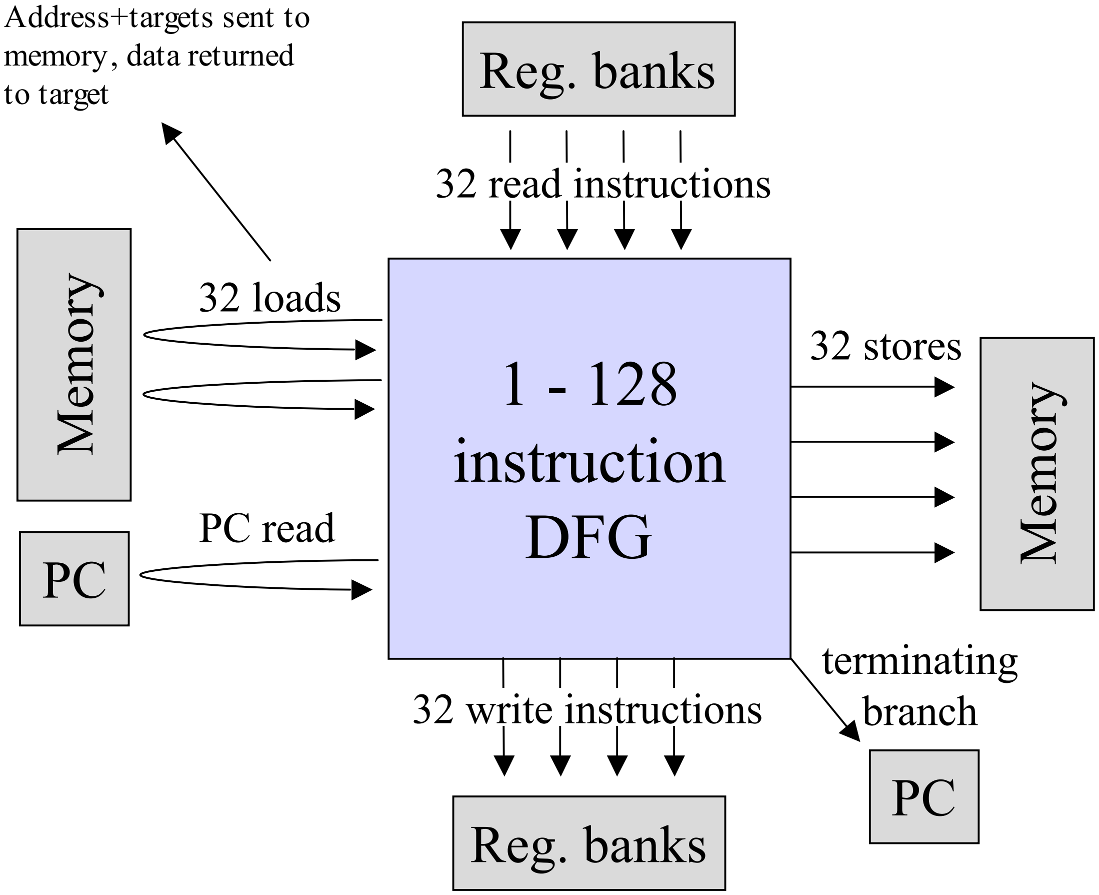
What is an EDGE Architecture?

- **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.)

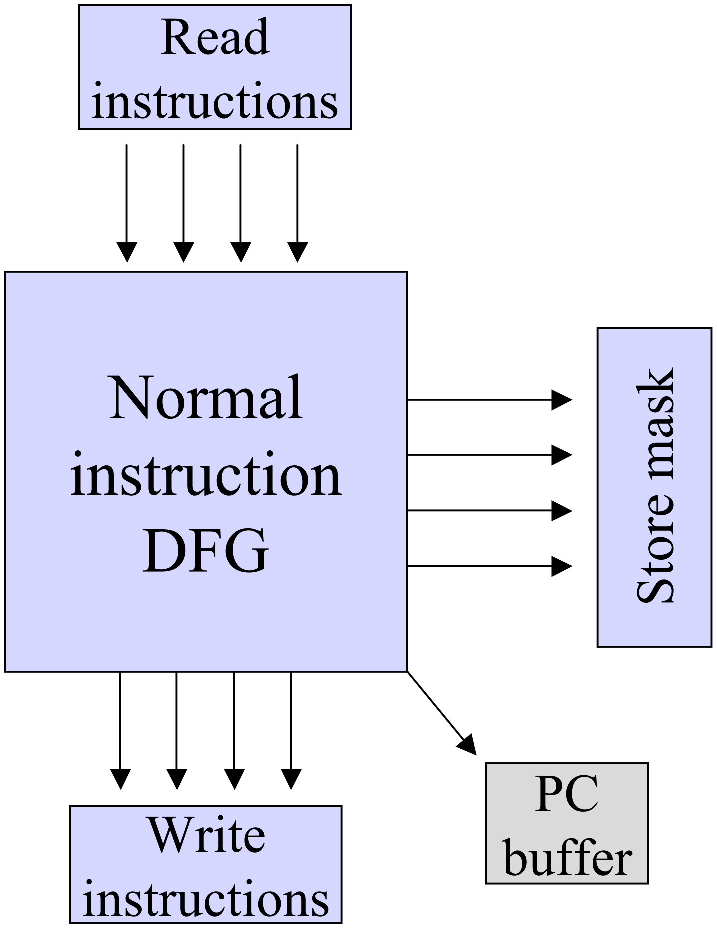
Architectural Structure of a TRIPS Block



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

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 (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

RISC code

```
ld    R3, 4(R2)
add   R4, R1, R3
st    R4, 4(R2)
addi  R5, R4, #2
beqz  R4, Block3
```

TIL (operand format)

```
.bbegin  block1
read     $t1, $g1
read     $t2, $g2
ld       $t3, 4($t2)
add      $t4, $t1, $t3
st       $t4, 4($t2)
addi     $t5, $t4, 2
teqz     $t6, $t4
b_t<$t6> block3
b_f<$t6> block2
write    $g5, $t5
.bend    block1

.bbegin  block2 ...
```

TASL (target format)

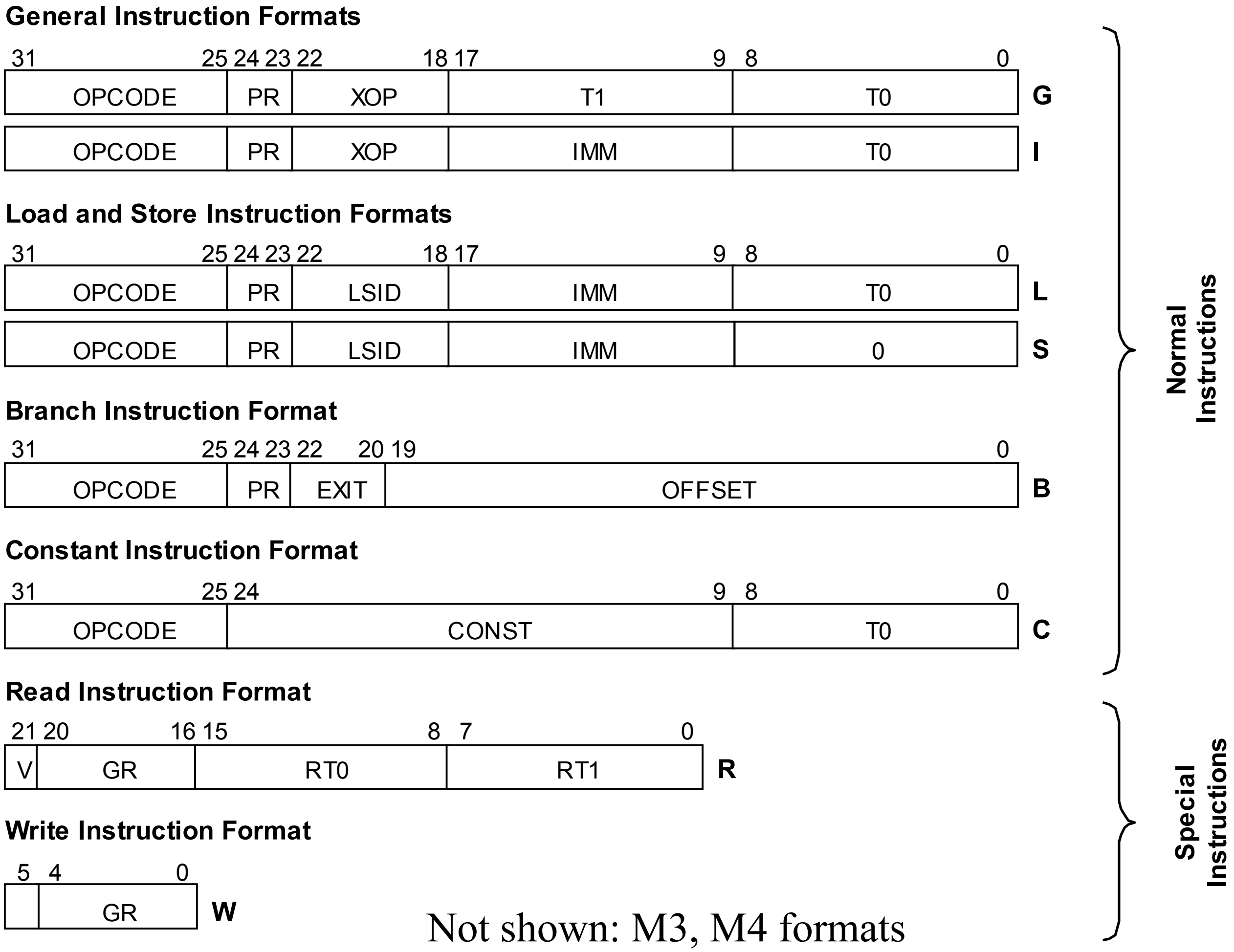
```
[R1] $g1   [2]
[R2] $g2   [1] [4]
[1]  ld    L[1] 4 [2]
[2]  add   [3] [4]
[3]  mov   [5] [6]
[4]  st    S[2] 4
[5]  addi  2 [W1]
[6]  teqz  [7] [8]
[7]  b_t   block3
[8]  b_f   block2
[W1] $g5
```

- 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

31 25	24 23	22 18	17 9	8 0	
OPCODE	PR	XOP	T1	T0	**G**
OPCODE	PR	XOP	IMM	T0	**I**

Load and Store Instruction Formats

31 25	24 23	22 18	17 9	8 0	
OPCODE	PR	LSID	IMM	T0	**L**
OPCODE	PR	LSID	IMM	0	**S**

Branch Instruction Format

31 25	24 23	22 20	19 0	
OPCODE	PR	EXIT	OFFSET	**B**

Constant Instruction Format

31 25	24 9	8 0	
OPCODE	CONST	T0	**C**

Normal Instructions

Read Instruction Format

21	20 16	15 8	7 0	
V	GR	RT0	RT1	**R**

Write Instruction Format

5	4 0	
	GR	**W**

Special Instructions

Not shown: M3, M4 formats

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

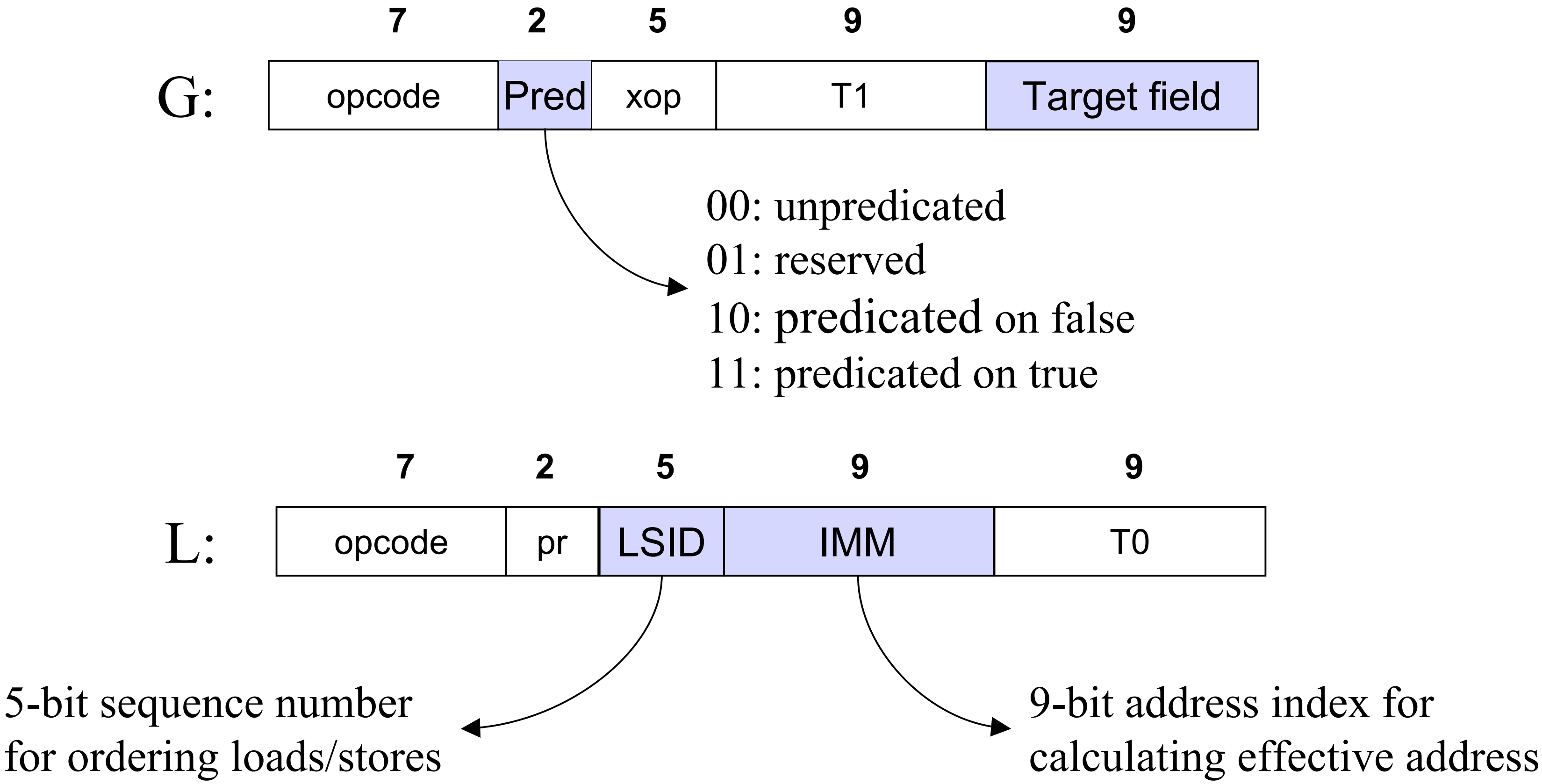
V = Valid Bit

GR = General Register Index

RT0 = Read Target 0 Specifier

RT1 = Read Target 1 Specifier

TRIPS G and L formats



G:

7	2	5	9	9
opcode	Pred	xop	T1	Target field

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

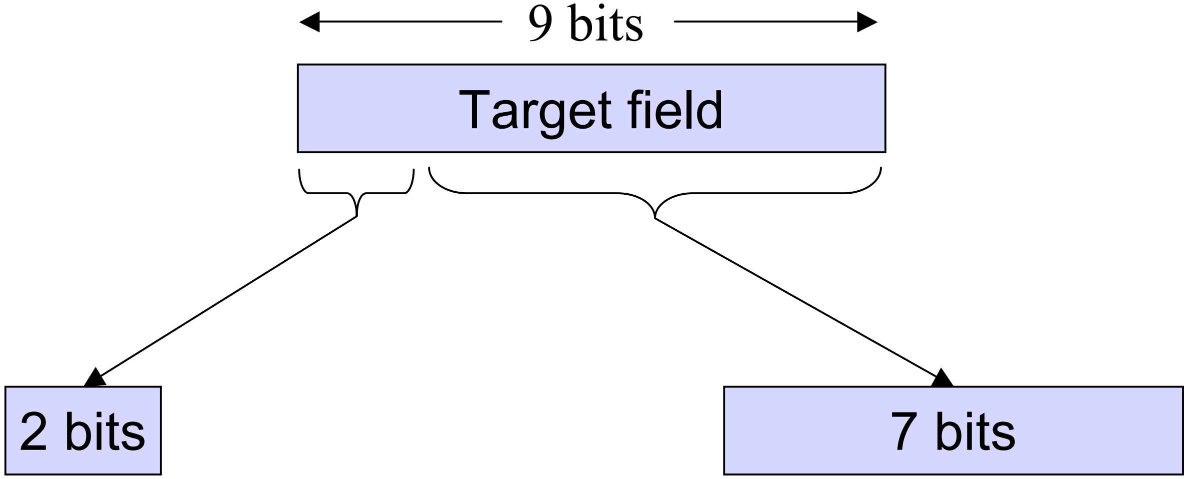
L:

7	2	5	9	9
opcode	pr	LSID	IMM	T0

5-bit sequence number for ordering loads/stores

9-bit address index for calculating effective address

Target (Operand) Formats



9 bits

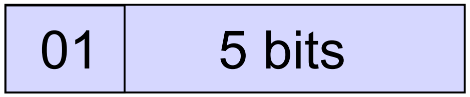
Target field

2 bits

7 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



01

5 bits

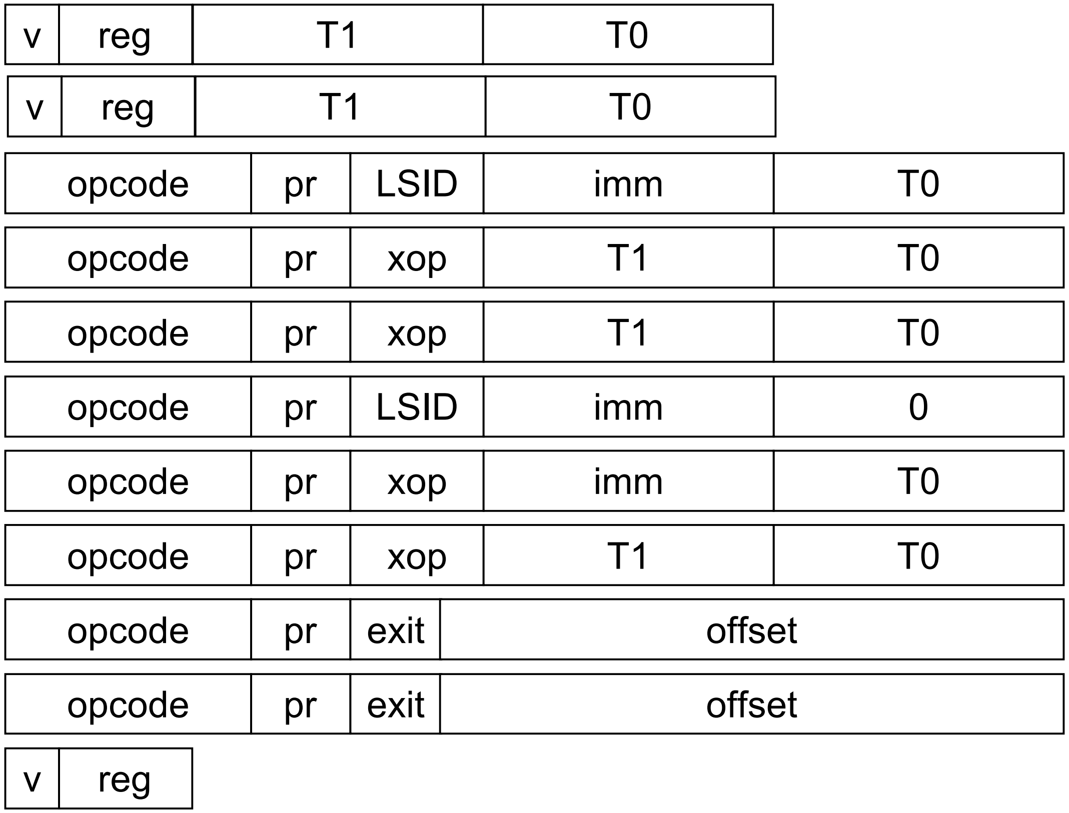
- Names one of 32 write instructions

Object Code Example

TASL (target format)

```
[R1] $g1   [2]
[R2] $g2   [1] [4]
[1]  ld    L[1] 4
[2]  add   [3] [4]
[3]  mov   [5] [6]
[4]  st    S[2] 4
[5]  addi  2 [W1]
[6]  teqz  [7] [8]
[7]  b_t   block3
[8]  b_f   block2
[W1] $g5
```

Object Code



v | reg | T1 | T0

v | reg | T1 | T0

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

v | reg

Object Code Example

TASL (target format)

```
[R1] $g1   [2]
[R2] $g2   [1] [4]
[1]  ld    L[1] 4
[2]  add   [3] [4]
[3]  mov   [5] [6]
[4]  st    S[2] 4
[5]  addi  2 [W1]
[6]  teqz  [7] [8]
[7]  b_t   block3
[8]  b_f   block2
[W1] $g5
```

Object Code

1	00001	[2]	----
1	00010	[1]	[4]

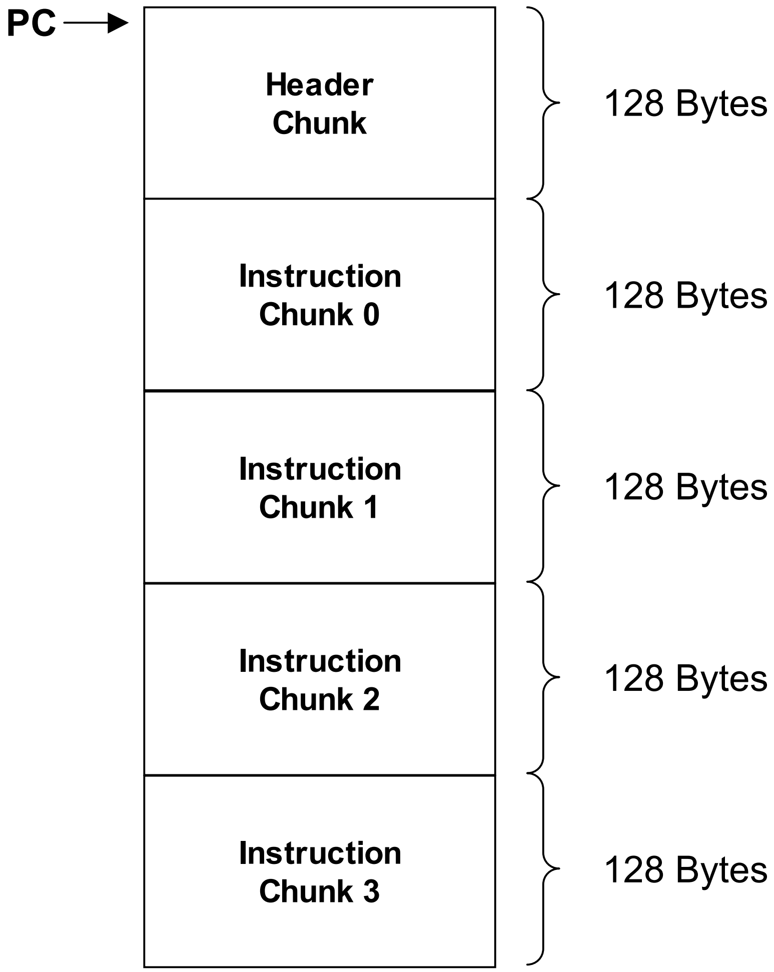
ld	00	00001	4	[2]
add	00	---	[3]	[4]
mov	00	---	[5]	[6]
st	00	00010	4	---
addi	00	---	2	[W1]
teqz	00	---	[7]	[8]

b	11 (t)	E0	block3 - PC
b	10 (f)	E1	block2 - PC

1	00101

Store mask: 00000000000000000000000000000100

TRIPS Block Format

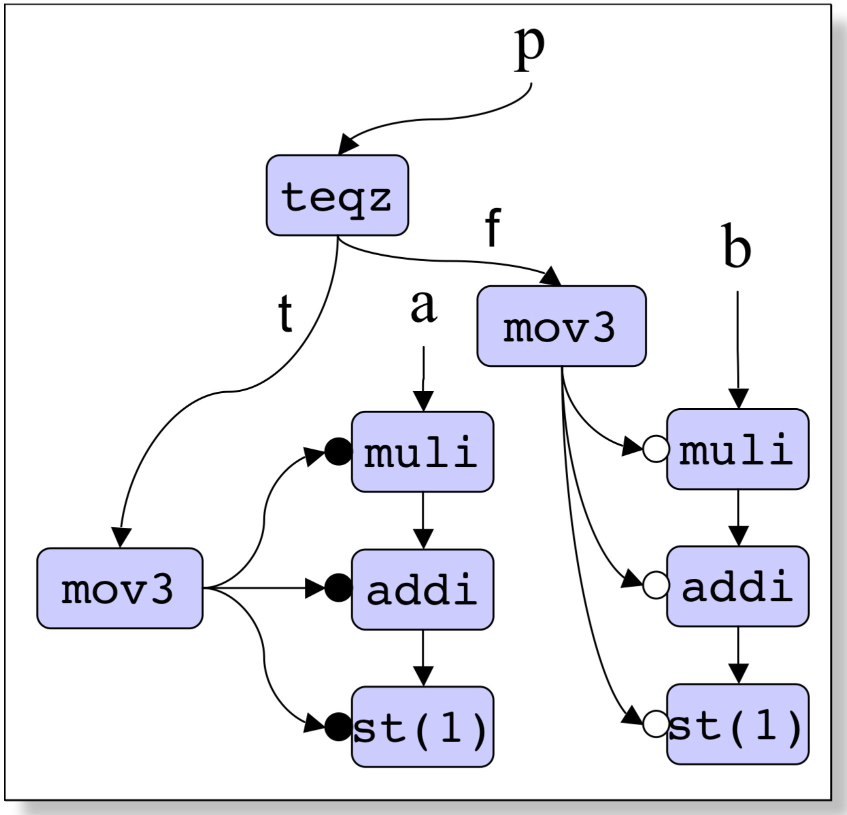


- Each block is formed from two to five 128-byte program“chunks”
- Blocks with fewer than five chunks are expanded to five chunks in the L1 I-cache
- The header chunk includes a block header (execution flags plus a store mask) and register read/write instructions
- Each instruction chunk holds 32 4-byte instructions (including NOPs)
- A maximally sized block contains 128 regular instructions, 32 read instructions, and 32 write instructions

Example of Predicate Operation

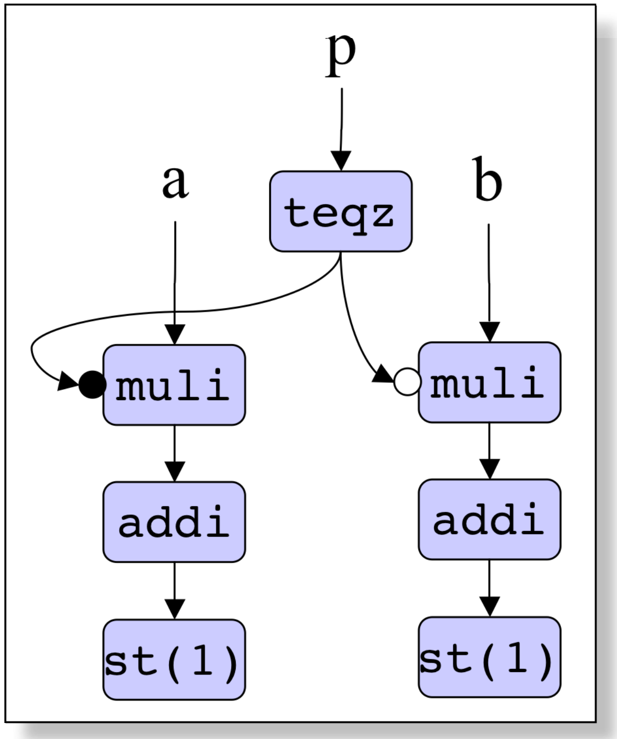
- Test instructions output low-order bit of *1* or *0* (*T* or *F*)
- Predicated instructions either match the arriving predicate or not
  - An ADD with PR=10 is predicated on false, an arriving 0 predicate will match and enable the ADD
  - Predicated instructions must receive all operands and a matching 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;
```

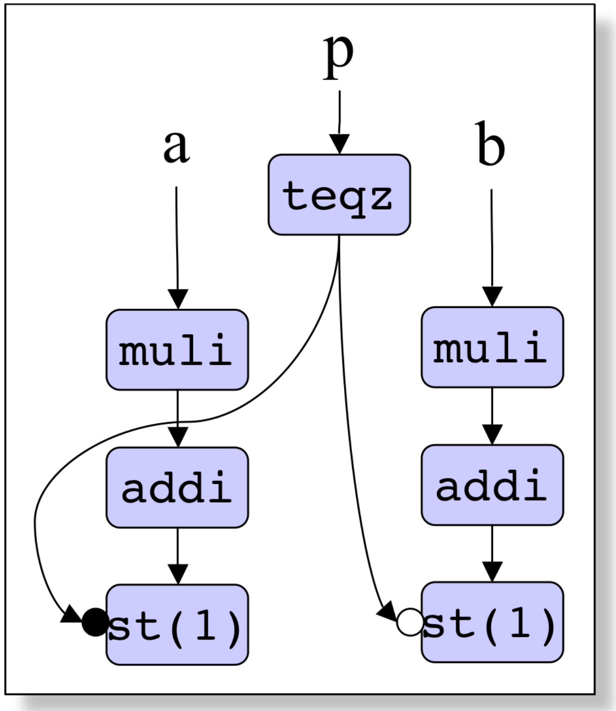


Predicate Optimization and Fanout Reduction

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

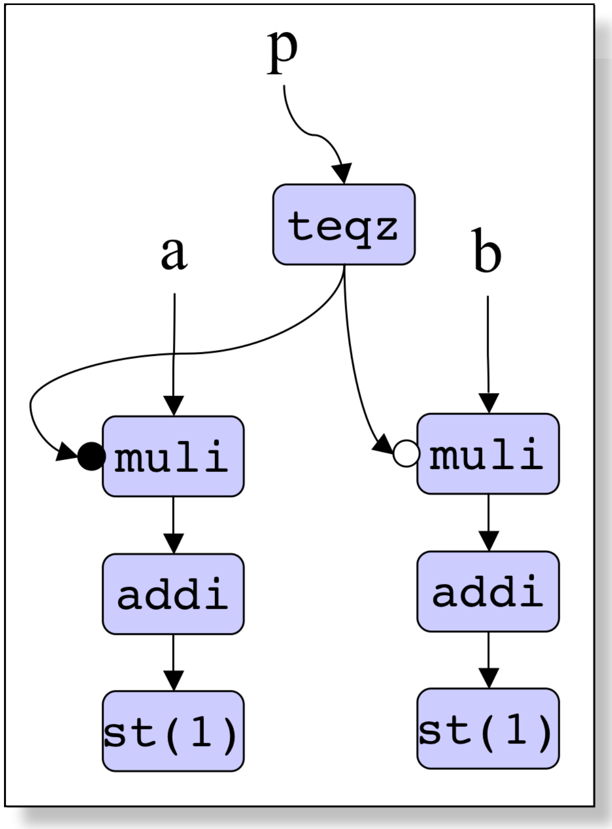


- Predicate dependence tails
  - Tolerate predicate latencies with speculatively executed insts.
  - Cannot speculate on block outputs

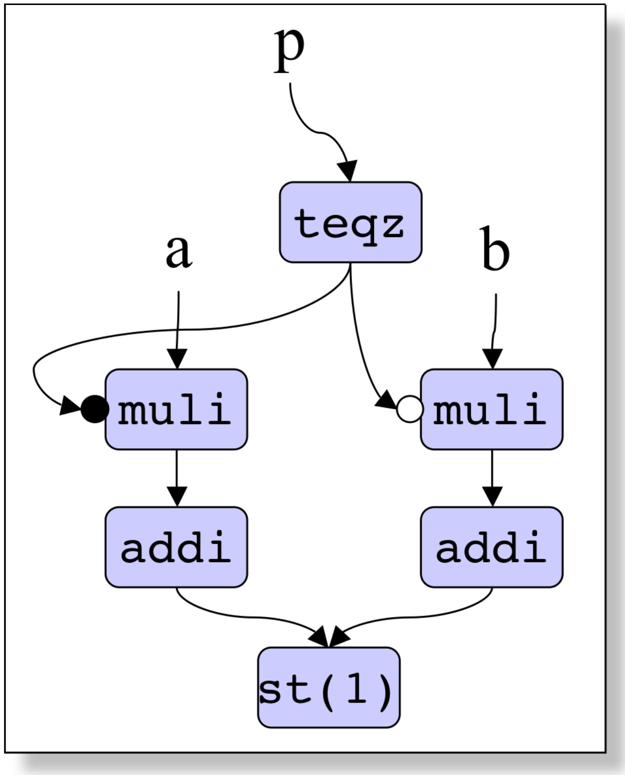


Instruction (Store) Merging

- Old: two copies of stores down distinct predicate paths



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

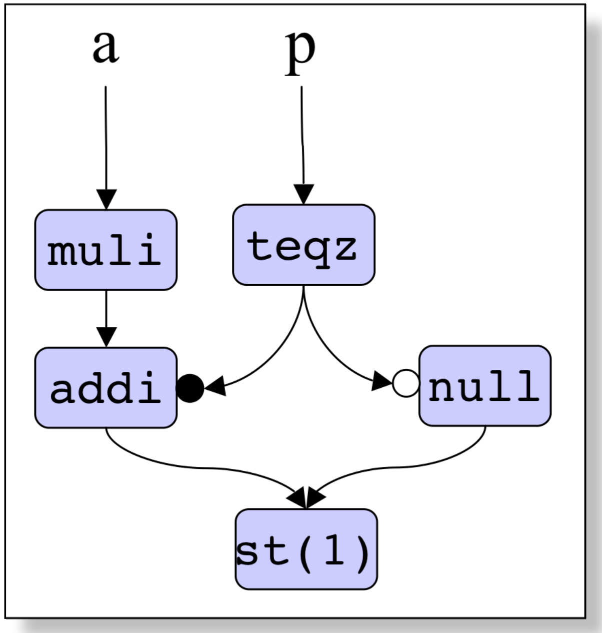


Nullified Outputs

- 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;
...
```



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

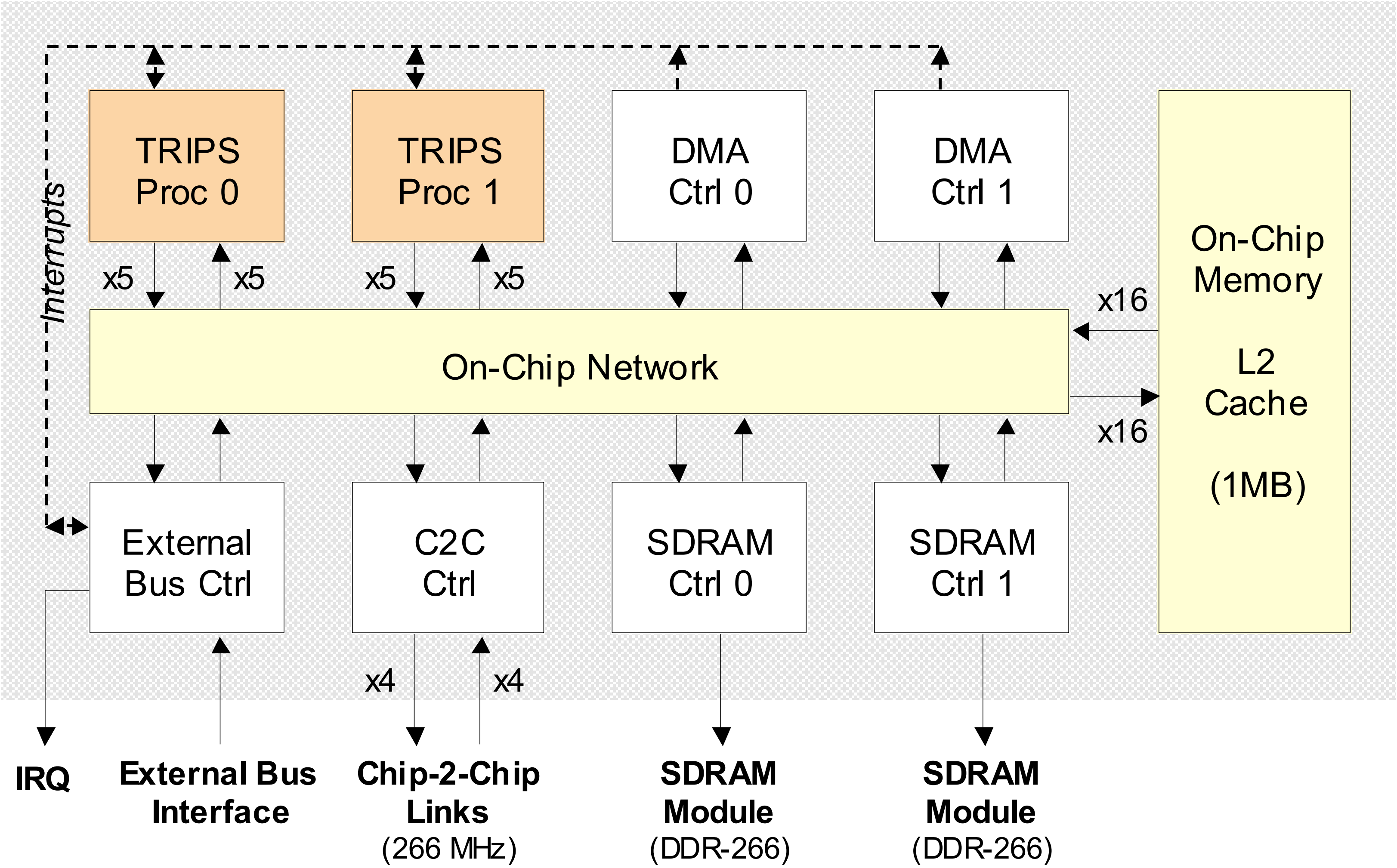
Department of Computer Sciences
The University of Texas at Austin

TRIPS Microarchitecture Principles

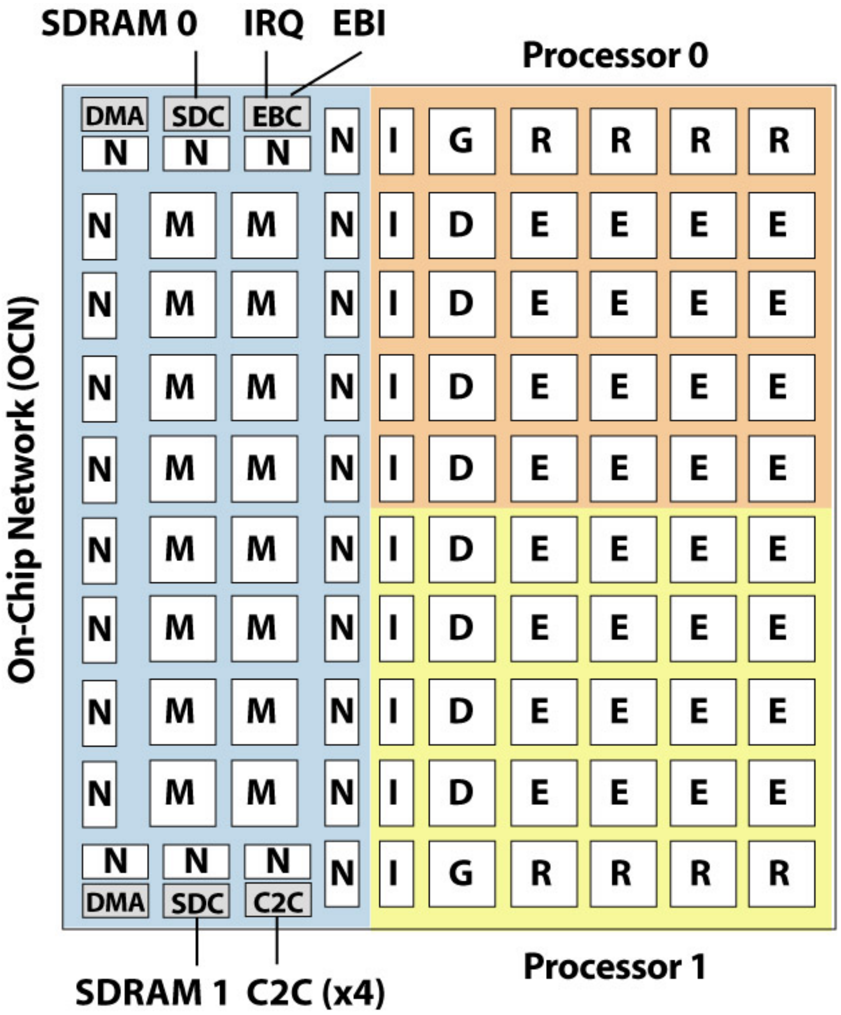
- Limit wire lengths
  - Architecture is partitioned and distributed
    - Microarchitecture composed of different types of tiles
  - No centralized resources
  - Local wires are short
    - Tiles are small (2-5 mm^2 per tile is typical)
  - No global wires
  - Networks connect only nearest neighbors

- 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 Tile-level Microarchitecture



DMA N	SDC N	EBC N	N	I	G	R	R	R	R
N	M	M	N	I	D	E	E	E	E
N	M	M	N	I	D	E	E	E	E
N	M	M	N	I	D	E	E	E	E
N	M	M	N	I	D	E	E	E	E
N	M	M	N	I	D	E	E	E	E
N	M	M	N	I	D	E	E	E	E
N	M	M	N	I	D	E	E	E	E
N	M	M	N	I	D	E	E	E	E
N DMA	N SDC	N C2C	N	I	G	R	R	R	R

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

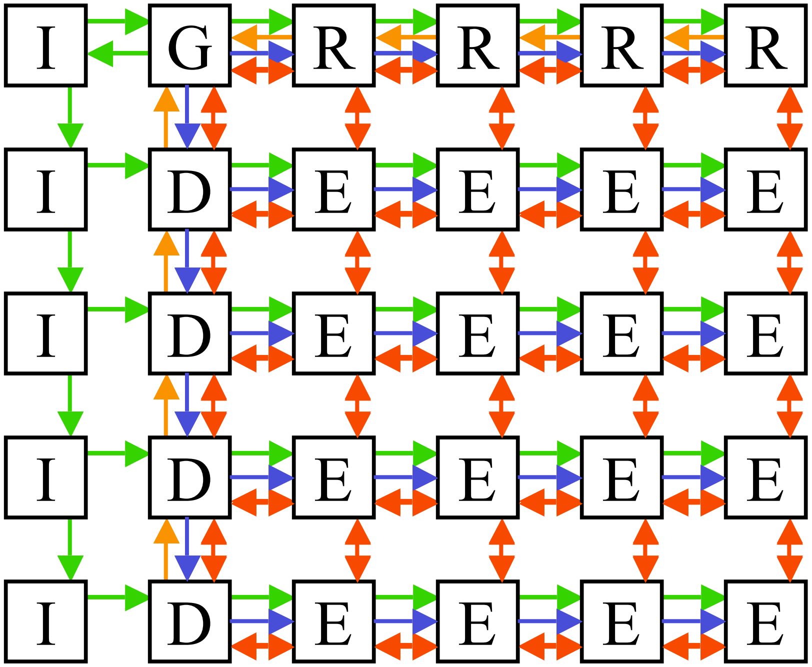
DMA: Direct memory access controller

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



I G R R R R

I D E E E E

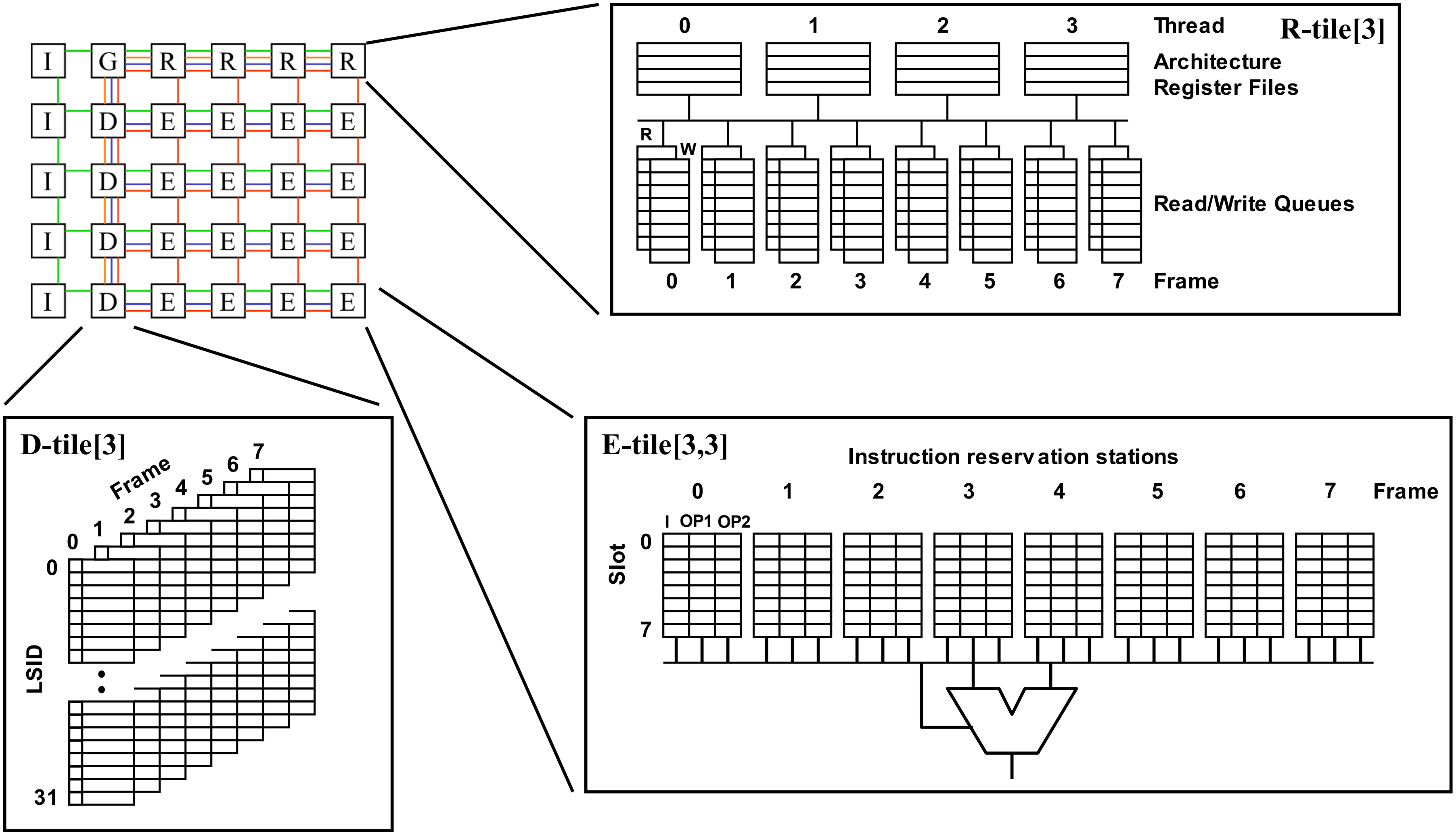
I D E E E E

I D E E E E

I D E E E E

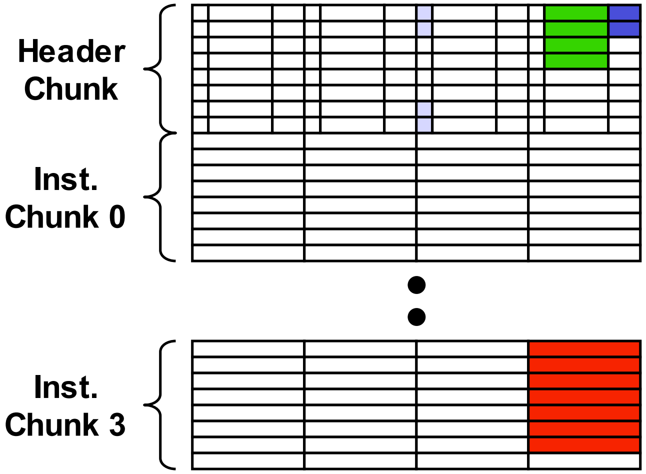
- GDN: global dispatch network
- OPN: operand network
- GSN: global status network
- GCN: global control network

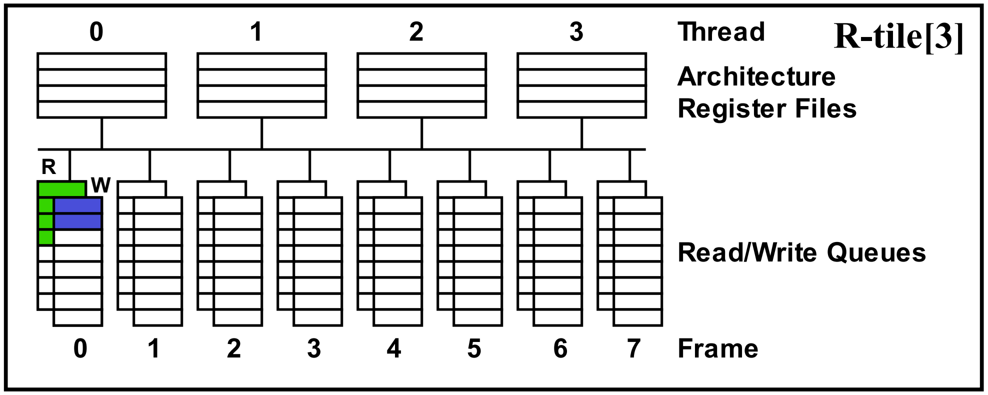
Mapping TRIPS Blocks to the Microarchitecture

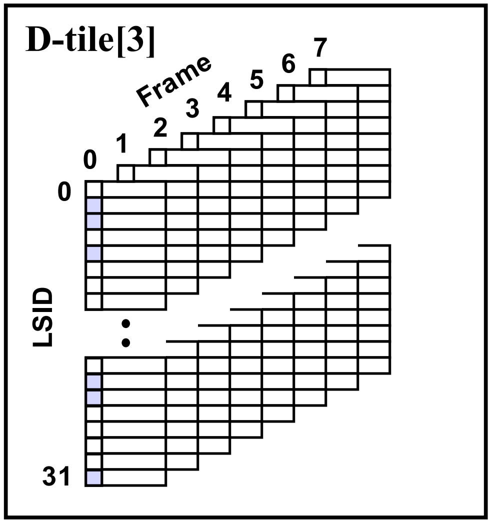


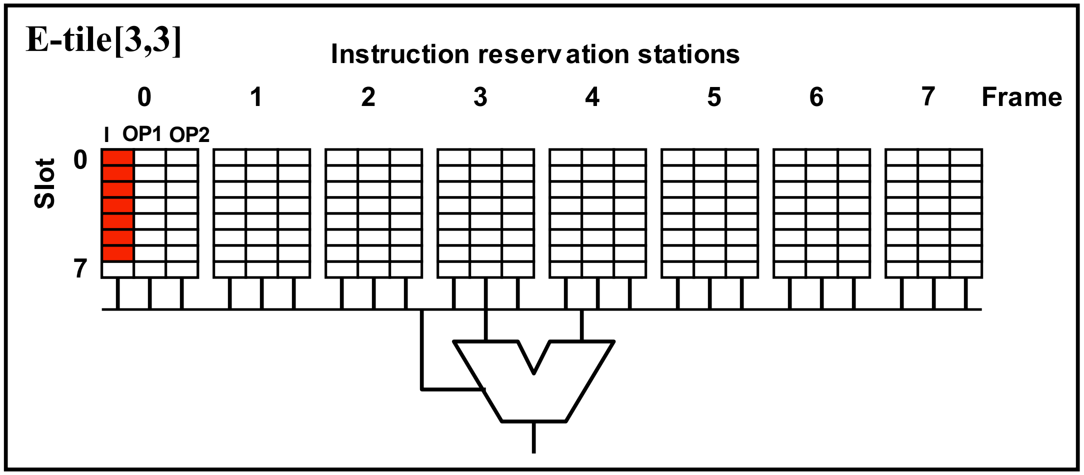
Mapping TRIPS Blocks to the Microarchitecture

Block i mapped into Frame 0

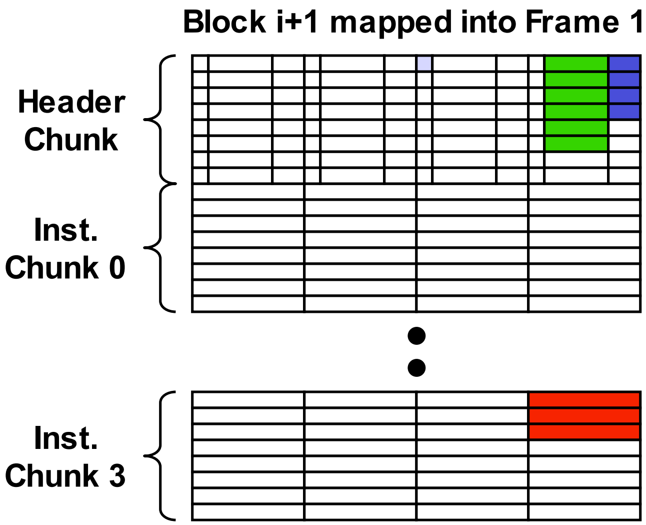


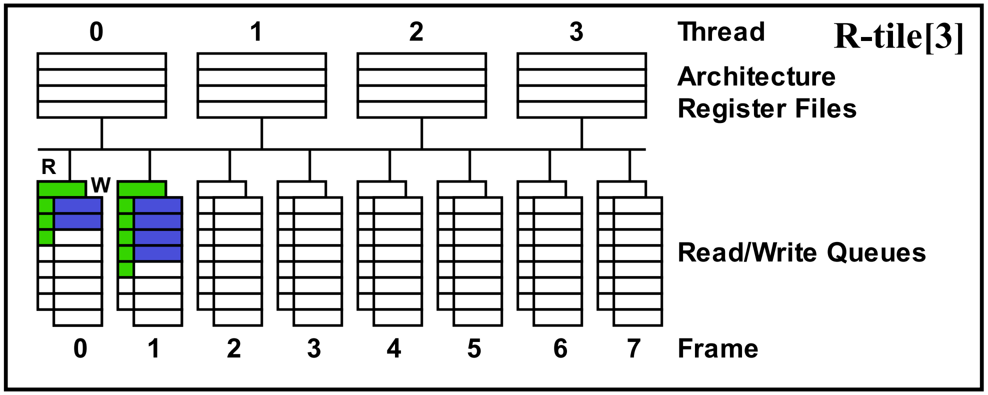


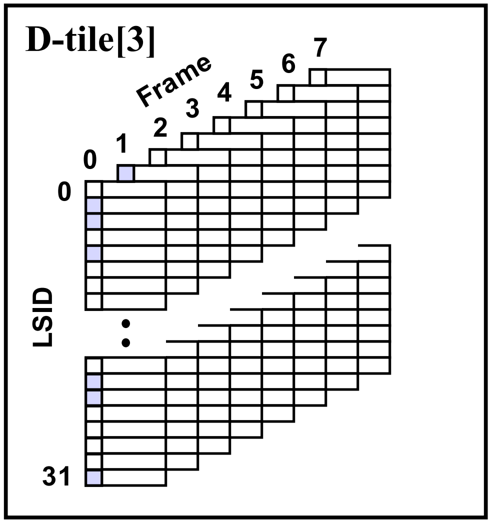


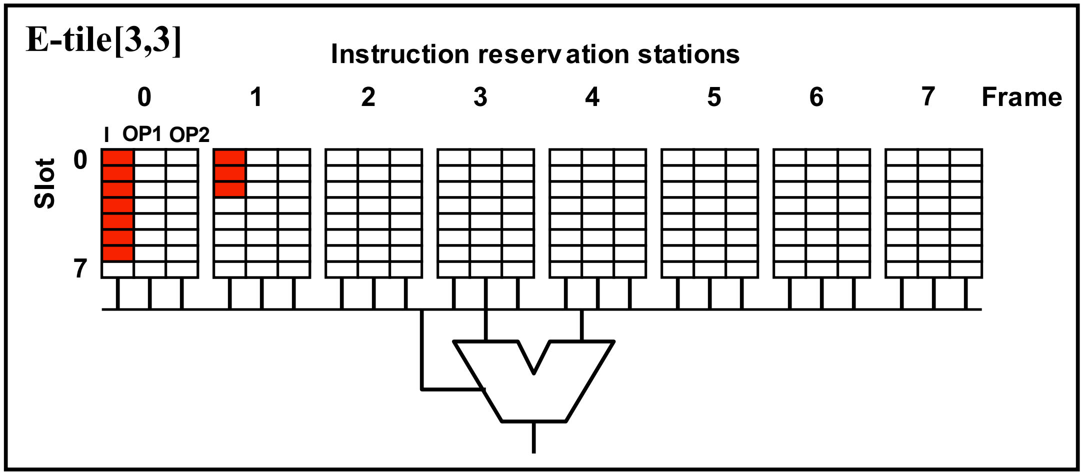


Mapping TRIPS Blocks to the Microarchitecture





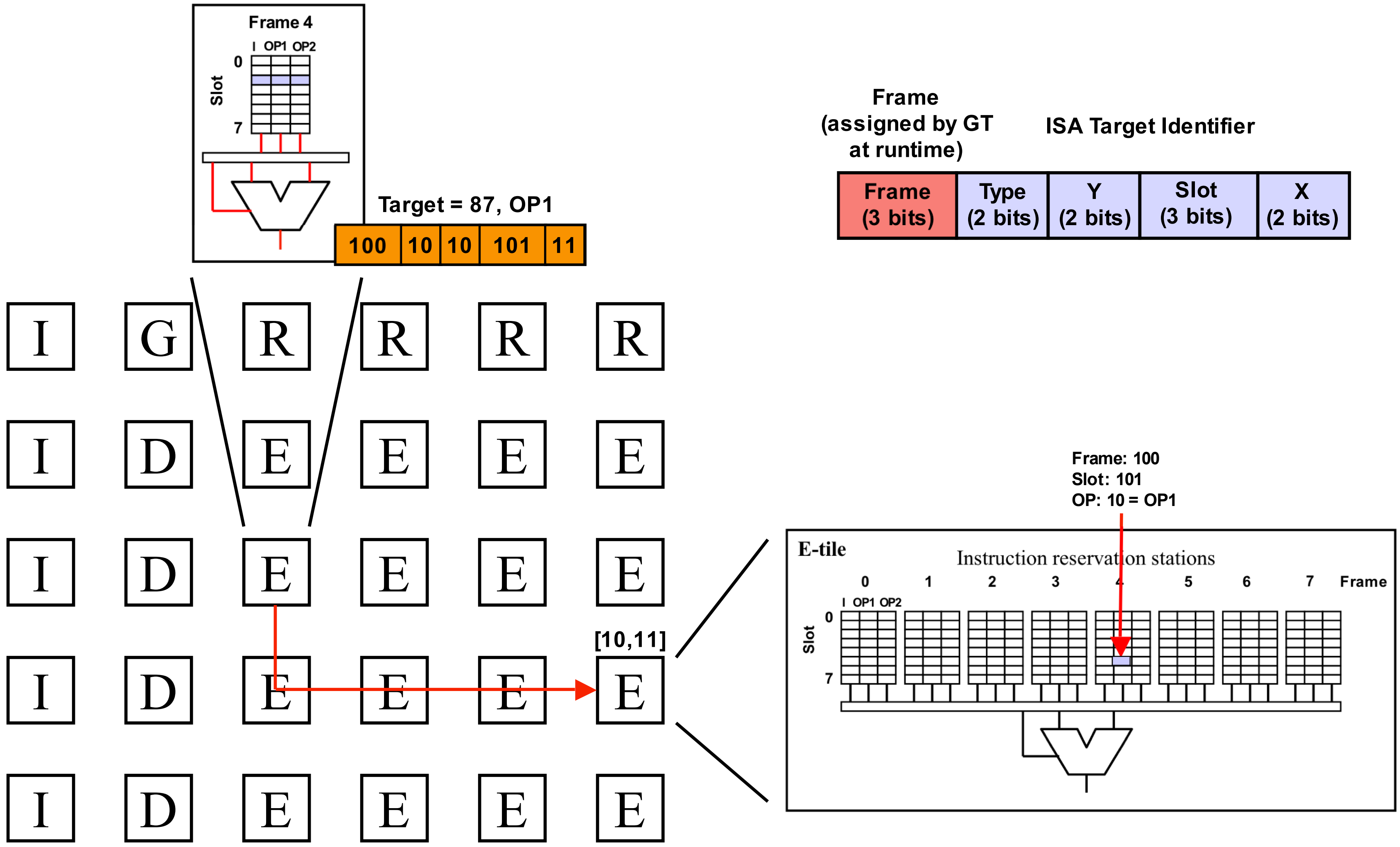




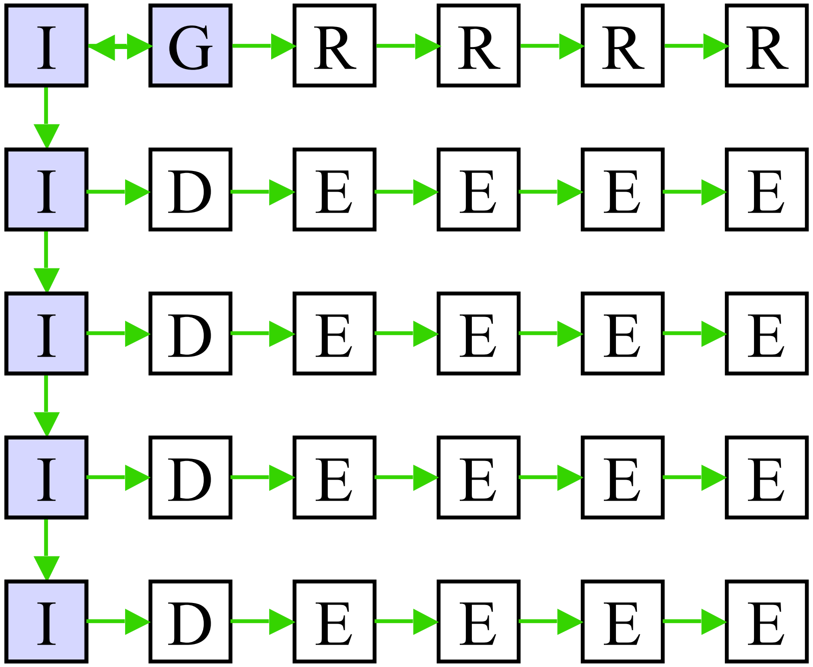
Mapping Target Identifiers to Reservation Stations

Frame (assigned by GT at runtime) | ISA Target Identifier

Frame (3 bits)	Type (2 bits)	Y (2 bits)	Slot (3 bits)	X (2 bits)

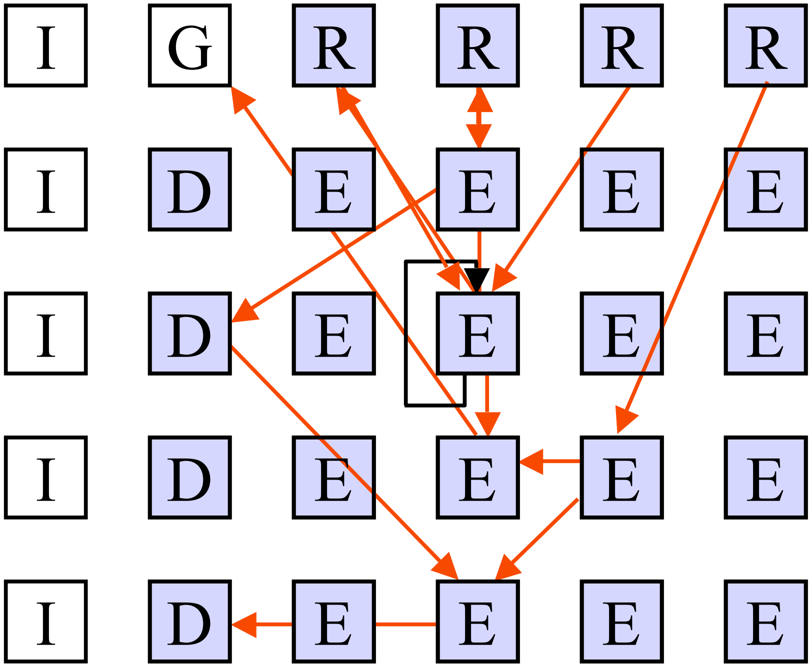


Processor Core Tiles and Interfaces



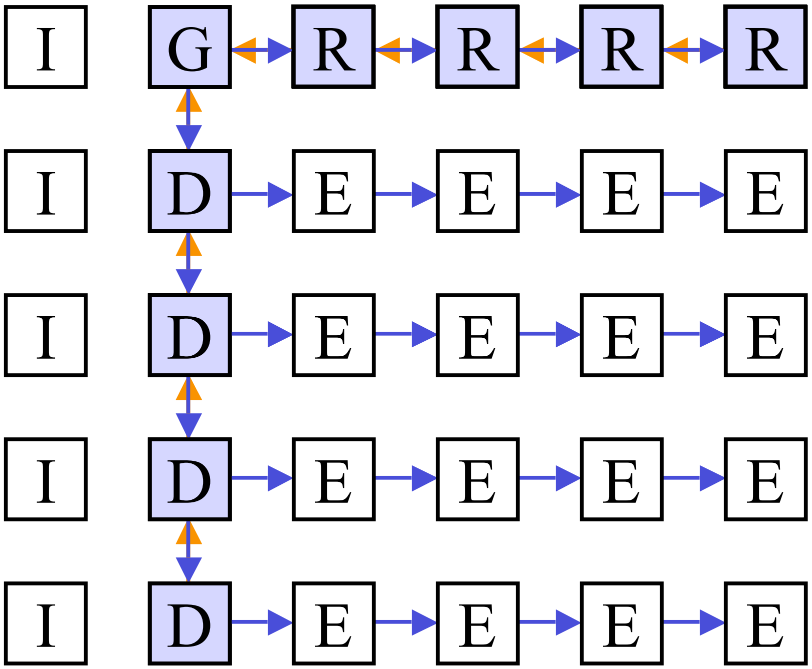
- Fetch
  - G-tile examines I$ tags
  - Sends dispatch command to I-tiles
  - I-tiles fetch instructions, distribute to R-tiles, E-tiles

Processor Core Tiles and Interfaces



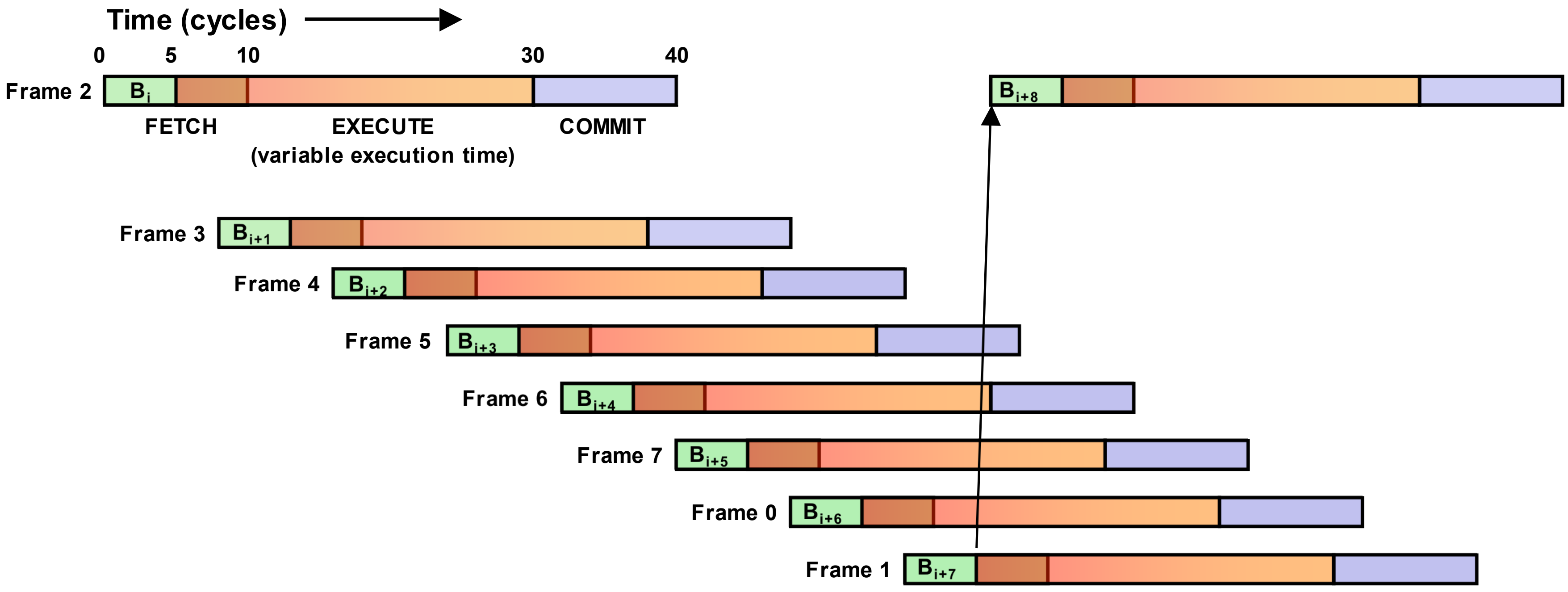
- Execute
  - R-tiles inject register values
  - E-tiles execute block instructions
    - compute, load/store
  - E-tiles delivers outputs
    - Results to R-tiles/D-tiles
    - Branch to G-tile

Processor Core Tiles and Interfaces



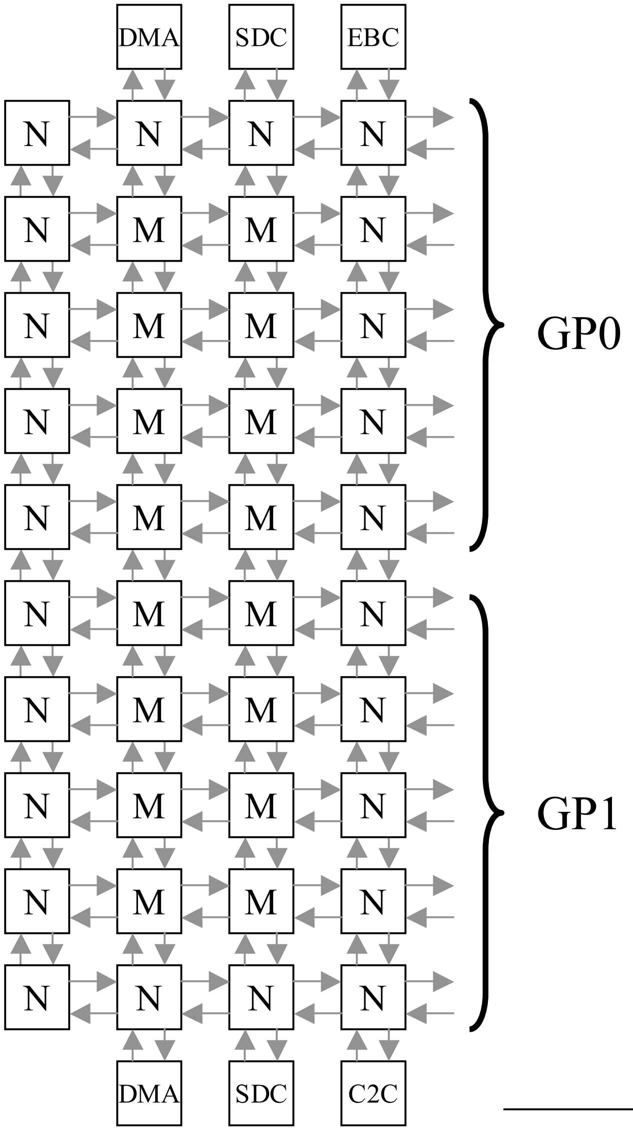
- Commit
  - R-tiles/D-tiles accumulate outputs
    - Send completion messages to G-tile
  - G-tile sends commit message
  - R-tiles/D-tiles respond with commit acknowledge
    - R-tiles commit state to register files
    - D-tiles commit state to data cache

Block Execution Timeline



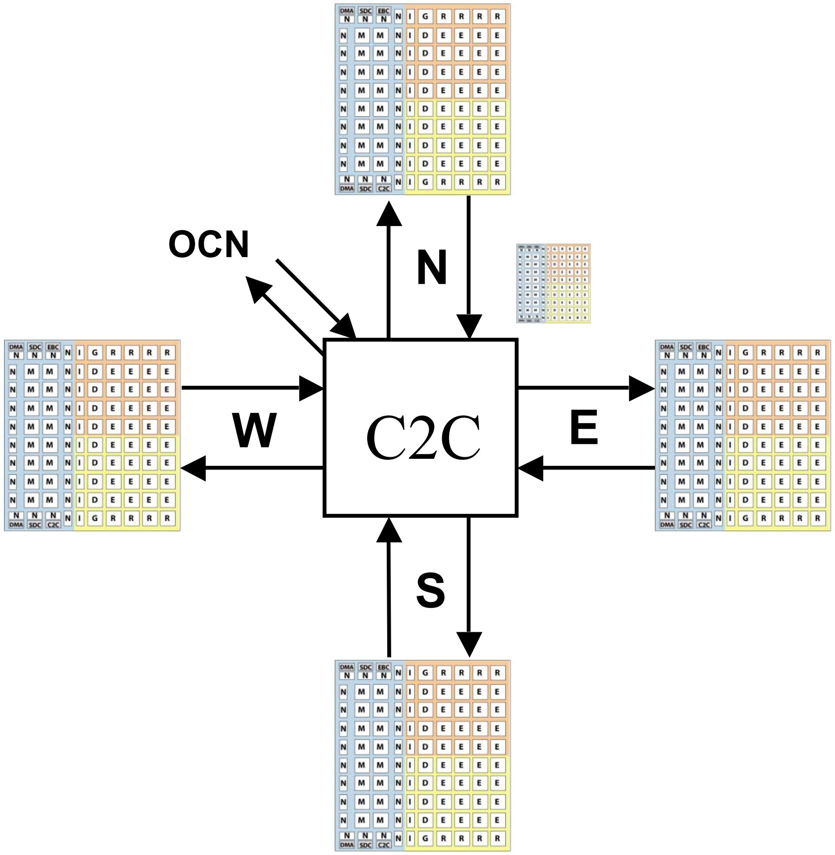
- Execute/commit overlapped across multiple blocks
- G-tile manages frames as a circular buffer
  - D-morph: 1 thread, 8 frames
  - T-morph: up to 4 threads, 2 frames each

On-Chip Network



- 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 ⇒ 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

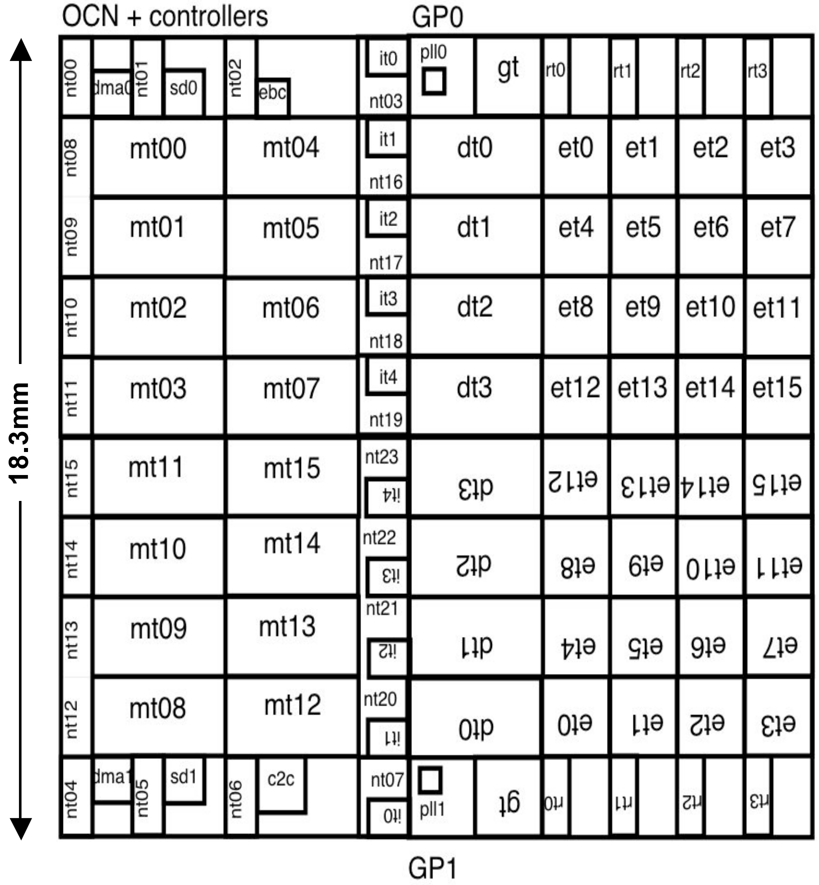
Chip-to-Chip (C2C) Network



- 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


OCN + controllers
GP0
18.3mm
nt00 dma0 nt01 sd0 nt02 ebc it0 nt03 pll0 gt rt0 rt1 rt2 rt3
nt08 mt00 mt04 it1 nt16 dt0 et0 et1 et2 et3
nt09 mt01 mt05 it2 nt17 dt1 et4 et5 et6 et7
nt10 mt02 mt06 it3 nt18 dt2 et8 et9 et10 et11
nt11 mt03 mt07 it4 nt19 dt3 et12 et13 et14 et15
nt15 mt11 mt15 nt23 it4 dt3 et12 et13 et14 et15
nt14 mt10 mt14 nt22 it3 dt2 et8 et9 et10 et11
nt13 mt09 mt13 nt21 it2 dt1 et4 et5 et6 et7
nt12 mt08 mt12 nt20 it1 dt0 et0 et1 et2 et3
nt04 dma1 nt05 sd1 nt06 c2c nt07 it0 pll1 gt rt0 rt1 rt2 rt3
GP1

TRIPS Working Floorplan (to scale)

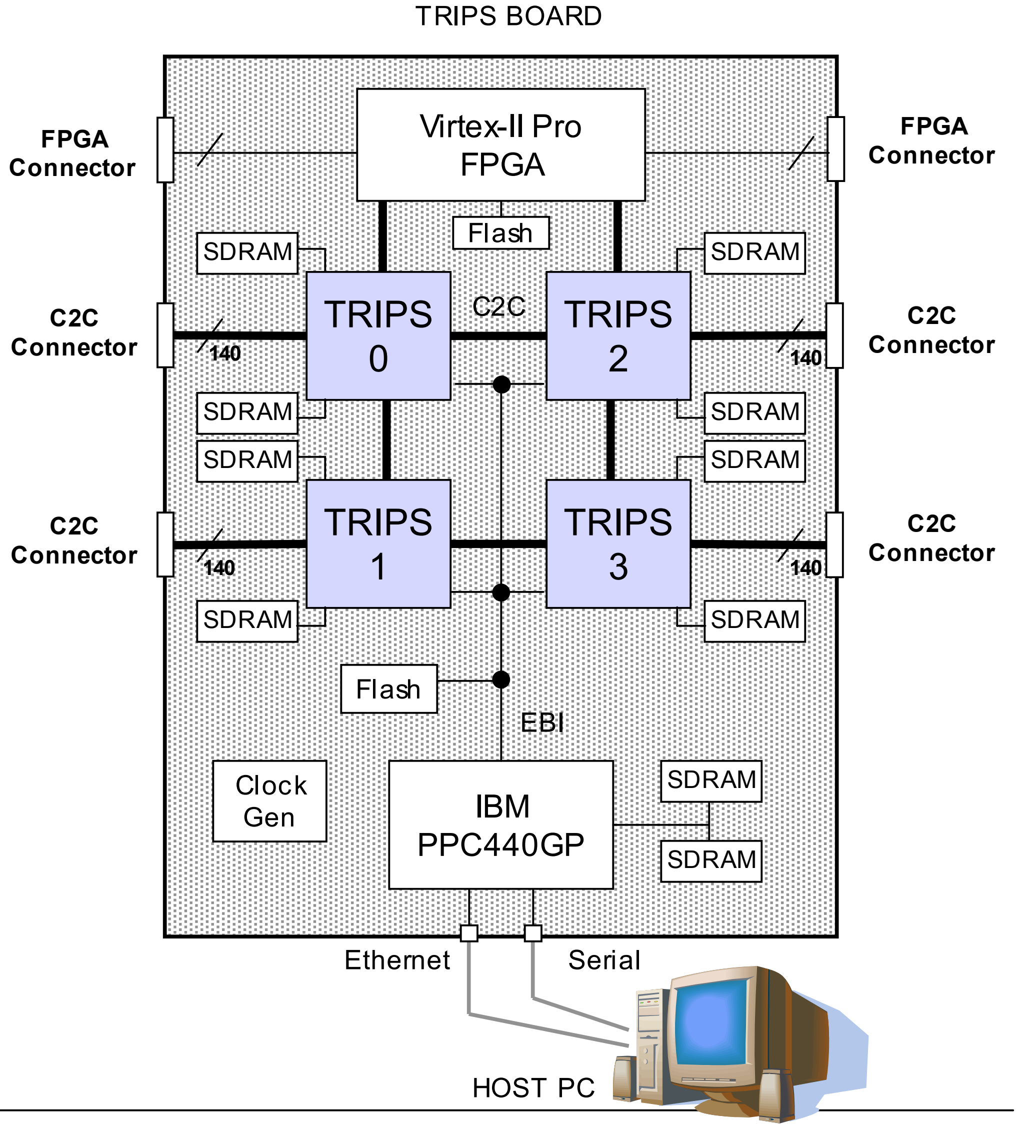
Prototype Simplifications

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

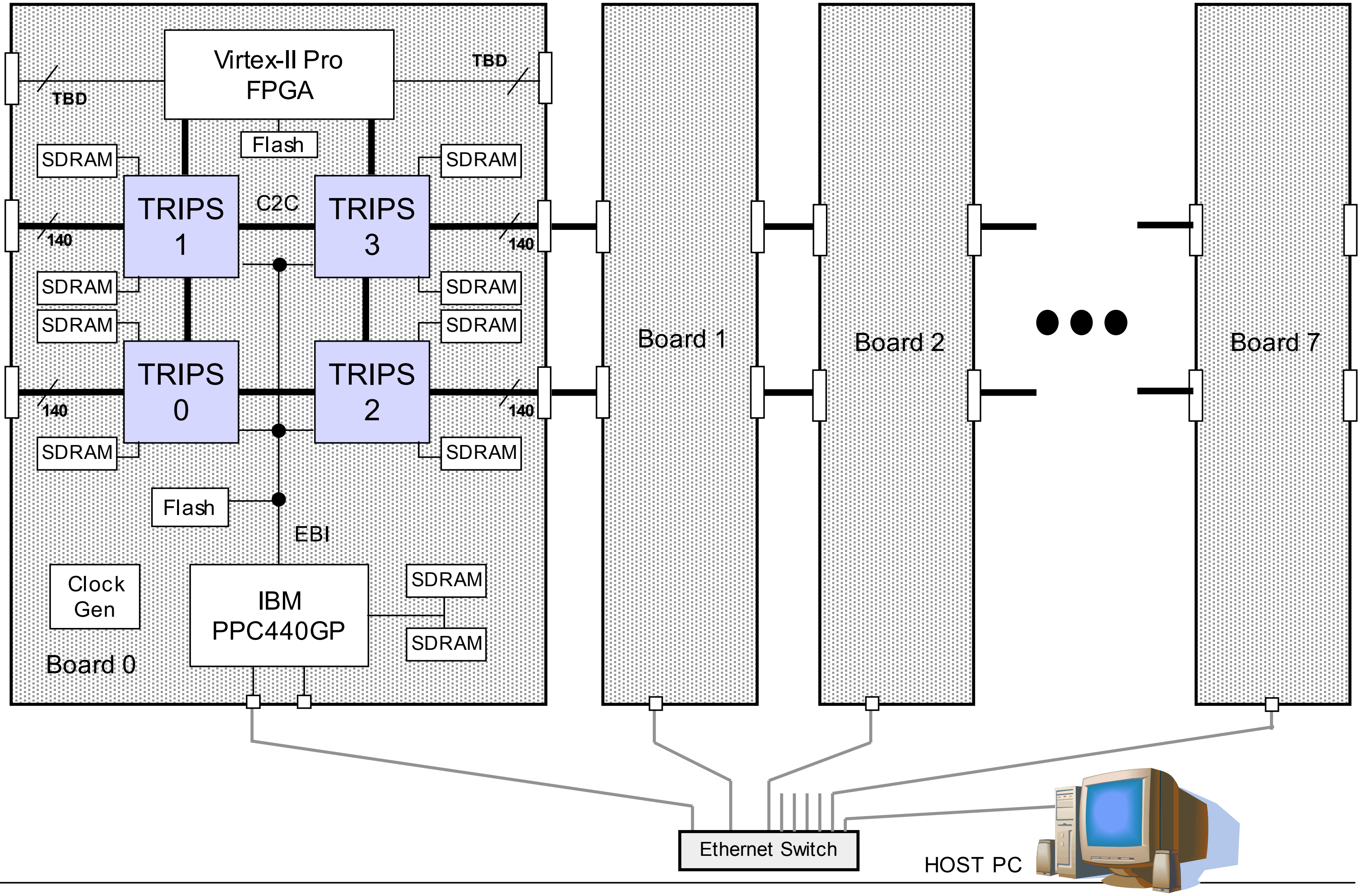
- Microarchitecture simplifications
  - No I-cache prefetching or block predict for I-cache refill
  - No variable resource allocation (for smaller blocks)
  - One cycle per hop on every network
  - No clock-gating
  - Direct-mapped and replicated LSQ (non-associative, oversized)
  - No selective re-execution
  - No fast block refetch

TRIPS Prototype System Board

- 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

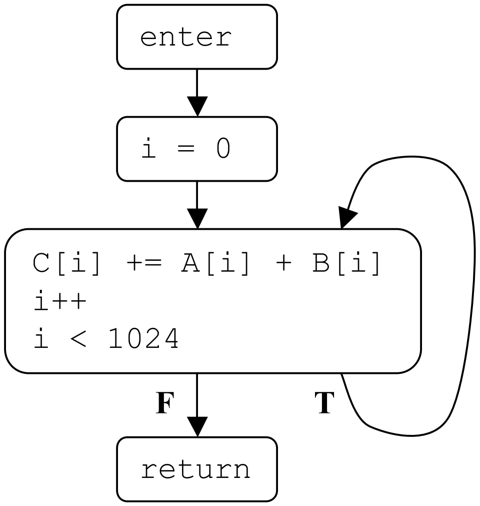
Two Examples

- 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

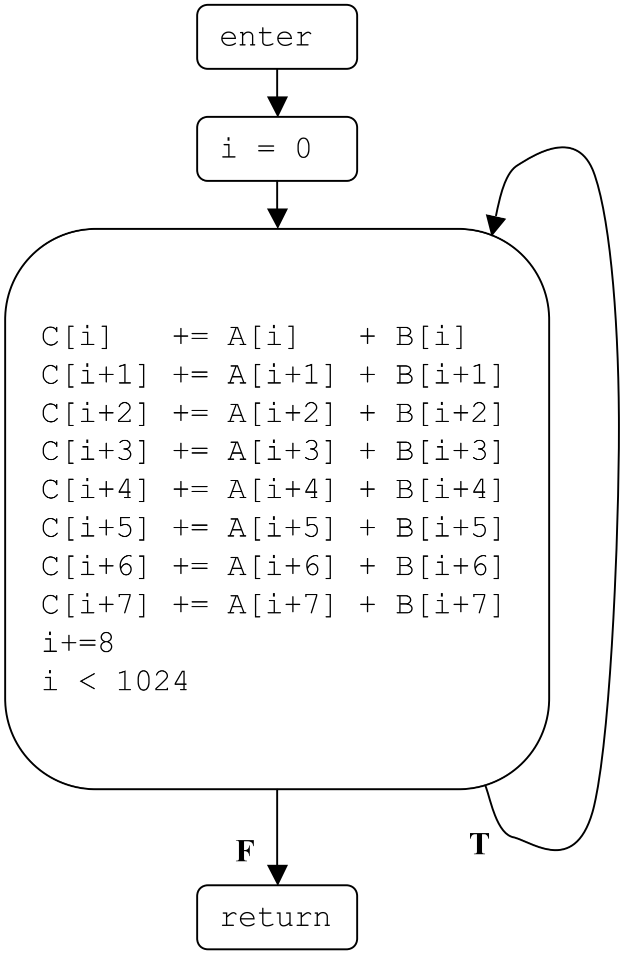
Vector Add – C Code

```
#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] );
  }
}
```



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

Vector Add – Unrolled



```
C[i]   += A[i]   + B[i]
C[i+1] += A[i+1] + B[i+1]
C[i+2] += A[i+2] + B[i+2]
C[i+3] += A[i+3] + B[i+3]
C[i+4] += A[i+4] + B[i+4]
C[i+5] += A[i+5] + B[i+5]
C[i+6] += A[i+6] + B[i+6]
C[i+7] += A[i+7] + B[i+7]
i+=8
i < 1024
```

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

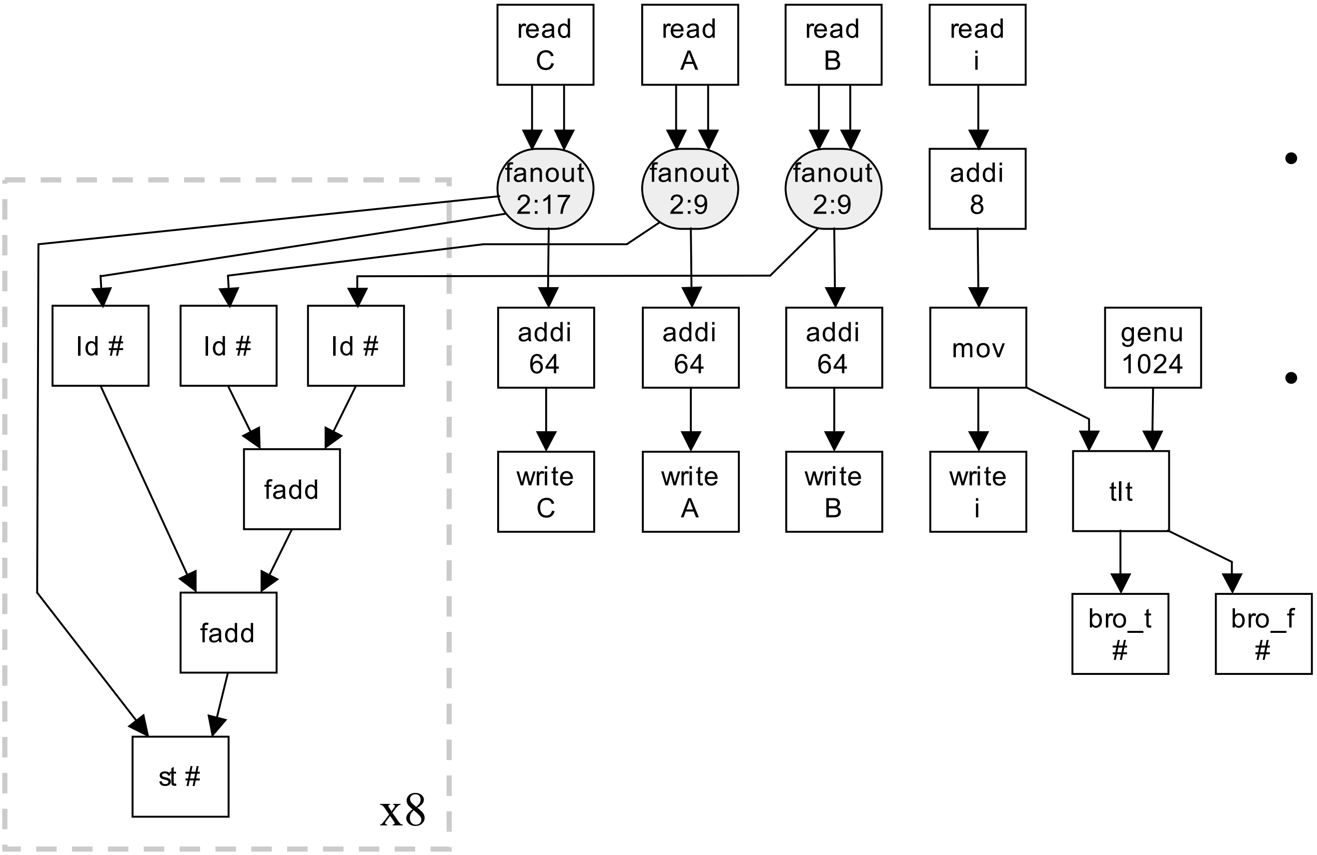
Vector Add – TIL Code

```
.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)
```

```
  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
```

- 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



(abbreviated graph)

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

Vector Add – Placed Instructions

ET0: ld, mov3, mov3, mov, mov, mov	ET1: mov3, ld, mov, mov, mov, mov	ET2: mov, mov3, ld, addi, addi, mov	ET3: mov, mov, mov, mov3, ld
ET4: ld, ld, ld, mov3, addi	ET5: ld, mov3, mov3, fadd, sd	ET6: ld, fadd, ld, ld, ld	ET7: ld, genu, addi
ET8: ld, ld, ld, mov3, fadd, sd	ET9: ld, ld, ld, ld, fadd	ET10: ld, fadd, fadd, sd	ET11: fadd, fadd, sd, tlt, bro_t
ET12: ld, fadd, fadd, sd	ET13: ld, fadd, fadd, sd	ET14: fadd, fadd, sd, ld	ET15: fadd, fadd, sd, bro_f

- 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)

Vector Add – TASL Code

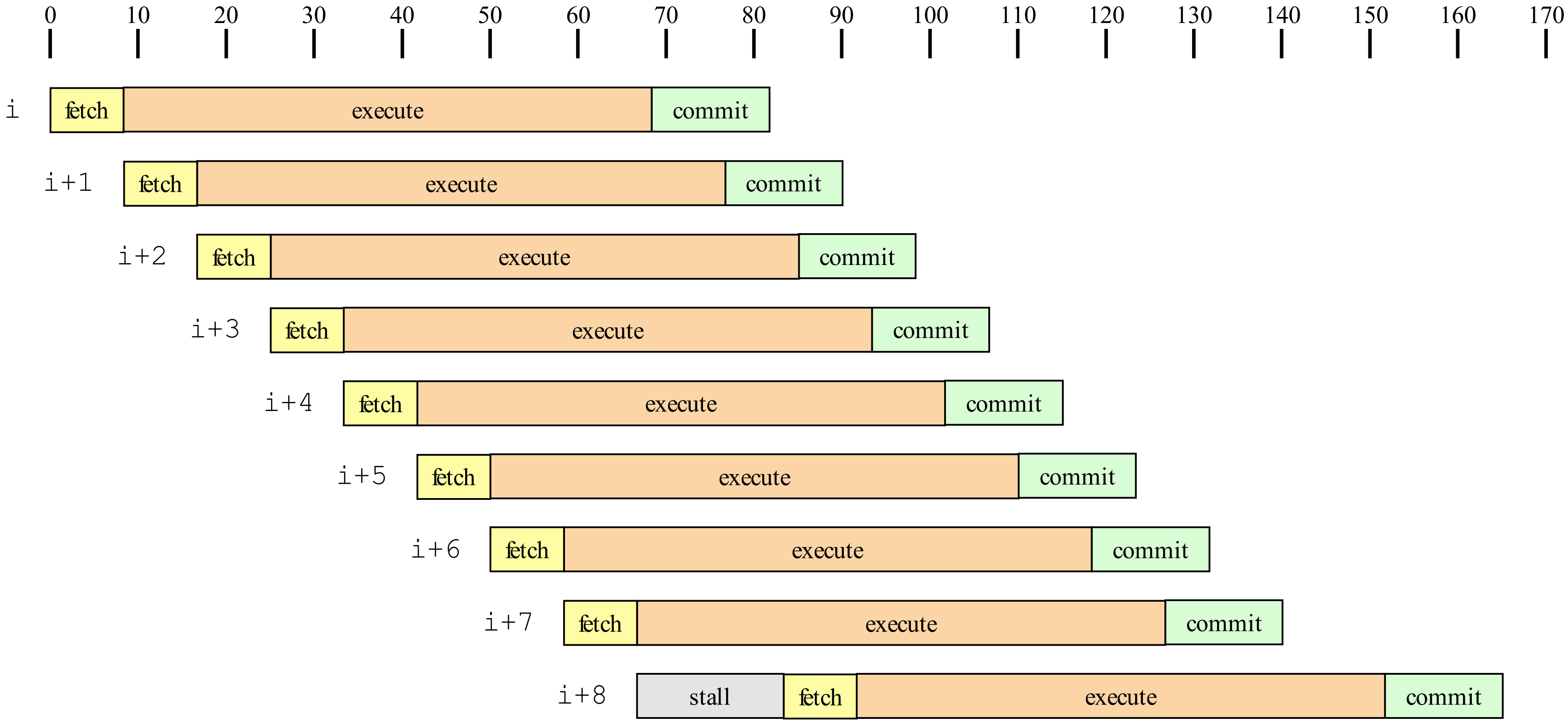
```
.bbegin vadd$1
R[ 2]    read G[ 70]  N[ 14,0]
R[ 3]    read G[ 71]  N[ 43,0]  N[ 3,0]
R[ 0]    read G[ 72]  N[ 48,0]  N[ 12,0]
R[ 1]    read G[ 73]  N[ 18,0]  N[ 13,0]

N[ 3]    mov N[ 7,0]  N[ 9,0]
N[ 7]    mov N[ 11,0]  N[ 2,0]
N[ 9]    mov N[ 4,0]  N[ 37,0]
N[ 11]   mov N[ 15,0]  N[ 6,0]
N[ 2]    mov N[ 1,0]  N[ 75,0]
N[ 4]    mov3 N[ 32,0]  N[ 78,0]  N[ 64,0]
N[ 37]   mov3 N[ 49,0]  N[ 65,0]  N[ 84,0]
N[ 15]   mov3 N[ 19,0]  N[ 109,0]  N[ 10,0]
N[ 6]    mov3 N[ 108,0]  N[ 5,0]  N[ 106,0]
N[ 1]    mov3 N[ 0,0]  N[ 107,0]  N[ 33,0]
N[ 12]   mov N[ 16,0]  N[ 76,0]
N[ 16]   mov N[ 44,0]  N[ 21,0]
N[ 76]   mov3 N[ 66,0]  N[ 77,0]  N[ 110,0]
N[ 44]   mov3 N[ 72,0]  N[ 96,0]  N[ 73,0]
N[ 21]   mov N[ 35,0]  N[ 42,0]
N[ 13]   mov N[ 17,0]  N[ 41,0]
N[ 17]   mov N[ 8,0]  N[ 20,0]
N[ 41]   mov3 N[ 69,0]  N[ 97,0]  N[ 34,0]
N[ 8]    mov3 N[ 36,0]  N[ 68,0]  N[ 40,0]
N[ 20]   mov N[ 46,0]  N[ 50,0]
N[ 32]   ld L[ 0]   N[ 74,0]
N[ 66]   ld L[ 1]   N[ 70,0]
N[ 69]   ld L[ 2]   N[ 70,1]
N[ 70]   fadd N[ 74,1]
N[ 74]   fadd N[ 78,1]
N[ 78]   sd S[ 3]
N[ 64]   ld L[ 4]  8 N[ 45,0]
N[ 77]   ld L[ 5]  8 N[ 38,0]
(cont)
```

```
N[ 97]   ld L[ 6]  8 N[ 38,1]
N[ 38]   fadd N[ 45,1]
N[ 45]   fadd N[ 49,1]
N[ 49]   sd S[ 7]  8
N[ 65]   ld L[ 8]  16 N[ 80,0]
N[ 110]  ld L[ 9]  16 N[ 81,0]
N[ 34]   ld L[ 10]  16 N[ 81,1]
N[ 81]   fadd N[ 80,1]
N[ 80]   fadd N[ 84,1]
N[ 84]   sd S[ 11]  16
N[ 19]   ld L[ 12]  24 N[ 105,0]
N[ 72]   ld L[ 13]  24 N[ 101,0]
N[ 36]   ld L[ 14]  24 N[ 101,1]
N[ 101]  fadd N[ 105,1]
N[ 105]  fadd N[ 109,1]
N[ 109]  sd S[ 15]  24
N[ 10]   ld L[ 16]  32 N[ 104,0]
N[ 96]   ld L[ 17]  32 N[ 100,0]
N[ 68]   ld L[ 18]  32 N[ 100,1]
N[ 100]  fadd N[ 104,1]
N[ 104]  fadd N[ 108,1]
N[ 108]  sd S[ 19]  32
N[ 5]    ld L[ 20]  40 N[ 102,0]
N[ 73]   ld L[ 21]  40 N[ 98,0]
N[ 40]   ld L[ 22]  40 N[ 98,1]
N[ 98]   fadd N[ 102,1]
N[ 102]  fadd N[ 106,1]
N[ 106]  sd S[ 23]  40
N[ 0]    ld L[ 24]  48 N[ 103,0]
N[ 35]   ld L[ 25]  48 N[ 99,0]
N[ 46]   ld L[ 26]  48 N[ 99,1]
N[ 99]   fadd N[ 103,1]
N[ 103]  fadd N[ 107,1]
N[ 107]  sd S[ 27]  48
(cont)
```

```
N[ 33]   ld L[ 28]  56 N[ 71,0]
N[ 42]   ld L[ 29]  56 N[ 67,0]
N[ 50]   ld L[ 30]  56 N[ 67,1]
N[ 67]   fadd N[ 71,1]
N[ 71]   fadd N[ 75,1]
N[ 75]   sd S[ 31]  56
N[ 14]   addi 8 N[ 22,0]
N[ 22]   mov N[ 79,0]  W[ 2]
N[ 43]   addi 64 W[ 3]
N[ 48]   addi 64 W[ 0]
N[ 18]   addi 64 W[ 1]
N[ 39]   genu 1017 N[ 79,1]
N[ 79]   tlt N[ 83,p]  N[ 111,p]
N[ 83]   bro_t vadd$1
N[ 111]  bro_f vadd$2

W[ 2]    write G[ 70]
W[ 3]    write G[ 71]
W[ 0]    write G[ 72]
W[ 1]    write G[ 73]
.bend
```

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

Vector Add – Block-Level Execution



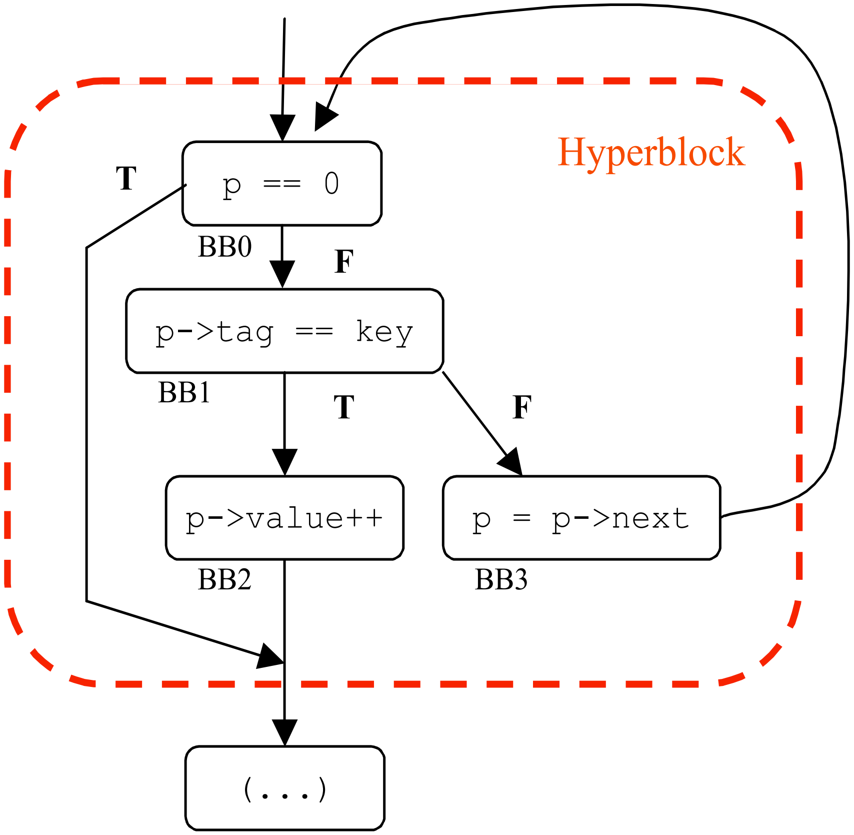
- The TRIPS processor can execute up to 8 blocks concurrently
- This diagram shows block latency and throughput for an optimal vector add loop
- It achieves ~7.4 IPC, ~3.2 loads & stores per cycle

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)
  {
    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



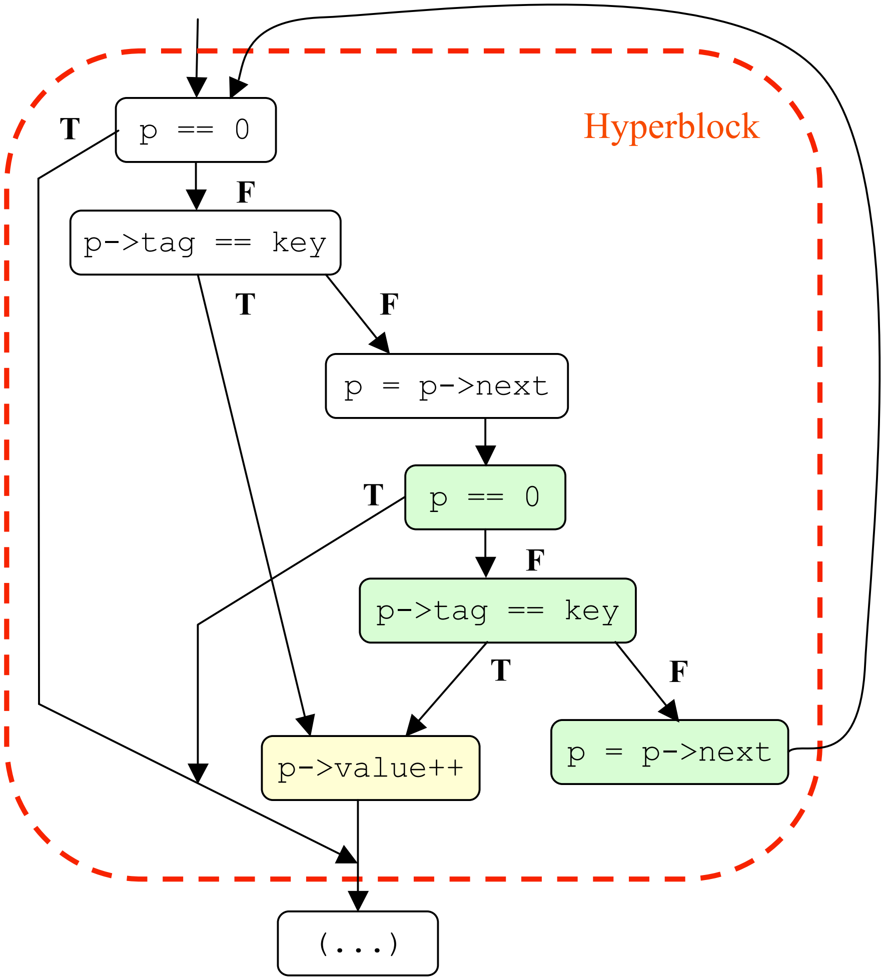
- The while loop's control flow graph is shown here
- It consists of four rather small basic blocks
- Its hard to get anywhere fast with all of those branches
- We'd like to use predication to form a larger program block (or *hyperblock*)

Linked List – Hyperblock #1

```
.bbegin block1
  read $t0, $g70              ; key
  read $t1, $g71              ; p
  ; BB0
  teqi $p0, $t1, 0
  ; BB1
  ld $t3, ($t1)               ; p->tag
  teq_f <$p0> $p1, $t3, $t0
  ld $t4, 16($t1)            ; p->next
  ; BB2
  ld $t10, 8($t1)             ; p->data
  null_t <$p0> $t11
  addi_t <$p1> $t11, $t10, 1
  null_f <$p1> $t11
  sd 8($t1), $t11             ; p->data
  ; BB3
  null_t <$p0,$p1> $t99
  mov_f <$p1> $t99, $t4
  write $g71, $t99            ; p
  ; branches
  bro_t <$p0,$p1> block2
  bro_f <$p1> block1
.bend
```

- 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
- 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

```
.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
```

- 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

p0	p1	p2	p3
T	-	-	-
F	T	-	-
F	F	T	-
F	F	F	T
F	F	F	F

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

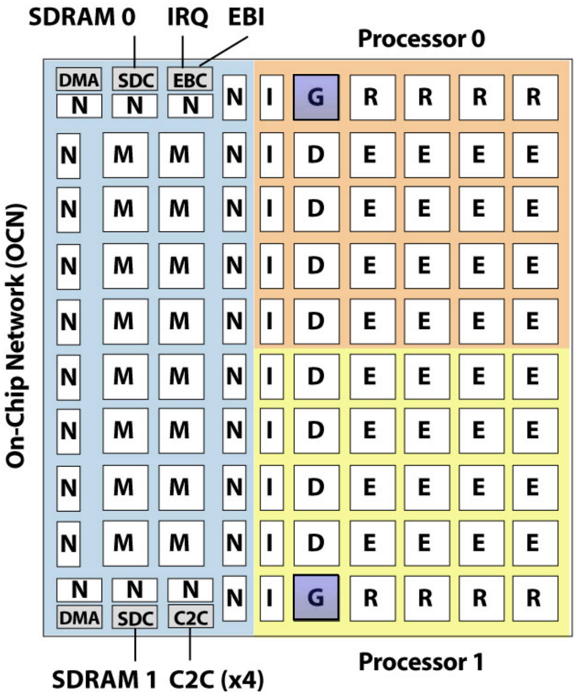
Global Control

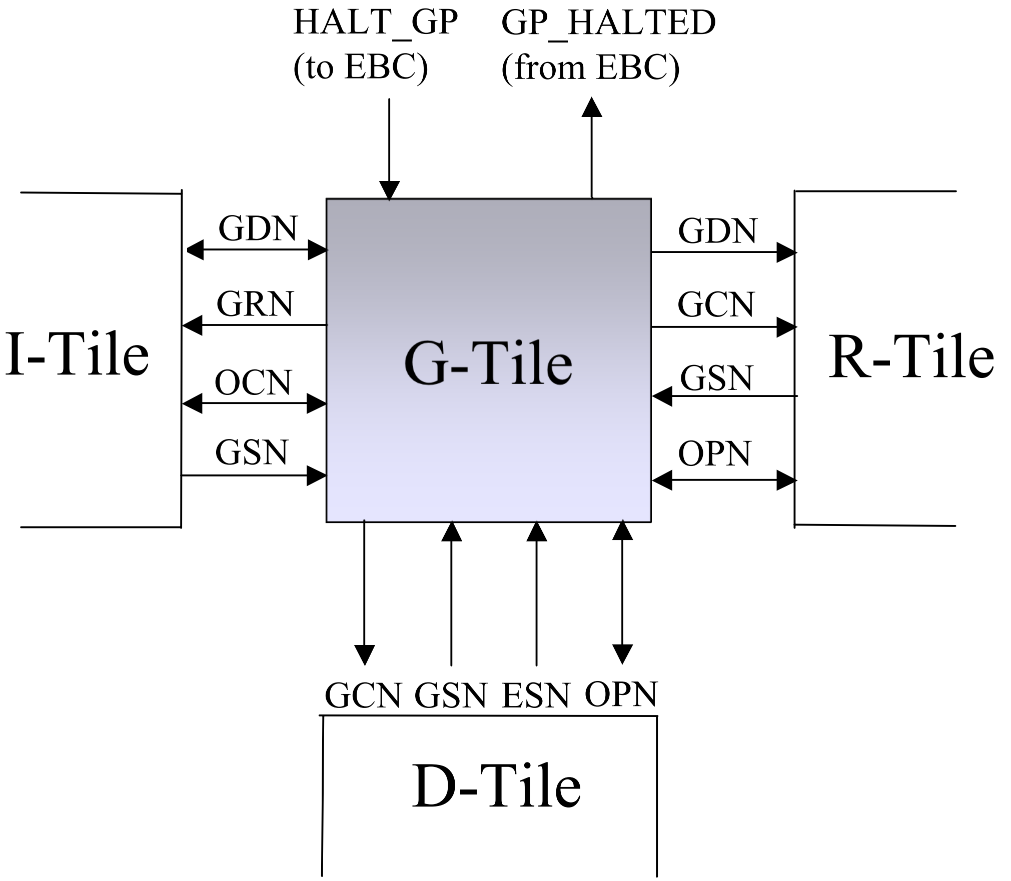
ISCA-32 Tutorial

Ramadass Nagarajan

Department of Computer Sciences
The University of Texas at Austin

G-Tile Location and Connections





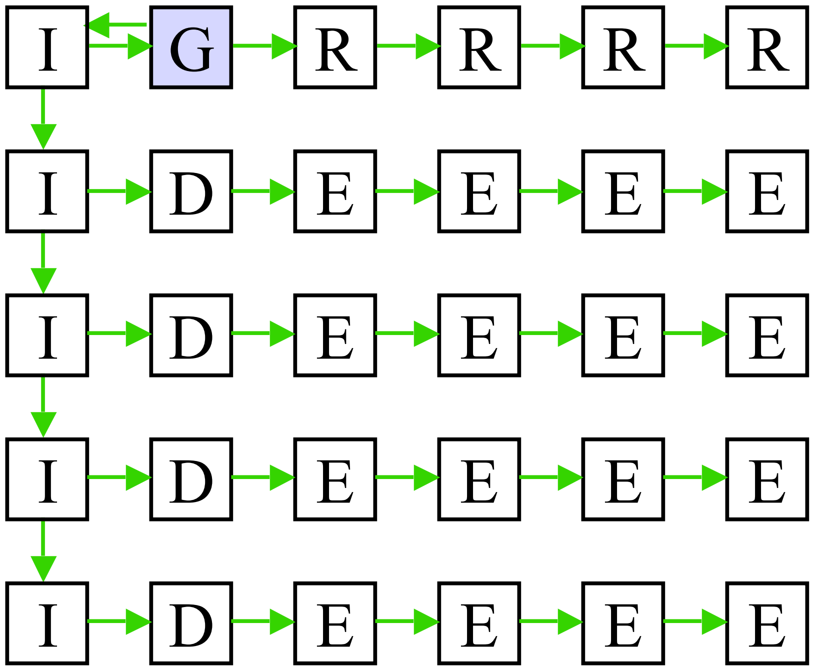
Major G-Tile Responsibilities

- 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)

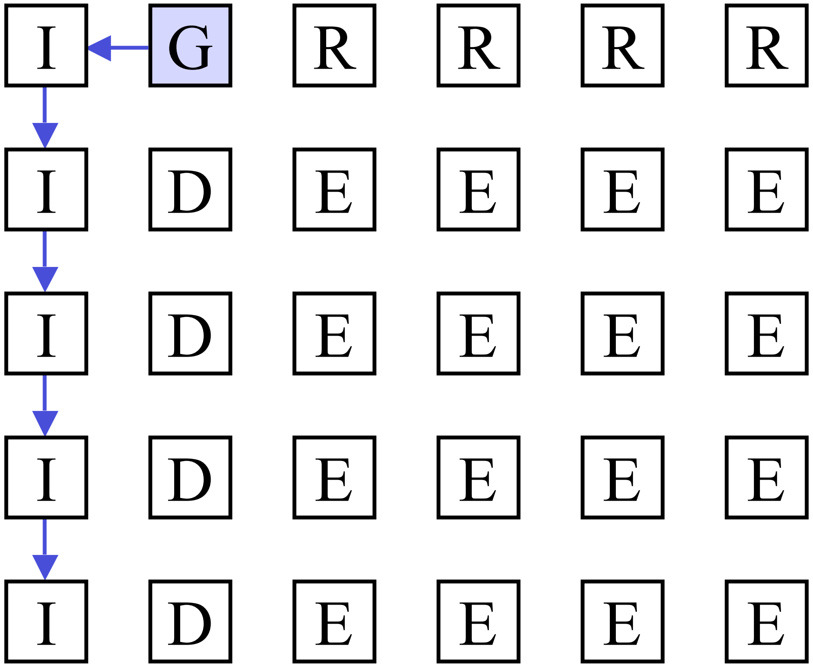


Interface Ports

CMD:	None/Fetch/Dispatch
ADDR:	Address of block
SMASK:	Store mask in block
FLAGS:	Block execution flags
ID:	Frame identifier
INST:	Instruction bits

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

Global Refill Network (GRN)



I G R R R R

I D E E E E

I D E E E E

I D E E E E

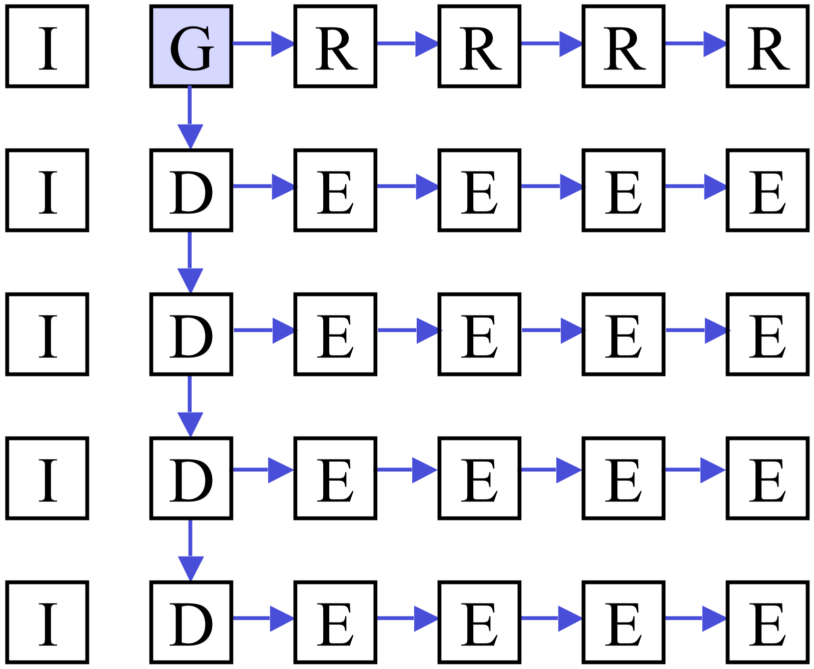
I D E E E E

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)



I G R R R R

I D E E E E

I D E E E E

I D E E E E

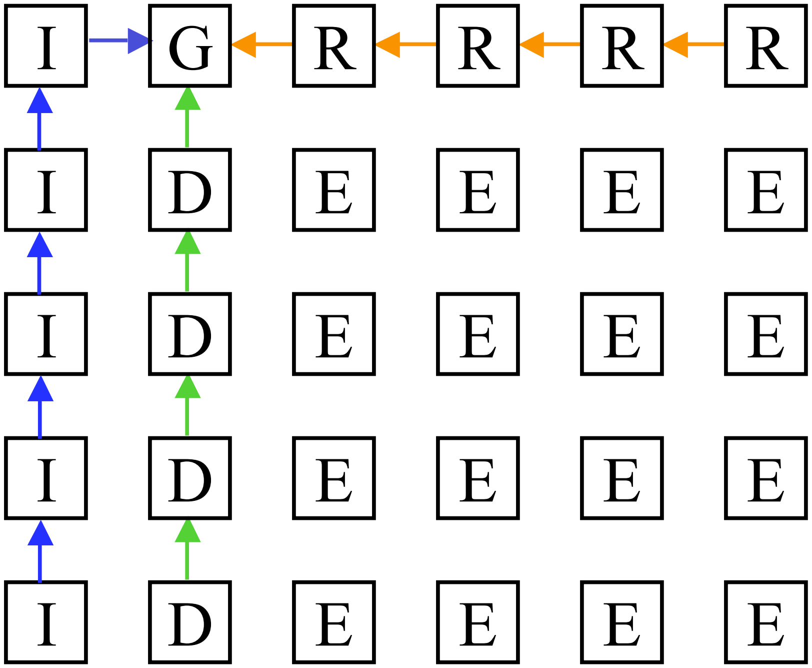
I D E E E E

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

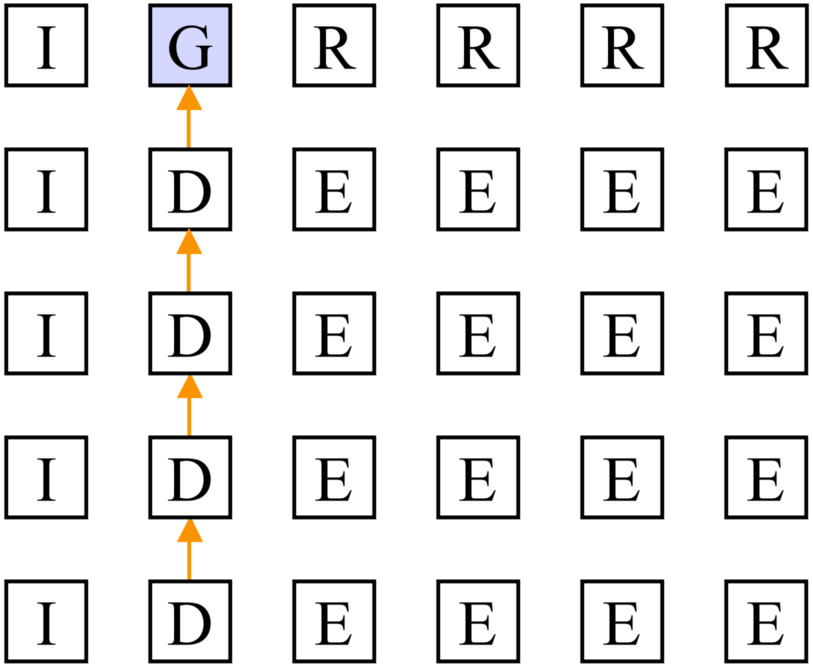
CMD:	None/Completed/Committed
ID:	Frame/Refill buffer identifier
STATUS:	Exception Status

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)



I G R R R R

I D E E E E

I D E E E E

I D E E E E

I D E E E E

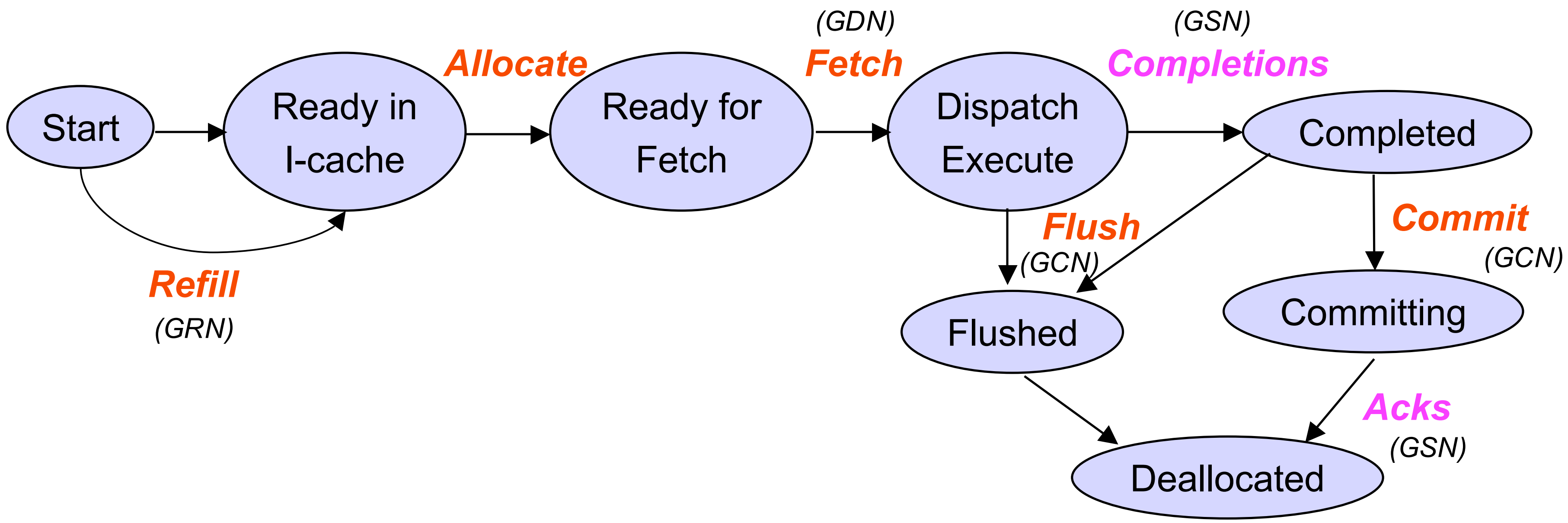
Interface Ports

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

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

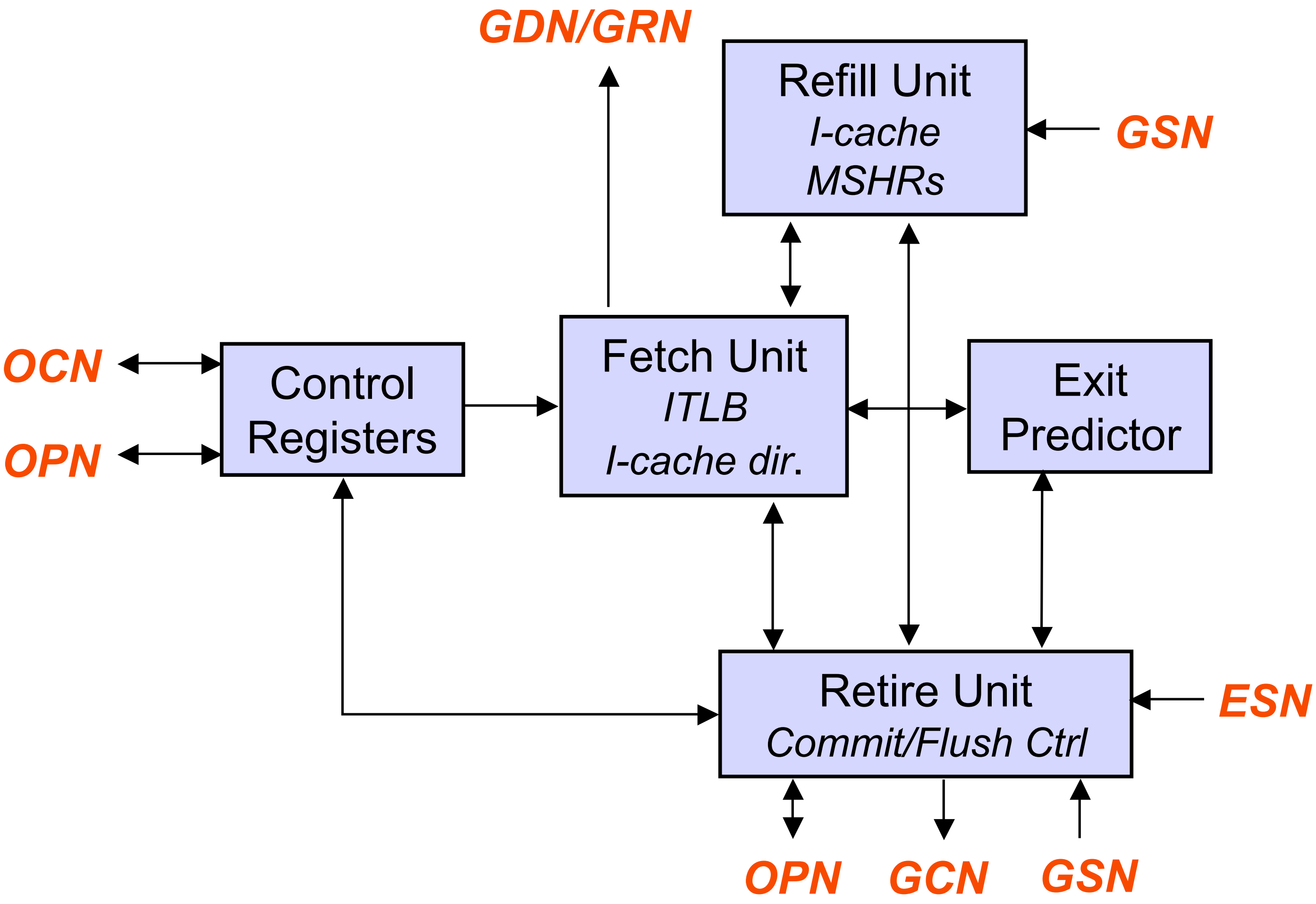
Delivers external store pending and error status from D-tiles

Overview of Block Execution



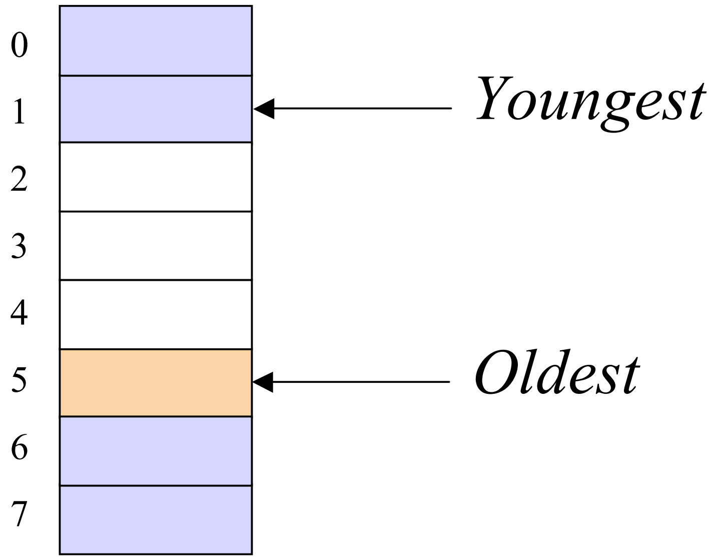
Different states in the execution of a block

Major G-Tile Structures



Frame Management

Frames allocated from a circular queue



Frames in use

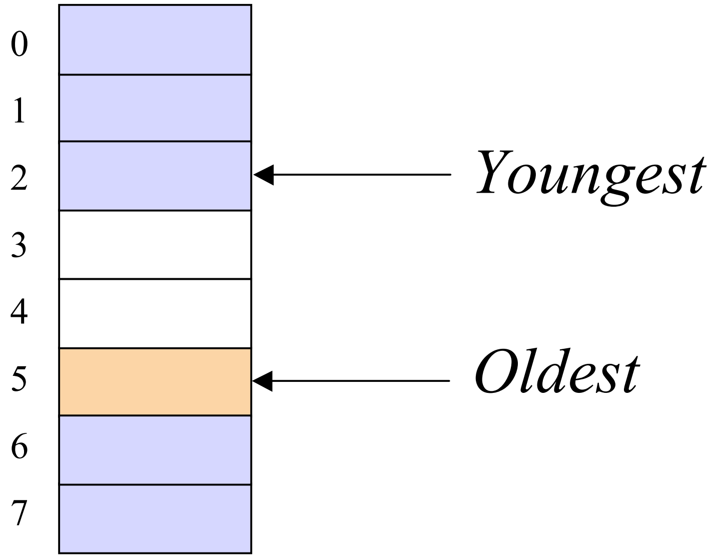
Control Speculation

🟧 Non-speculative block

🟦 Speculative blocks

Frame Management

Frames allocated from a circular queue



After a new fetch

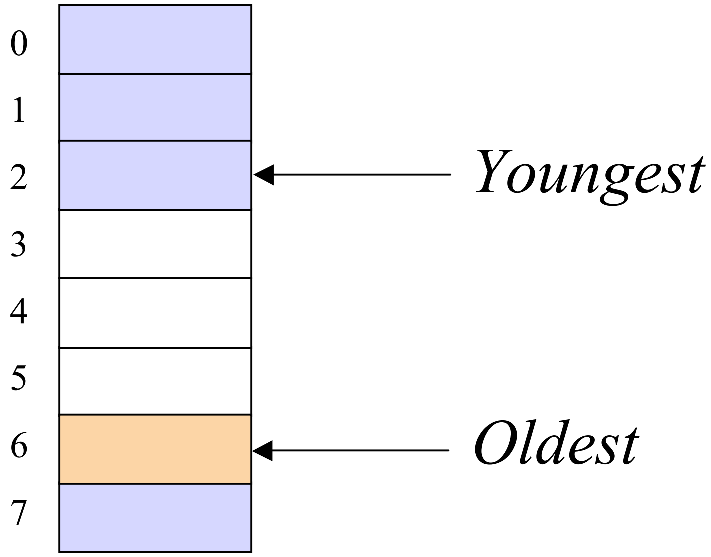
<u>Control Speculation</u>

🟧 Non-speculative block

🟪 Speculative blocks

Frame Management

Frames allocated from a circular queue



After a commit

<u>Control Speculation</u>

🟧 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

	Cycle 0	*Cycle 1*	*Cycle 2*	*Cycle 3*	*Cycle 4*	*Cycle 5*	*Cycle 6*	*Cycle 7*	*Cycle 8*
Block A	Predict (Stage 0)	Predict (Stage 1)	Predict (Stage 2) Address Select	TLB lookup I-cache lookup	Hit/Miss detection Frame allocation	Fetch SLOT 0	Fetch SLOT 1	Fetch SLOT 2	Fetch SLOT 3
Block B						Predict (Stage 0)	Predict (Stage 1)	Predict (Stage 2)	*Stall*

Cycle 9	*Cycle 10*	*Cycle 11*	*Cycle 12*	*Cycle 13*	*Cycle 14*	*Cycle 15*	*Cycle 16*	*Cycle 17*
Fetch SLOT 4	Fetch SLOT 5	Fetch SLOT 6	Fetch SLOT 7					
Stall	Address Select	TLB lookup I-cache lookup	Hit/Miss detection Frame allocation	Fetch SLOT 0	Fetch SLOT 1	Fetch SLOT 2	Fetch SLOT 3	Fetch SLOT 4

🟧 Predict cycle 🟨 Control cycle 🟦 Fetch cycle

Refill Pipeline

	Cycle 0	*Cycle 1*	*Cycle 2*	*Cycle 3*	*Cycle 4*	*Cycle 5*		*Cycle X*	*Cycle X+1*
Block A	Predict (Stage 0)	Predict (Stage 1)	Predict (Stage 2) Address Select	TLB lookup I-cache lookup	Hit/Miss detection **CACHE MISS**	Refill	……. ***Stall***	Refill Complete Frame allocation	Fetch SLOT 0

🟧 Predict cycle 🟨 Control cycle 🟦 Fetch cycle

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

Per-Frame State

Frame status

V	Valid
O	Oldest
Y	Youngest

Block execution status

RC	Registers completed
SC	Stores completed
BC	Branch completed
E	Exception reported
L	Load violation reported

Block commit status

CS	Commit sent
RCT	Registers committed
SCT	Stores committed

Branch prediction status

BADDR	Block address
NADDR	Next block address
NSTATE	Next block address state (predicted or resolved)
M	Branch misprediction detected
PC	Prediction completed
RC	Repair completed
UC	Update completed

```
Commit  ← V & O & RC & SC & BC
          & ~E & ~L & ~CS

Dealloc ← V & O & CS
           & RCT & SCT & 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

Cycle 0	*Cycle 1*	*Cycle 2*		*Cycle X*	*Cycle X+1*
Register/Branch/ Store Completion received (GSN/OPN)	Commit detect	Commit send (GCN) Predictor Commit	……	Commit acks received (GSN)	Frame deallocation

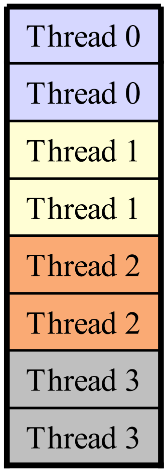
- Block ready for commit if:
  - Oldest in the thread
  - Completed with no exceptions
  - No load/store dependence violations detected

- 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

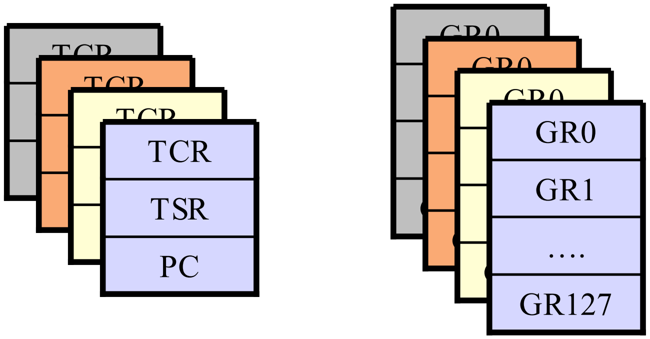
- Up to four threads supported in T-morph mode
  - Threads mapped to fixed set of frames
- Per-thread architected registers in the G-tile
  - Thread Control and Status Registers
  - Program Counter (PC)
- Other T-morph support state/logic
  - Per-thread frame management
  - Round-robin thread prioritization for block fetch/commit/flush

Frame Allocation



Thread 0
Thread 0
Thread 1
Thread 1
Thread 2
Thread 2
Thread 3
Thread 3

Per-thread Registers



TCR
TSR
PC

GR0
GR1
….
GR127

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

Exceptions detected at G-tile

- Breakpoint
- Timeout
- System Call
- Reset
- Fetch
- Interrupt

Exceptions detected at other tiles

- Execute
- Store
- Fetch

Control Flow Prediction

ISCA-32 Tutorial

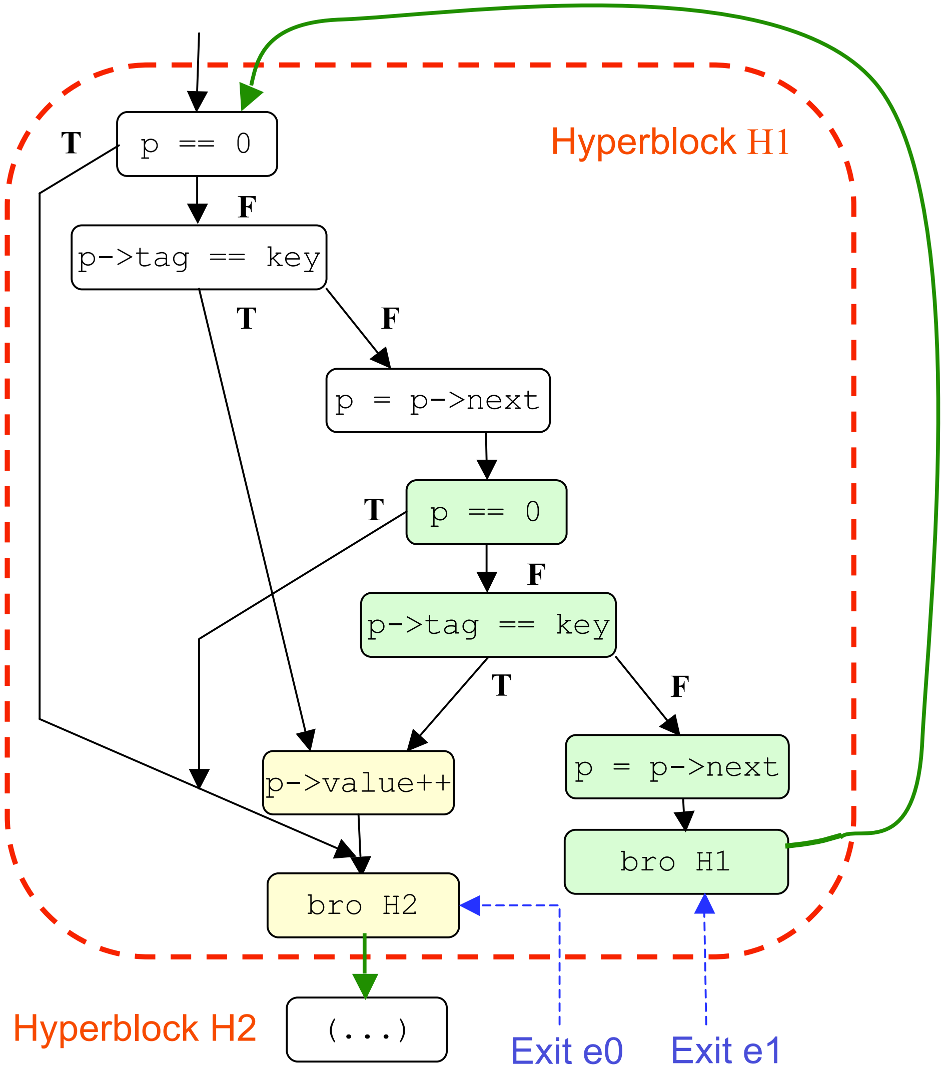
Nitya Ranganathan

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

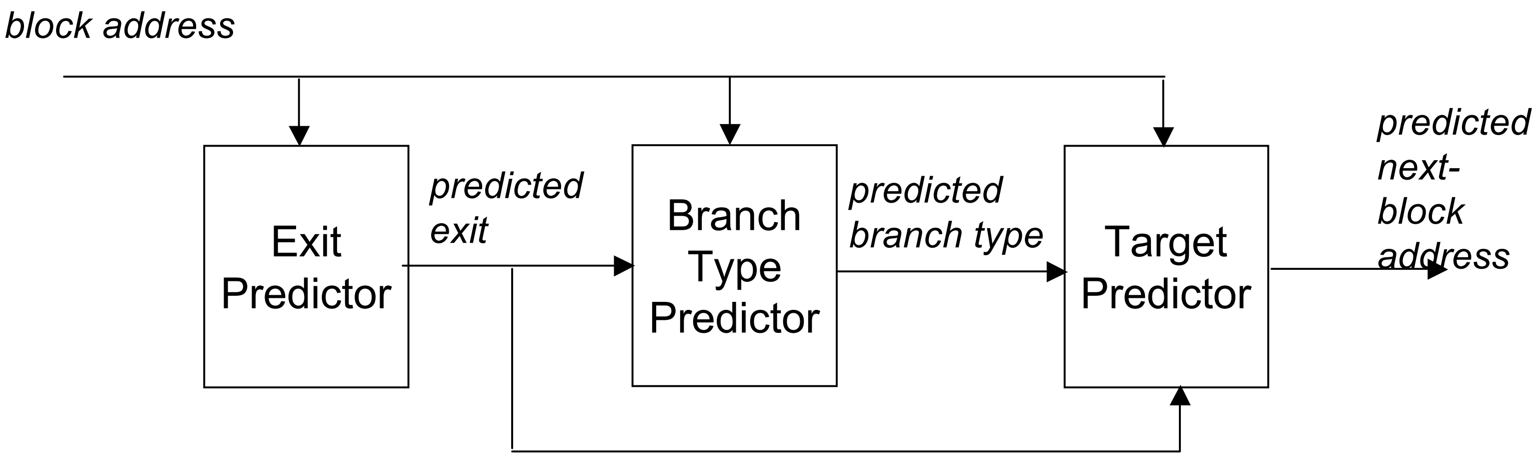
Example Hyperblock from Linked List Code



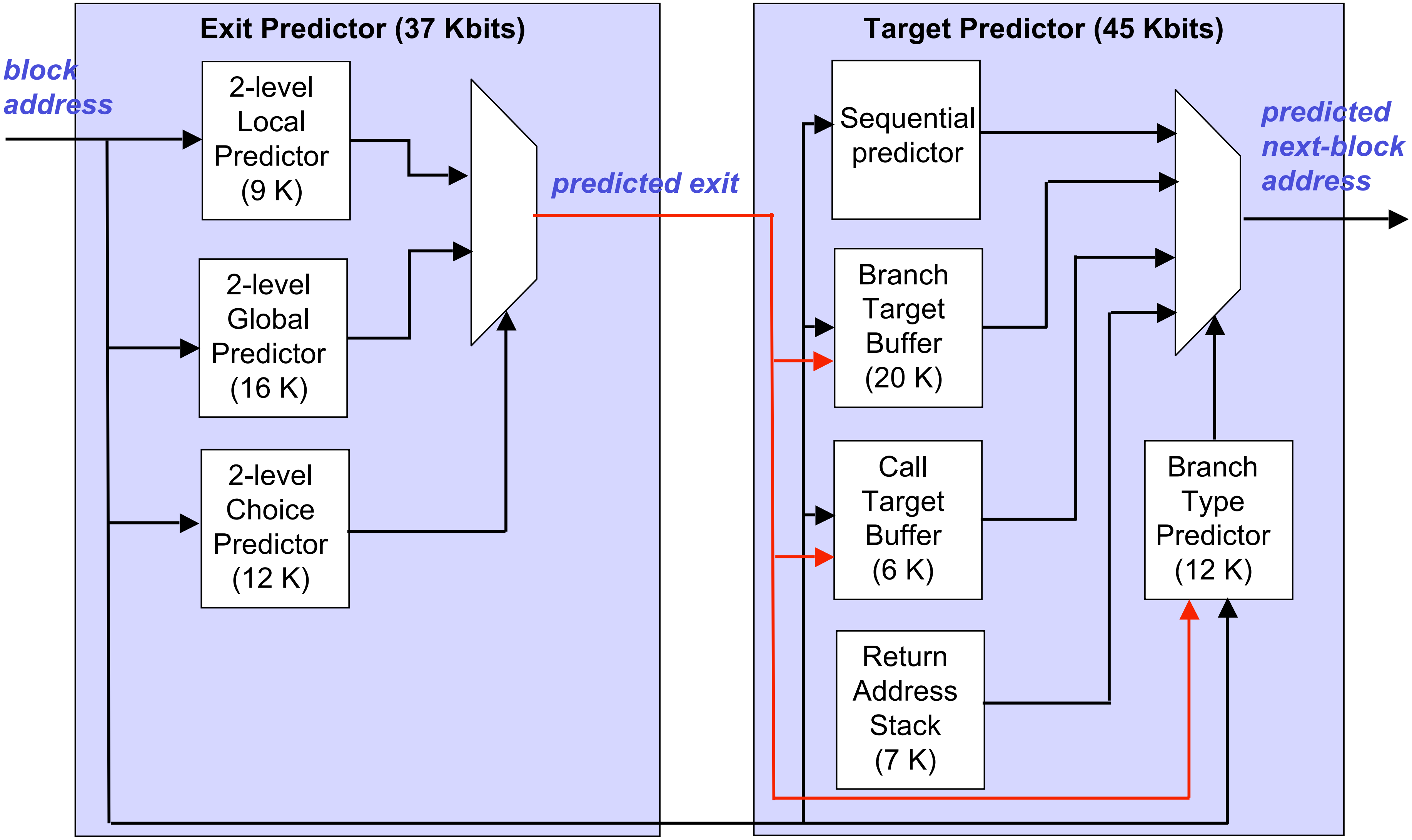
- Each hyperblock may have several predicated exit branches
  - Exit e0 ➔ block H2
  - Exit e1 ➔ block H1
- Only one branch will fire
- Predictor uses exit IDs to make prediction
  - 8 exit IDs supported
  - e0 ➔ “000”
  - e1 ➔ “001”
- Predictor predicts the ID of the exit that will fire
  - implicitly 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

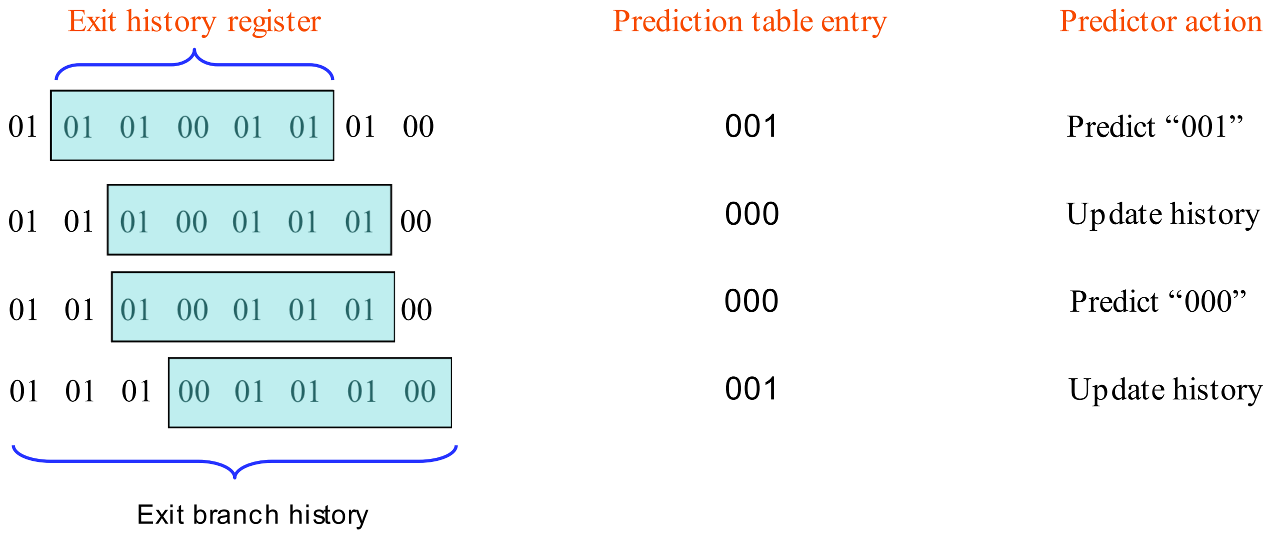


Predictor Components


Exit Predictor (37 Kbits)
block address
2-level Local Predictor (9 K)
2-level Global Predictor (16 K)
2-level Choice Predictor (12 K)
predicted exit
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)
predicted next-block address

Prediction using Exit History (linked list example)

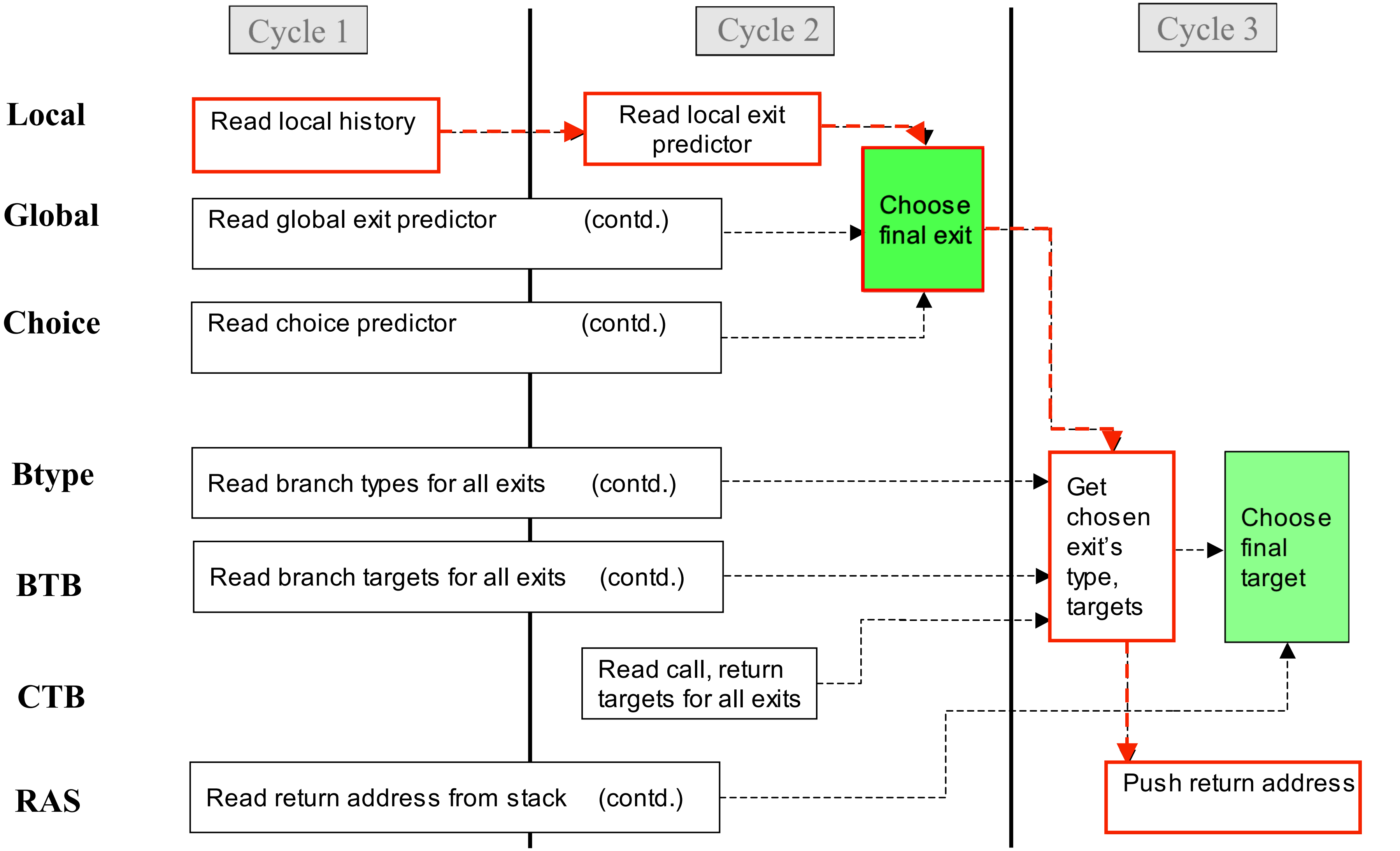
- Dynamic block sequence H1 H1 H1 H2 H1 H1 H1 H2
- Local exit history for block H1 e1 e1 e1 e0 e1 e1 e1 e0
- Local exit history encoding (2 bits) 01 01 01 00 01 01 01 00
  - Use only 2 bits from each 3-bit exit to permit longer histories
- History table entry holds 5 exit IDs
- Prediction table contains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



Exit history register	Prediction table entry	Predictor action
01 [01 01 00 01 01] 01 00	001	Predict “001”
01 01 [01 00 01 01 01] 00	000	Update history
01 01 [01 00 01 01 01] 00	000	Predict “000”
01 01 01 [00 01 01 01 00]	001	Update history

Exit branch history

Prediction Timing

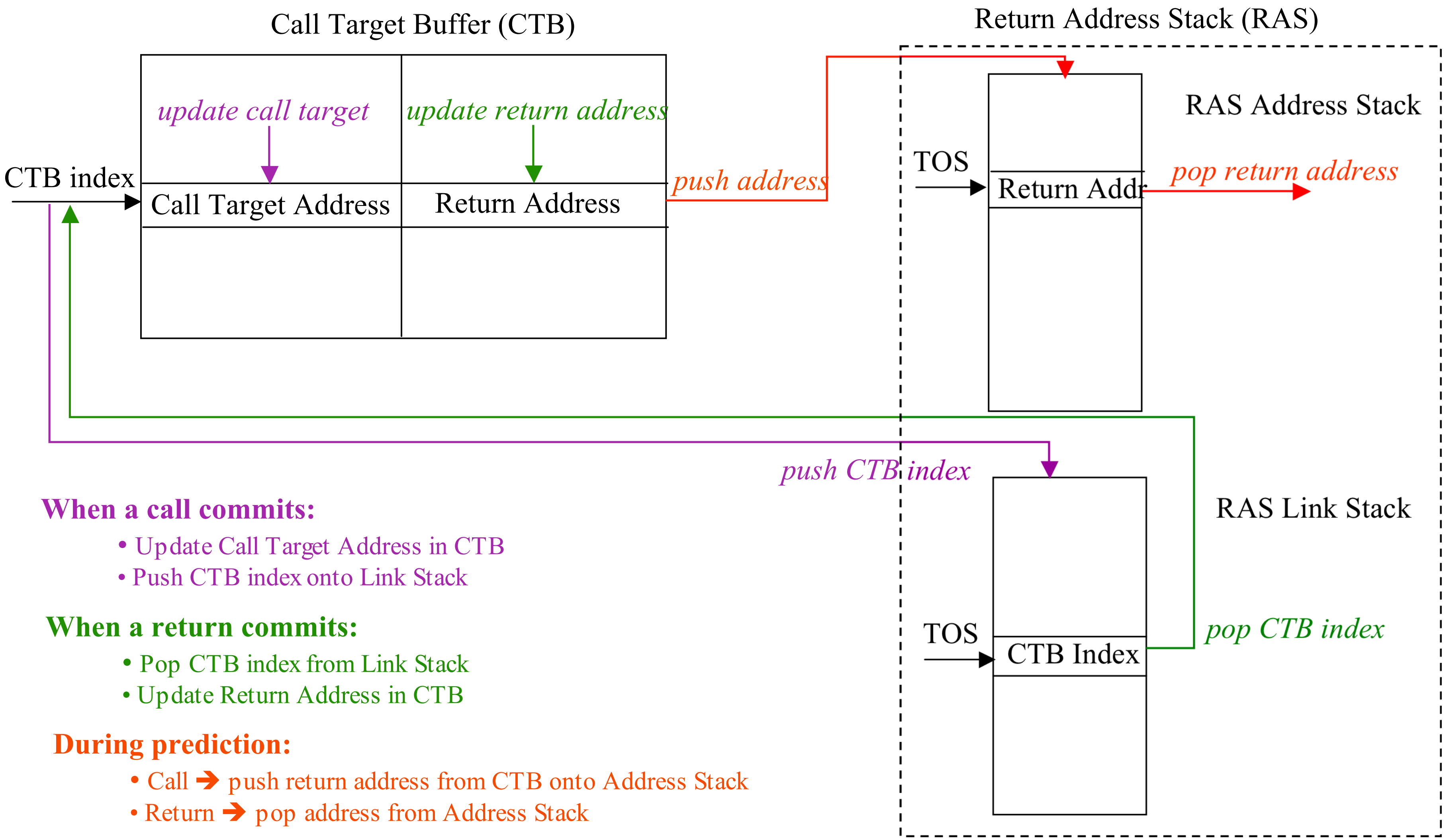


Cycle 1
Cycle 2
Cycle 3
Local
Read local history
Read local exit predictor
Global
Read global exit predictor (contd.)
Choose final exit
Choice
Read choice predictor (contd.)
Btype
Read branch types for all exits (contd.)
Get chosen exit's type, targets
Choose final target
BTB
Read branch targets for all exits (contd.)
CTB
Read call, return targets for all exits
RAS
Read return address from stack (contd.)
Push return address

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

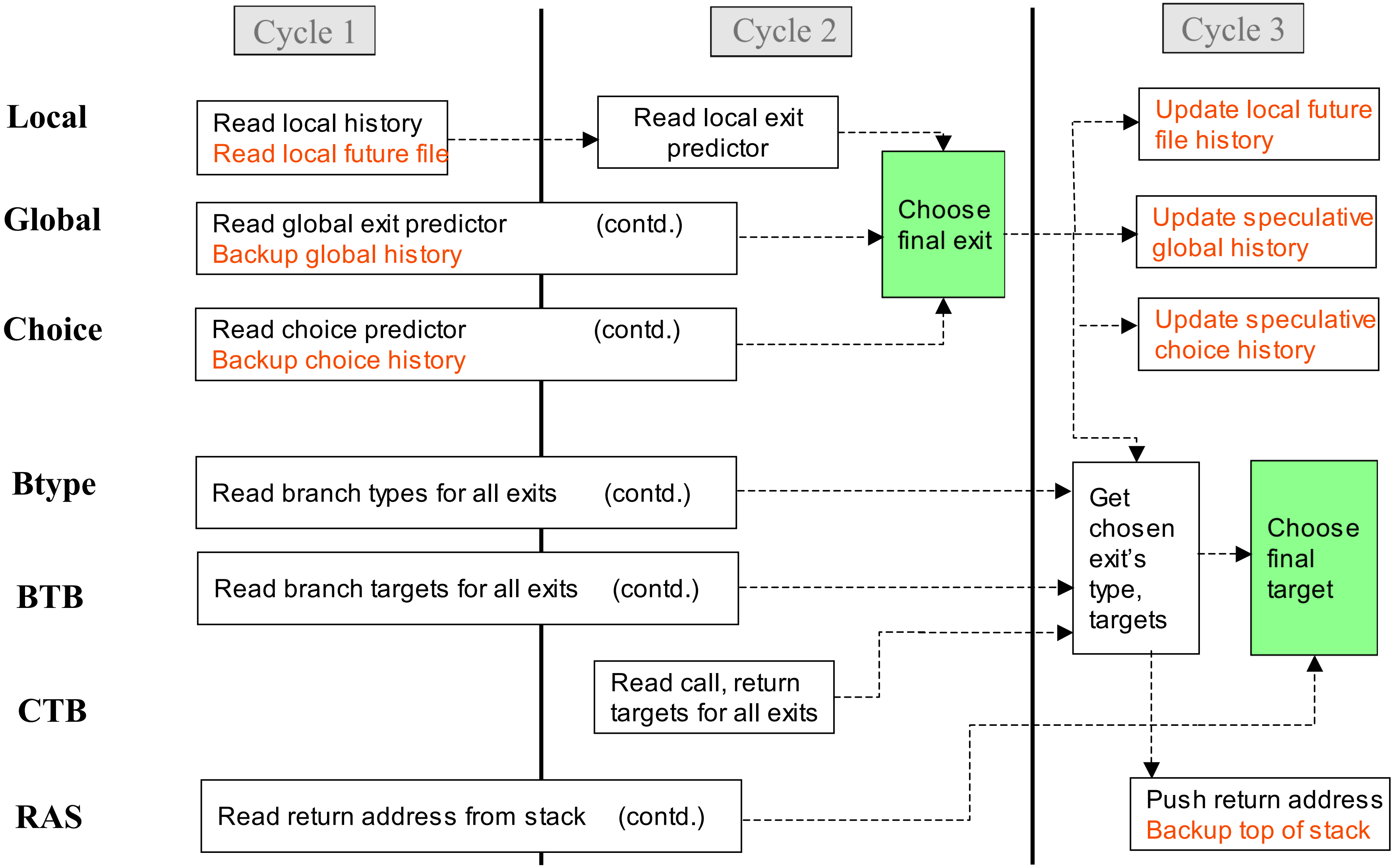
When a return commits:

- 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

Cycle 2

Cycle 3

Local

Read local history
Read local future file

Read local exit predictor

Choose final exit

Update local future file history

Global

Read global exit predictor (contd.)
Backup global history

Update speculative global history

Choice

Read choice predictor (contd.)
Backup choice history

Update speculative choice history

Btype

Read branch types for all exits (contd.)

Get chosen exit's type, targets

Choose final target

BTB

Read branch targets for all exits (contd.)

CTB

Read call, return targets for all exits

RAS

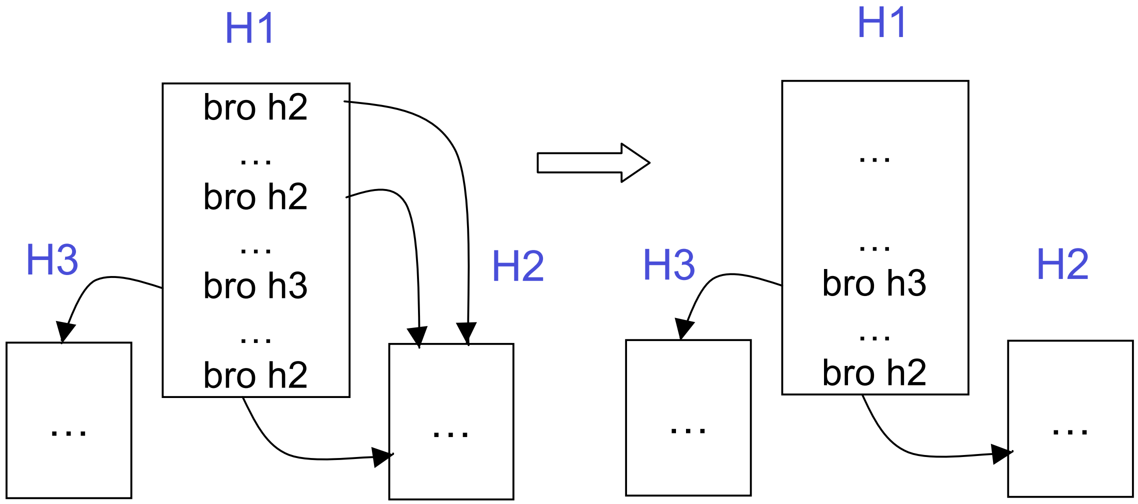
Read return address from stack (contd.)

Push return address
Backup top of stack

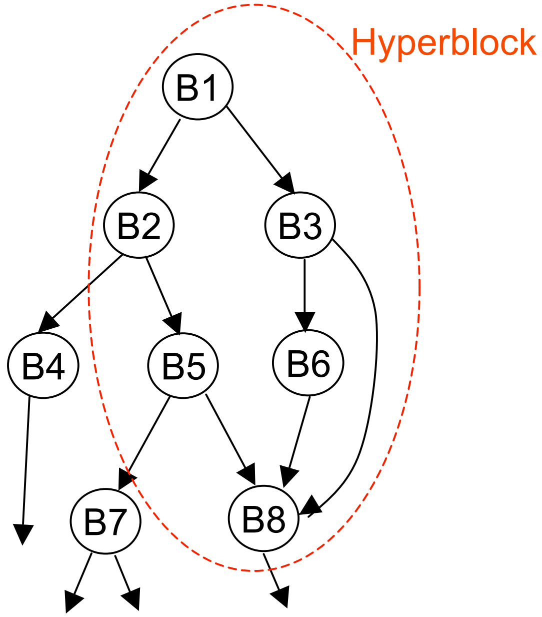
Optimizations for Better Prediction

- 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)



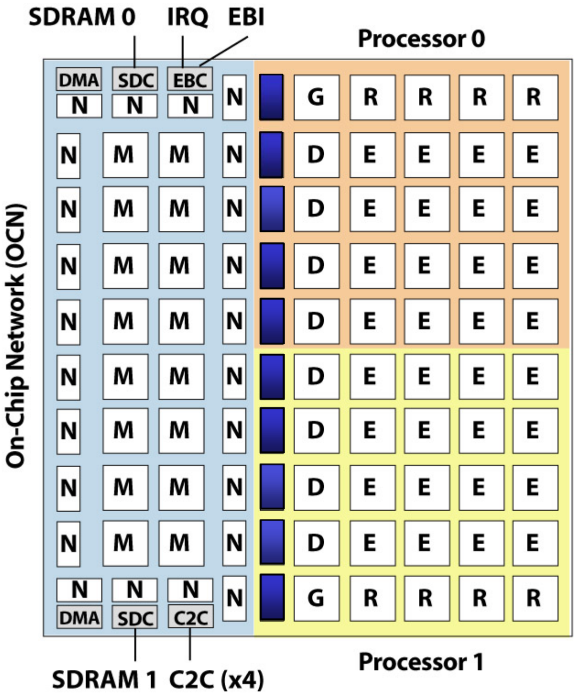
Instruction Fetch

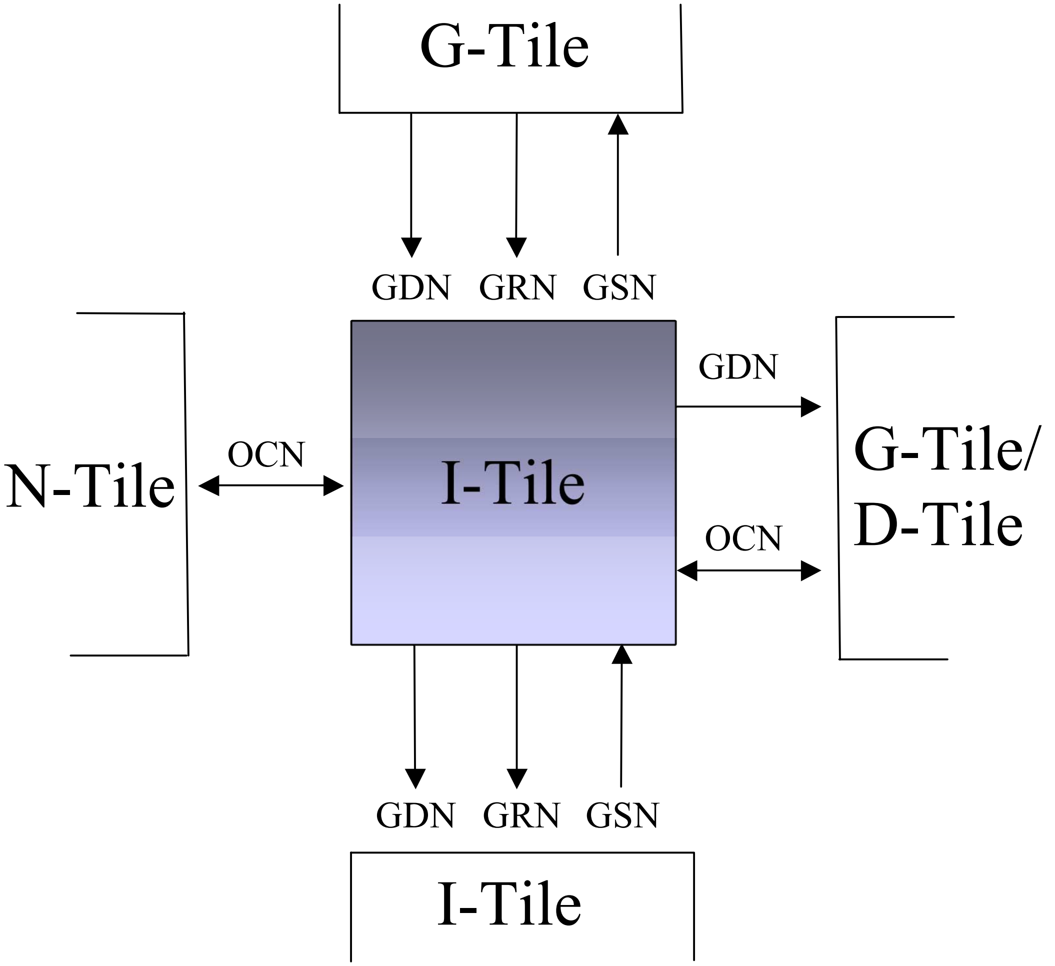
ISCA-32 Tutorial

Haiming Liu

Department of Computer Sciences
The University of Texas at Austin

I-Tile Location and Connections

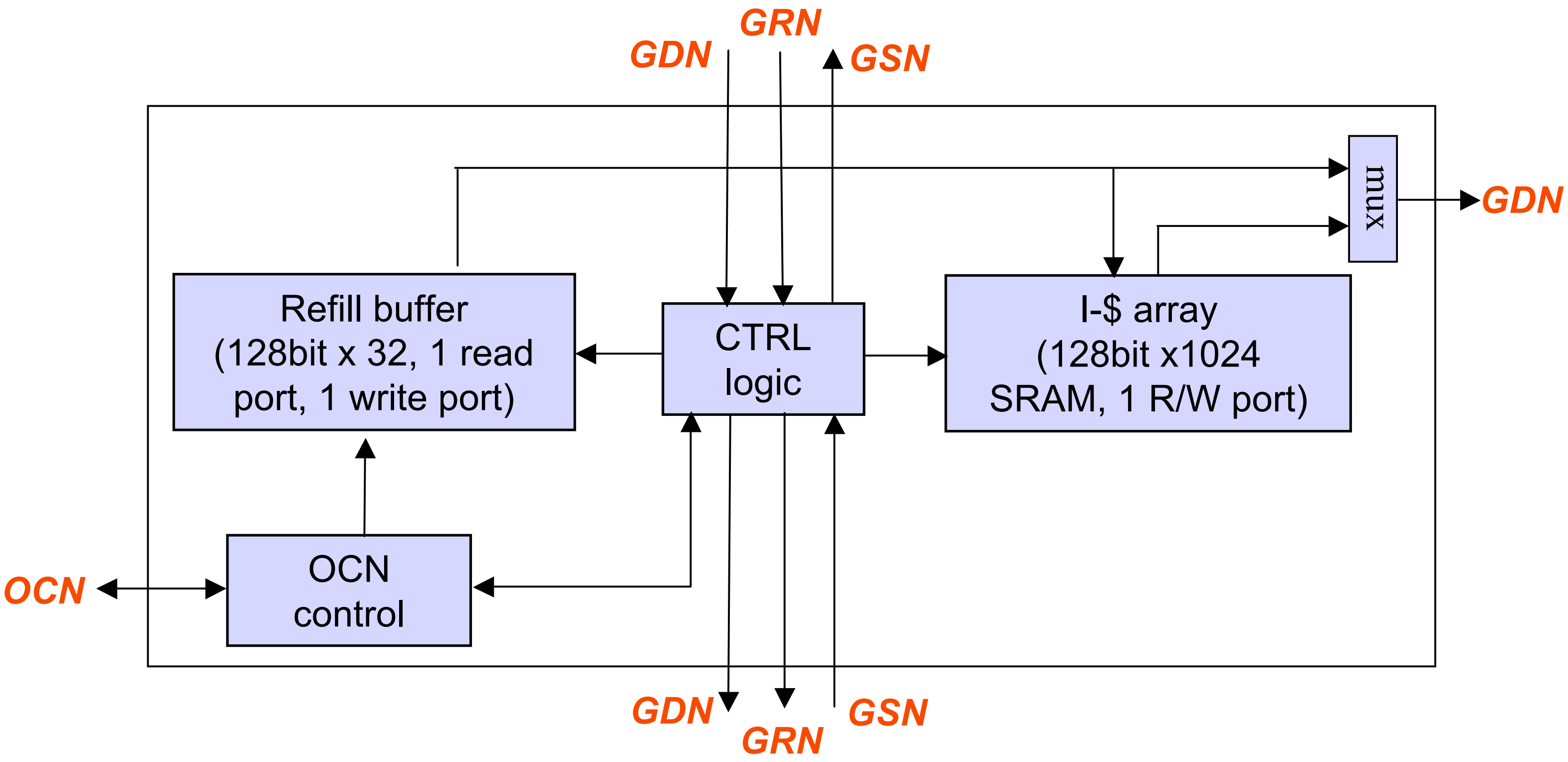




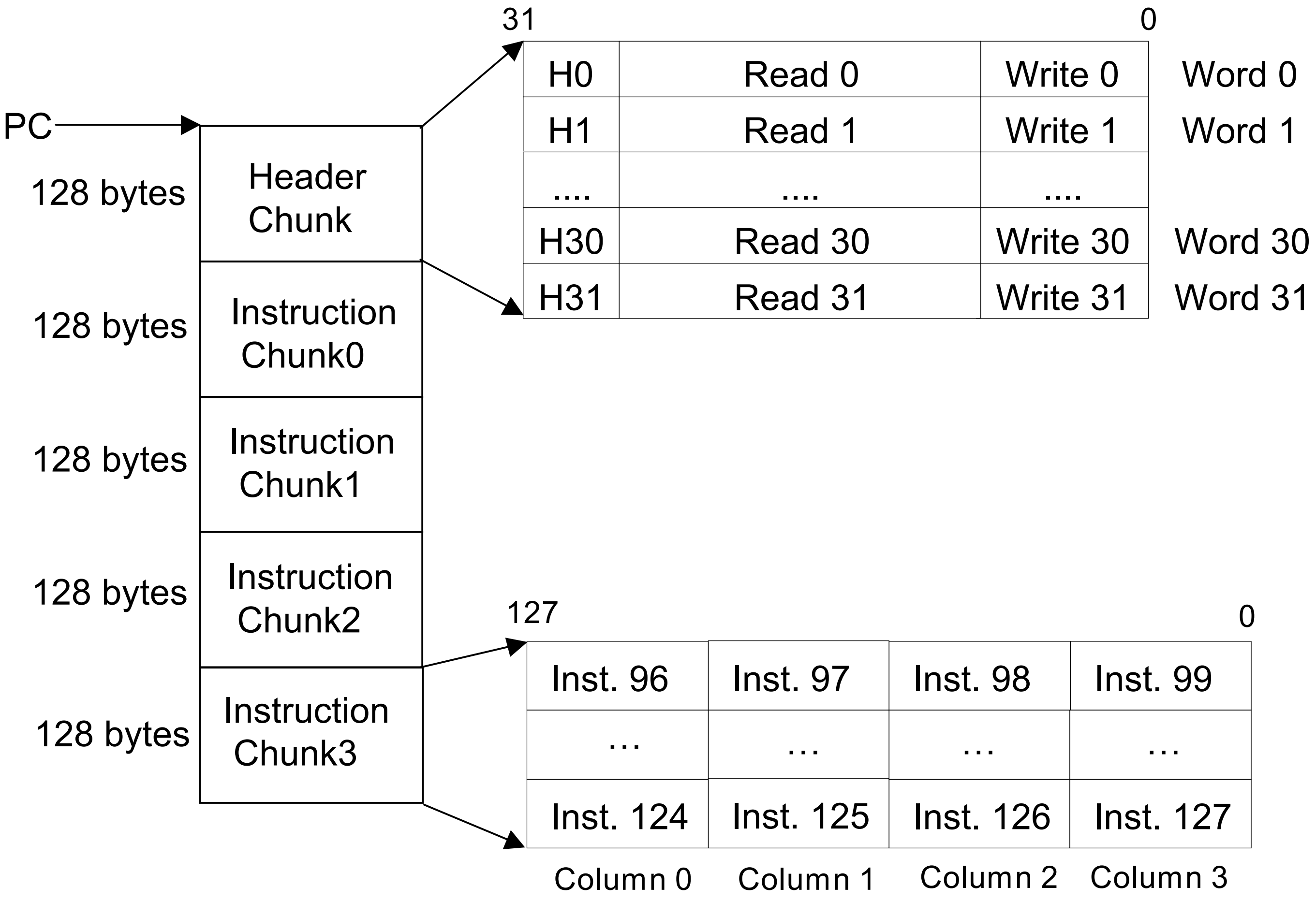
I-Tile Major Responsibilities

- 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

Major I-Tile Structures


GDN
GRN
GSN
mux
GDN
Refill buffer (128bit x 32, 1 read port, 1 write port)
CTRL logic
I-$ array (128bit x1024 SRAM, 1 R/W port)
OCN
OCN control
GDN
GRN
GSN

Instruction Layout in a Maximal Block



PC

128 bytes	Header Chunk
128 bytes	Instruction Chunk0
128 bytes	Instruction Chunk1
128 bytes	Instruction Chunk2
128 bytes	Instruction Chunk3

31 0

H0	Read 0	Write 0	Word 0
H1	Read 1	Write 1	Word 1
....	....	....	
H30	Read 30	Write 30	Word 30
H31	Read 31	Write 31	Word 31

127 0

Inst. 96	Inst. 97	Inst. 98	Inst. 99
…	…	…	…
Inst. 124	Inst. 125	Inst. 126	Inst. 127
Column 0	Column 1	Column 2	Column 3

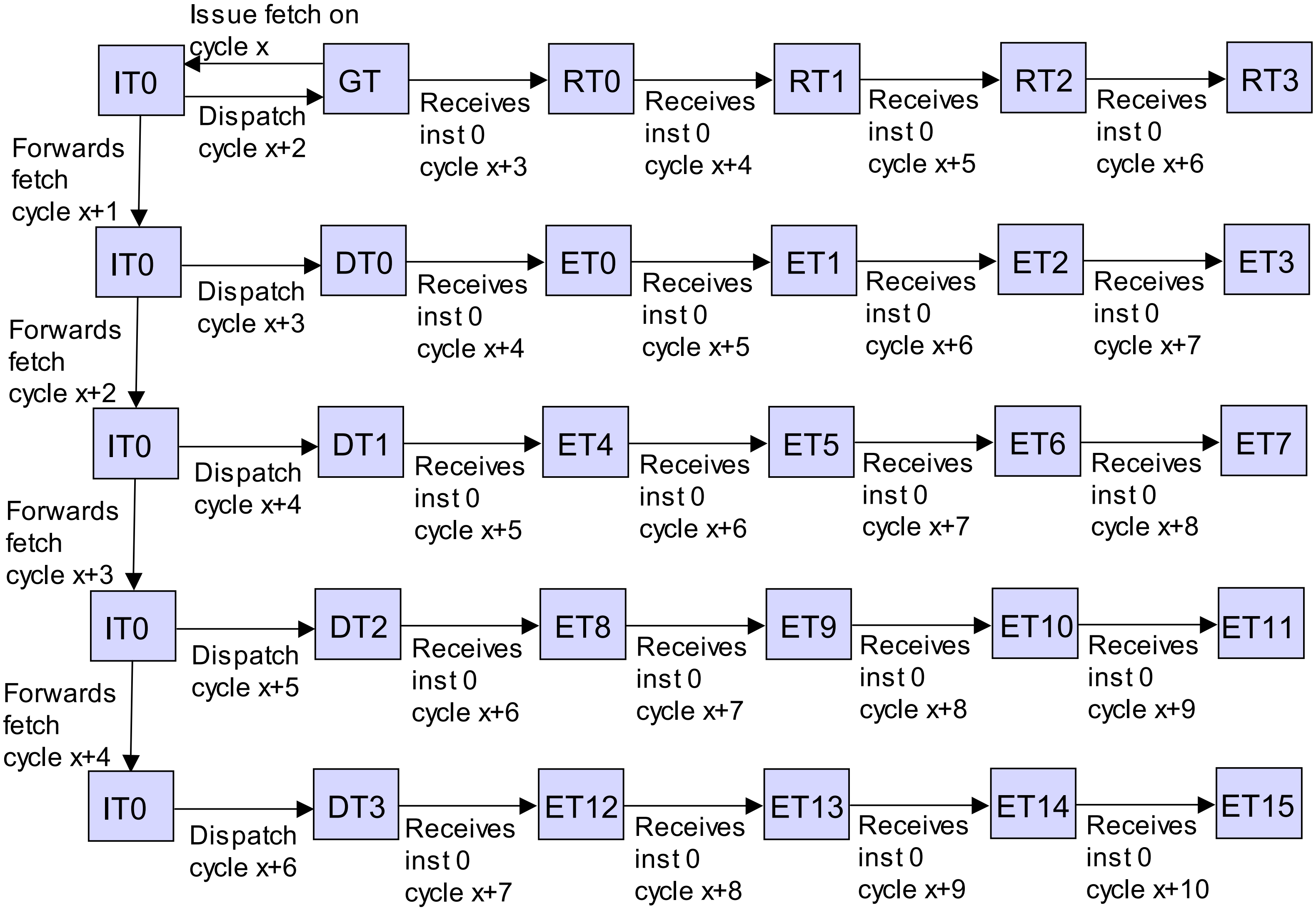
Instruction Fetch Pipeline

	Cycle 0	*Cycle 1*	*Cycle 2*	*Cycle 3*	*Cycle 4*		*Cycle 9*
GT	ITLB lookup I-$ Tag lookup	Hit/Miss detection Frame allocation	Fetch slot 0	Fetch slot 1	Fetch slot 2	......	Fetch slot 7

	Cycle 3	*Cycle 4*		*Cycle 9*	*Cycle 10*
IT0	R/W slot 0 Fetch slot 0	R/W slot 1 Fetch slot 1 Dispatch slot 0	....	R/W slot 6 Fetch slot 6 Dispatch slot 5	R/W slot 7 Fetch slot 7 Dispatch slot 6

	Cycle 4		*Cycle 9*	
IT1	R/W slot 0 Fetch slot 0	....	R/W slot 5 Fetch slot 5 Dispatch slot 4	....

Instruction Dispatch Timing Diagram



Issue fetch on cycle x

IT0 → GT: Dispatch cycle x+2

GT → RT0: Receives inst 0 cycle x+3

RT0 → RT1: Receives inst 0 cycle x+4

RT1 → RT2: Receives inst 0 cycle x+5

RT2 → RT3: Receives inst 0 cycle x+6

Forwards fetch cycle x+1

IT0 → DT0: Dispatch cycle x+3

DT0 → ET0: Receives inst 0 cycle x+4

ET0 → ET1: Receives inst 0 cycle x+5

ET1 → ET2: Receives inst 0 cycle x+6

ET2 → ET3: Receives inst 0 cycle x+7

Forwards fetch cycle x+2

IT0 → DT1: Dispatch cycle x+4

DT1 → ET4: Receives inst 0 cycle x+5

ET4 → ET5: Receives inst 0 cycle x+6

ET5 → ET6: Receives inst 0 cycle x+7

ET6 → ET7: Receives inst 0 cycle x+8

Forwards fetch cycle x+3

IT0 → DT2: Dispatch cycle x+5

DT2 → ET8: Receives inst 0 cycle x+6

ET8 → ET9: Receives inst 0 cycle x+7

ET9 → ET10: Receives inst 0 cycle x+8

ET10 → ET11: Receives inst 0 cycle x+9

Forwards fetch cycle x+4

IT0 → DT3: Dispatch cycle x+6

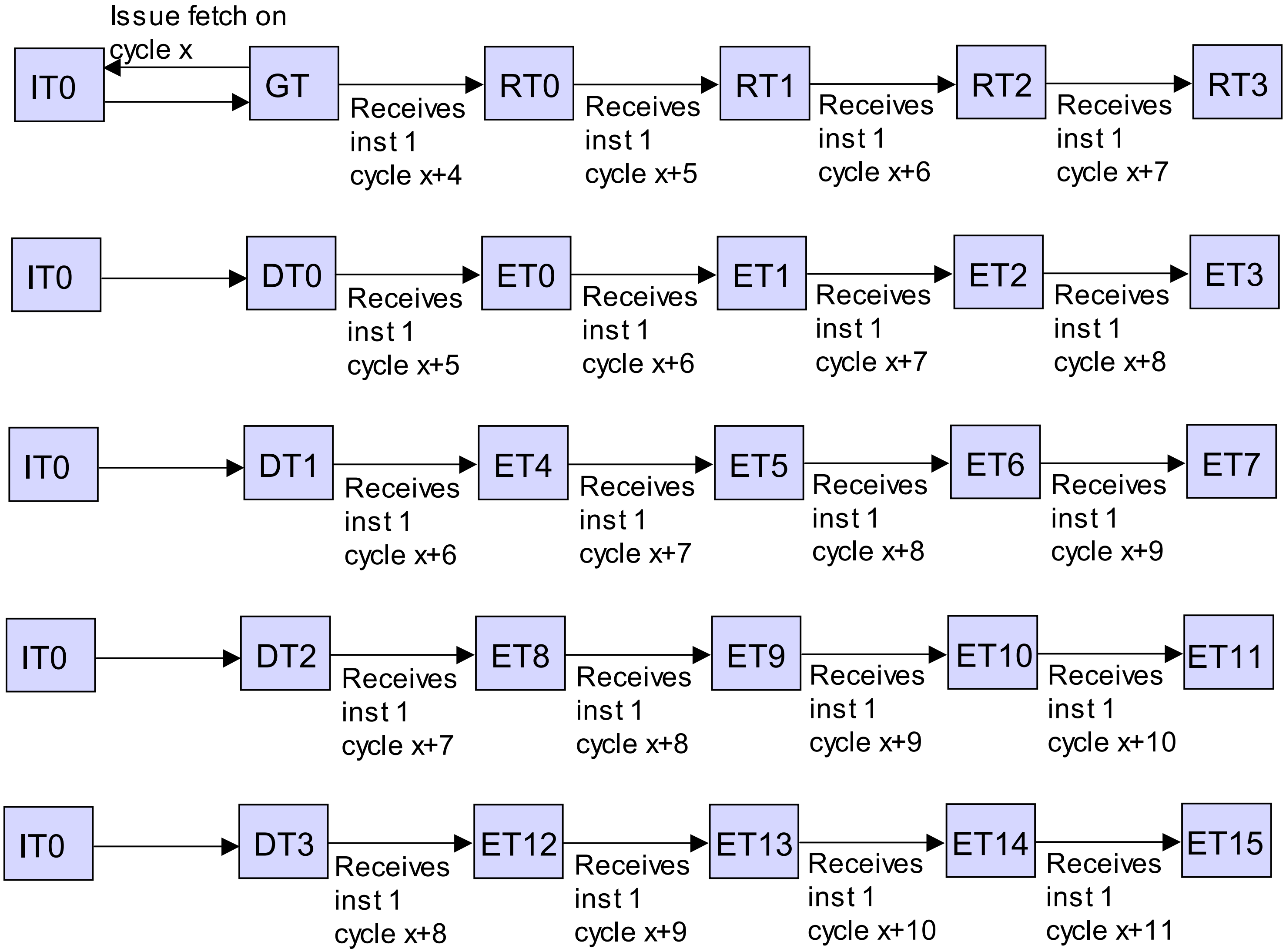
DT3 → ET12: Receives inst 0 cycle x+7

ET12 → ET13: Receives inst 0 cycle x+8

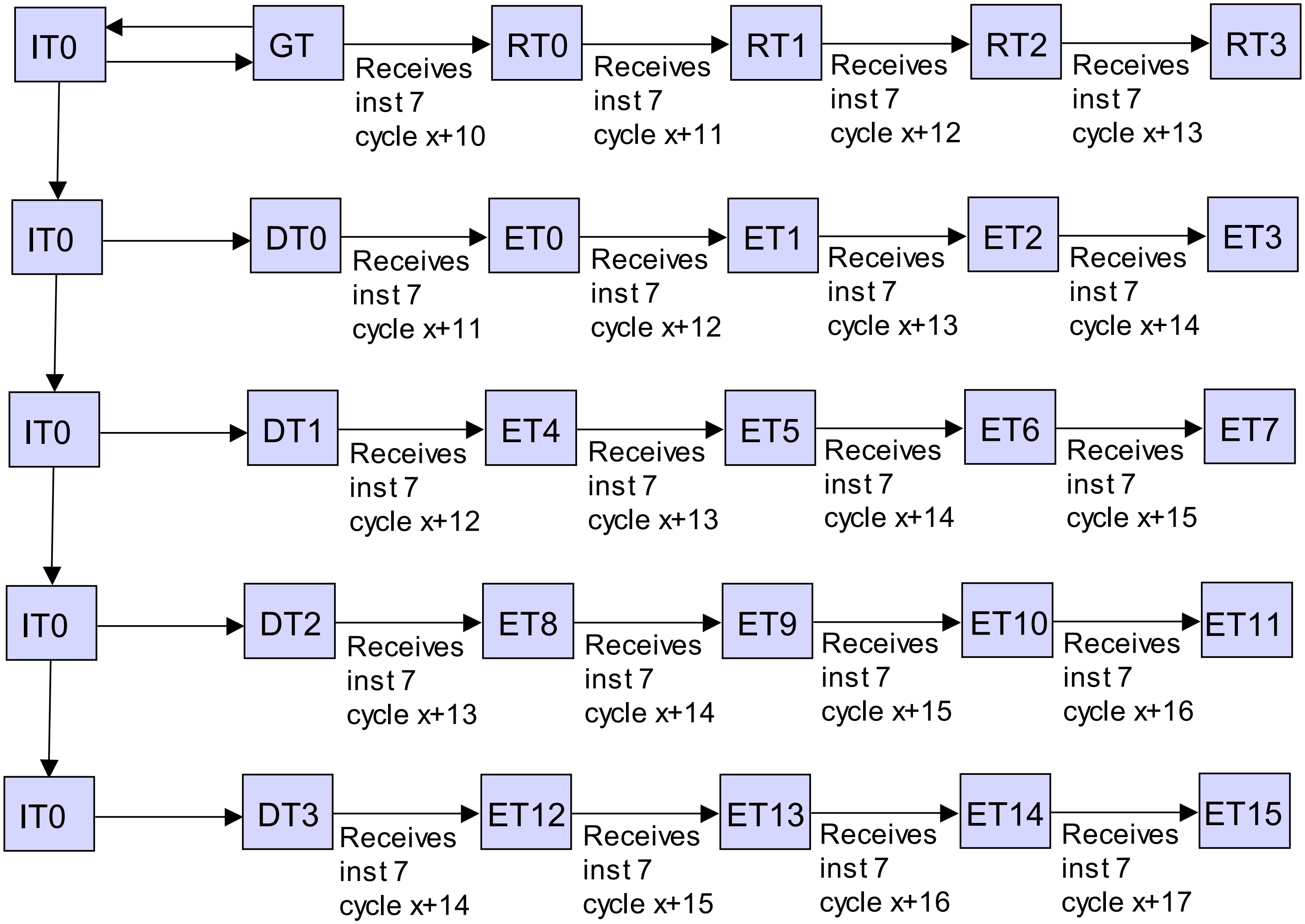
ET13 → ET14: Receives inst 0 cycle x+9

ET14 → ET15: Receives inst 0 cycle x+10

Instruction Dispatch Timing Diagram


Issue fetch on cycle x
IT0
GT
Receives inst 1 cycle x+4
RT0
Receives inst 1 cycle x+5
RT1
Receives inst 1 cycle x+6
RT2
Receives inst 1 cycle x+7
RT3
IT0
DT0
Receives inst 1 cycle x+5
ET0
Receives inst 1 cycle x+6
ET1
Receives inst 1 cycle x+7
ET2
Receives inst 1 cycle x+8
ET3
IT0
DT1
Receives inst 1 cycle x+6
ET4
Receives inst 1 cycle x+7
ET5
Receives inst 1 cycle x+8
ET6
Receives inst 1 cycle x+9
ET7
IT0
DT2
Receives inst 1 cycle x+7
ET8
Receives inst 1 cycle x+8
ET9
Receives inst 1 cycle x+9
ET10
Receives inst 1 cycle x+10
ET11
IT0
DT3
Receives inst 1 cycle x+8
ET12
Receives inst 1 cycle x+9
ET13
Receives inst 1 cycle x+10
ET14
Receives inst 1 cycle x+11
ET15

Instruction Dispatch Timing Diagram



IT0
GT
Receives inst 7 cycle x+10
RT0
Receives inst 7 cycle x+11
RT1
Receives inst 7 cycle x+12
RT2
Receives inst 7 cycle x+13
RT3

IT0
DT0
Receives inst 7 cycle x+11
ET0
Receives inst 7 cycle x+12
ET1
Receives inst 7 cycle x+13
ET2
Receives inst 7 cycle x+14
ET3

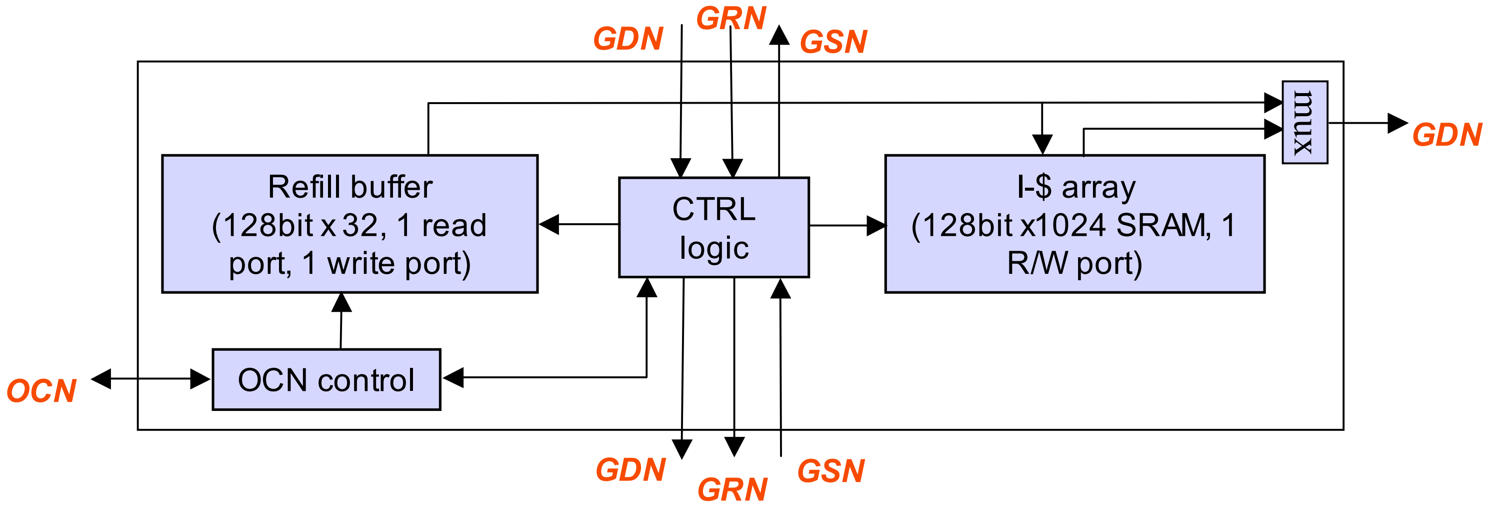
IT0
DT1
Receives inst 7 cycle x+12
ET4
Receives inst 7 cycle x+13
ET5
Receives inst 7 cycle x+14
ET6
Receives inst 7 cycle x+15
ET7

IT0
DT2
Receives inst 7 cycle x+13
ET8
Receives inst 7 cycle x+14
ET9
Receives inst 7 cycle x+15
ET10
Receives inst 7 cycle x+16
ET11

IT0
DT3
Receives inst 7 cycle x+14
ET12
Receives inst 7 cycle x+15
ET13
Receives inst 7 cycle x+16
ET14
Receives inst 7 cycle x+17
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


GDN
GRN
GSN
mux
GDN
Refill buffer (128bit x 32, 1 read port, 1 write port)
CTRL logic
I-$ array (128bit x1024 SRAM, 1 R/W port)
OCN
OCN control
GDN
GRN
GSN

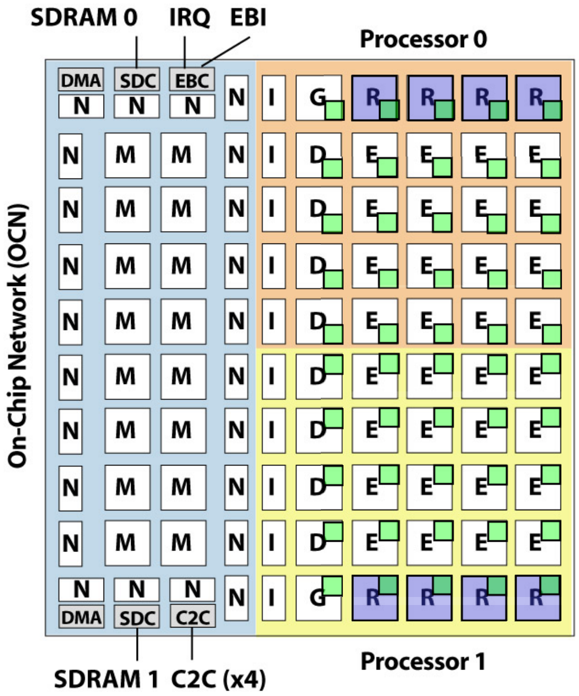
Register Accesses and Operand Routing

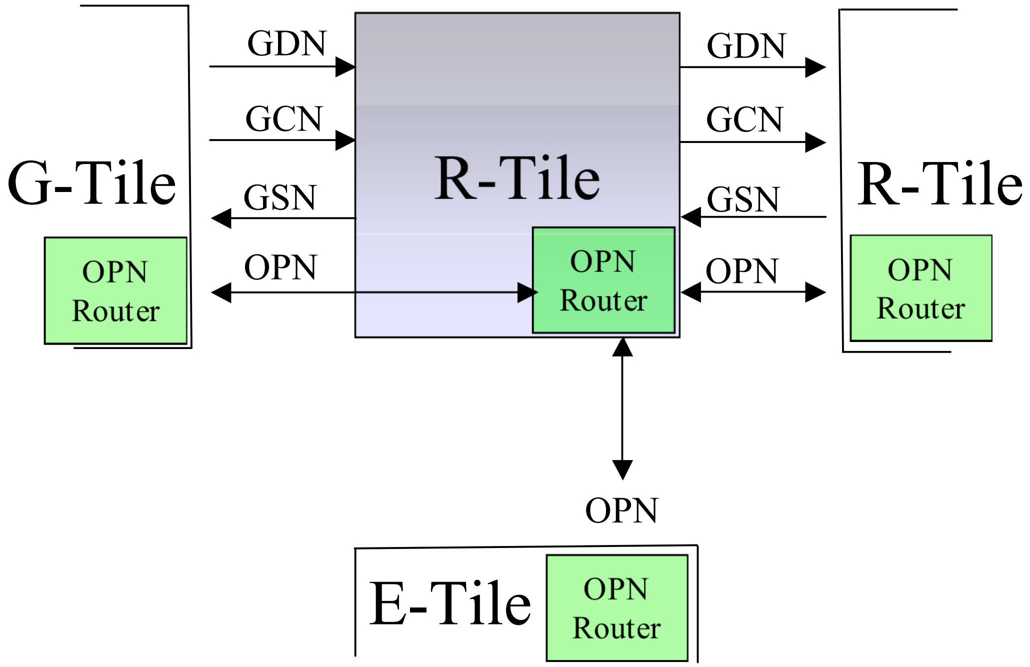
ISCA-32 Tutorial

Karu Sankaralingam

Department of Computer Sciences
The University of Texas at Austin

R-Tile and OPN Location and Connections

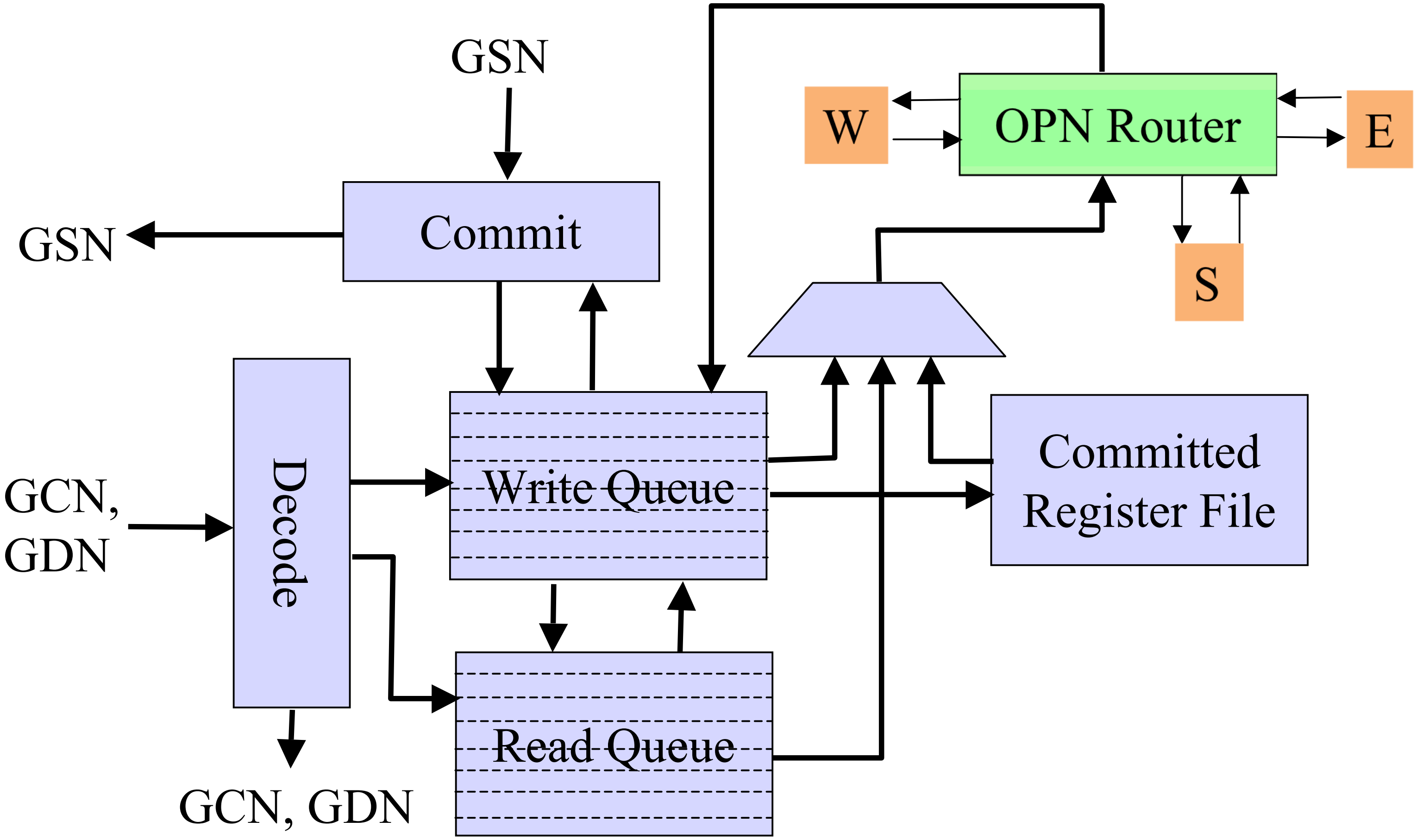




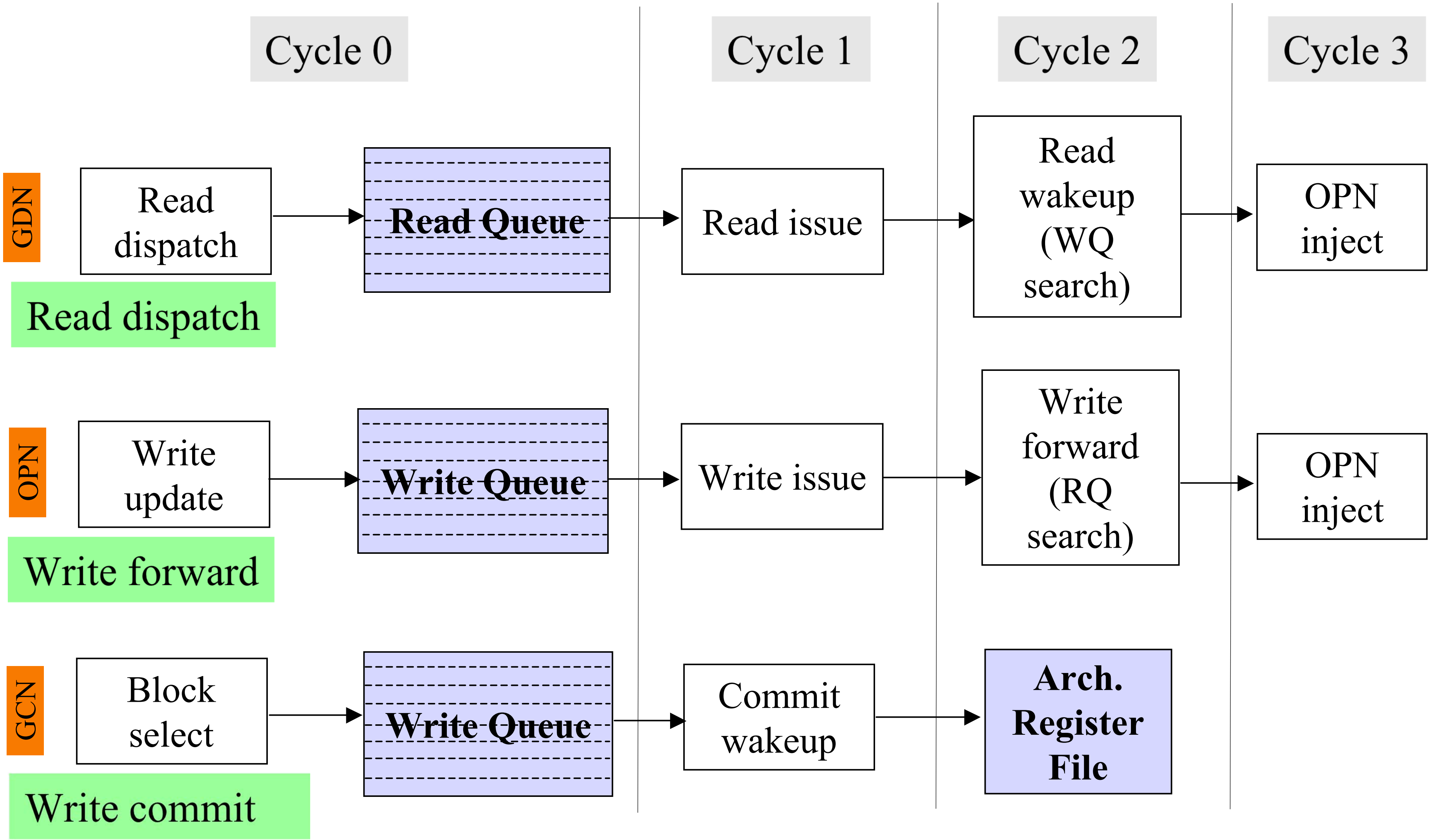
Major R-Tile Responsibilities

- 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

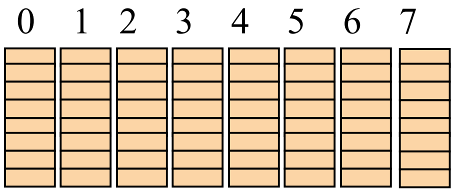
Major Structures in the R-tile



Pipeline diagrams: Register Read and Write

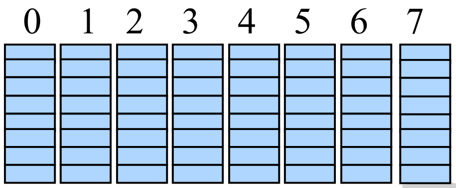


Register Forwarding: RQ and WQ contents



Read Queue

Valid	Status	Reg #	Target 1	Target 2	WQ wait	WQ pointer
1	3	5	8	8	1	6



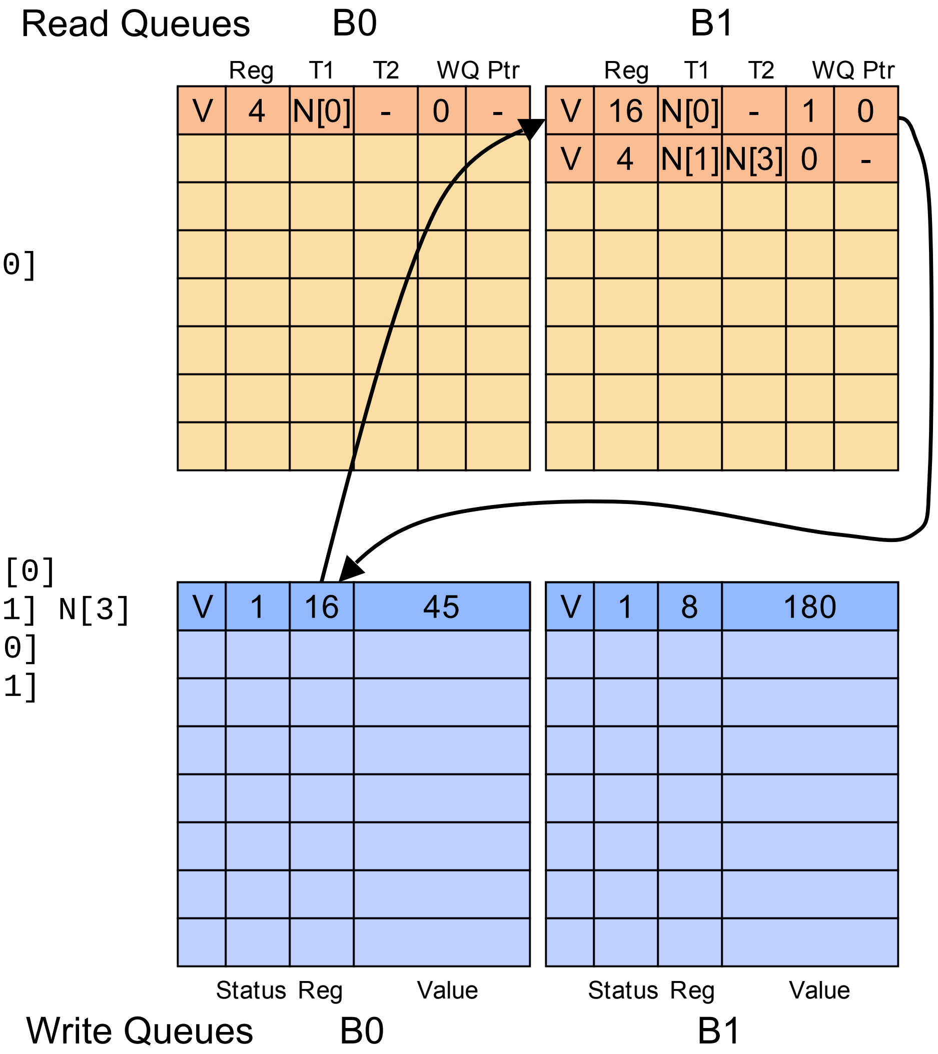
Write Queue

Valid	Status	Reg #	Value
1	8	5	64

Register Forwarding Example

```
.bbegin B0
R[0] read G[4] N[0]
N[0] addi 45 W[0]
W[0] write G[16]
.bend
```

```
.bbegin B1
R[0] read G[16] N[0]
R[4] read G[4] N[1] N[3]
N[0] addi 45 N[2,0]
N[1] addi 90 N[2,1]
N[2] add N[3,1]
N[3] add W[0]
W[0] write G[8]
.bend
```



Read Queues B0

	Reg	T1	T2	WQ Ptr	
V	4	N[0]	-	0	-

Read Queues B1

	Reg	T1	T2	WQ Ptr	
V	16	N[0]	-	1	0
V	4	N[1]	N[3]	0	-

Write Queues B0

Status		Reg	Value
V	1	16	45

Write Queues B1

Status		Reg	Value
V	1	8	180

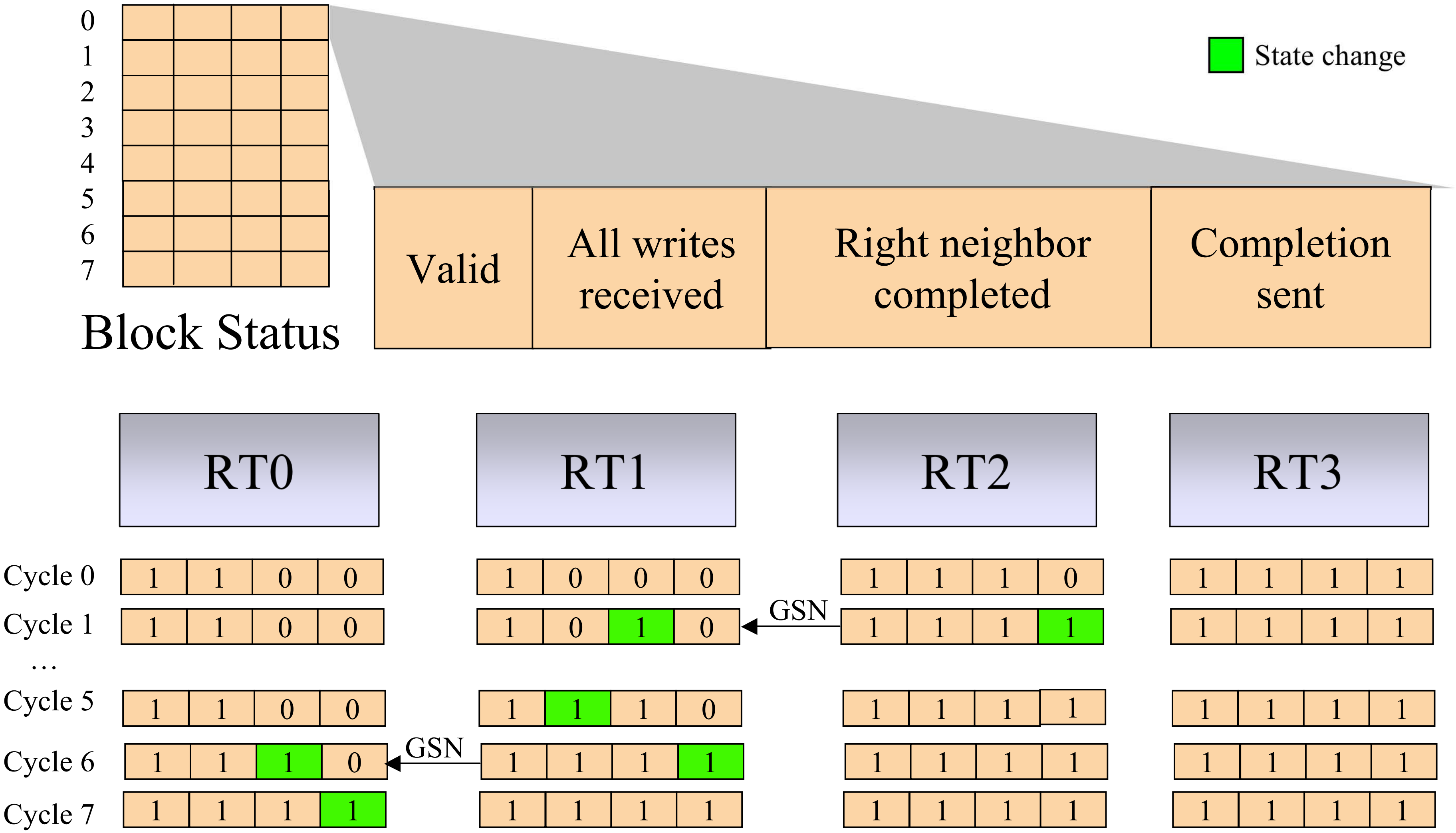
B0:
- R[0],W[0]: dispatch
- R[0]: search WQ, read from CRF

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

B0:
- W[0] receives value at WQ
- Wakes up R[0] in B1
- Forwards ***45*** to N[0]

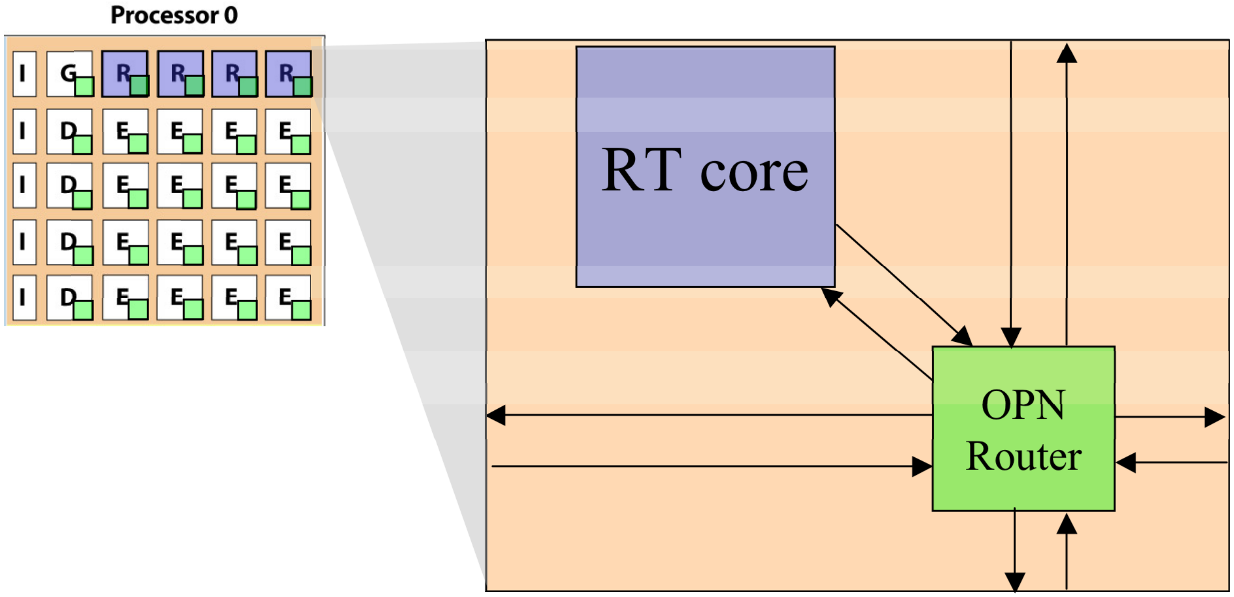
B1:
- N[0]…N[3] execute
- W[0] receives value at WQ

Block completion: Commit unit



	RT0	RT1	RT2	RT3
Cycle 0	1 1 0 0	1 0 0 0	1 1 1 0	1 1 1 1
Cycle 1	1 1 0 0	1 0 🟩1 0	1 1 1 🟩1	1 1 1 1
…				
Cycle 5	1 1 0 0	1 🟩1 1 0	1 1 1 1	1 1 1 1
Cycle 6	1 1 🟩1 0	1 1 1 🟩1	1 1 1 1	1 1 1 1
Cycle 7	1 1 1 🟩1	1 1 1 1	1 1 1 1	1 1 1 1

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
4 -	PC read
5 -	PC write
14 -	Special register read
15 -	Special register write

Control packet

1	4	3	6	5	6	5
valid	type	frame id	dst. node	dst. index	src. node	src. index

dst. node: XXX.YYY

src. node: XXX.YYY

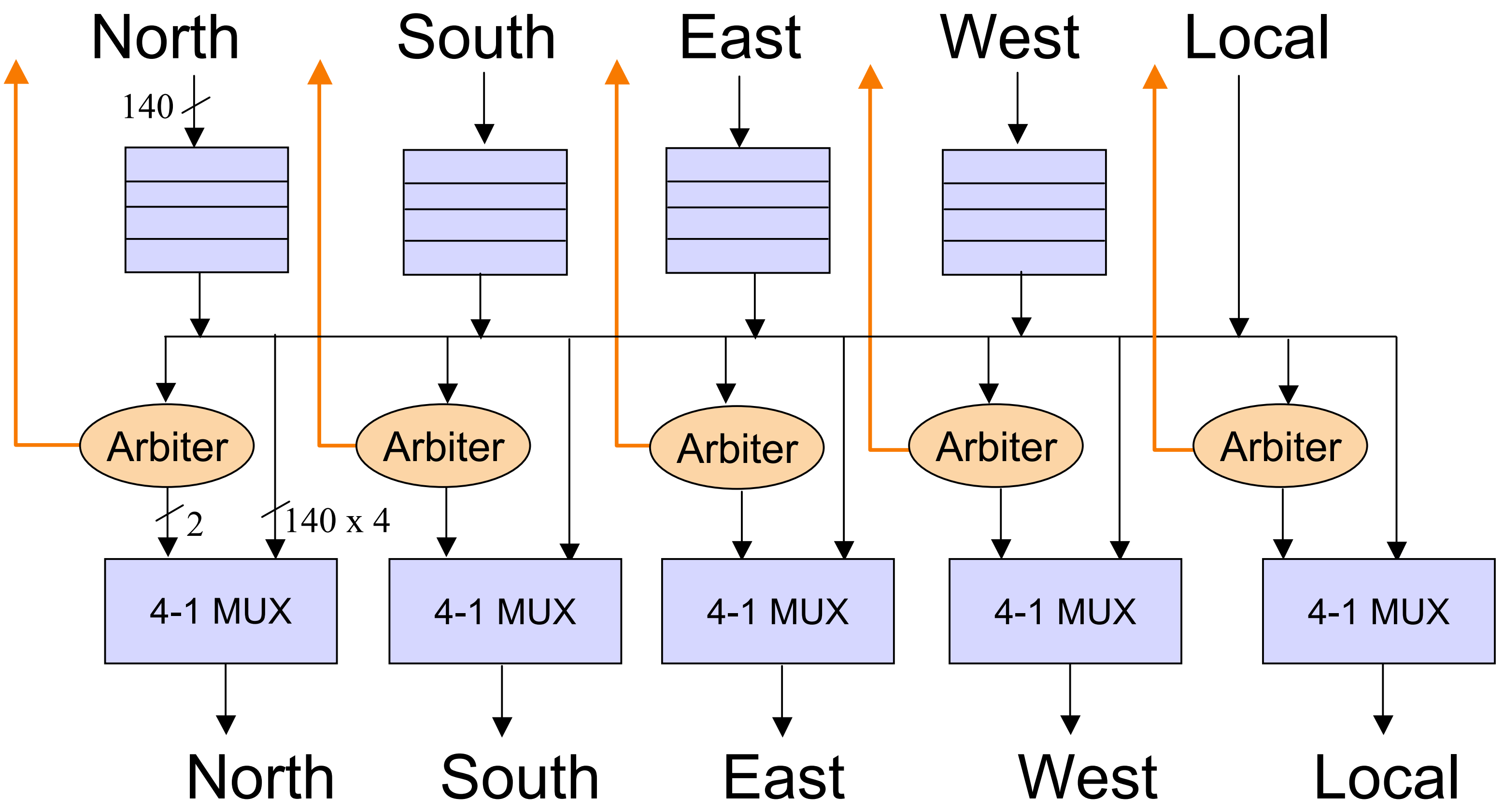
Data packet

1	2	64	40	3
valid	type	value	address	operation

valid, type: Normal, Null, Exception

operation: lb, lh, lw, ld
sb, sh, sw, sd

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]

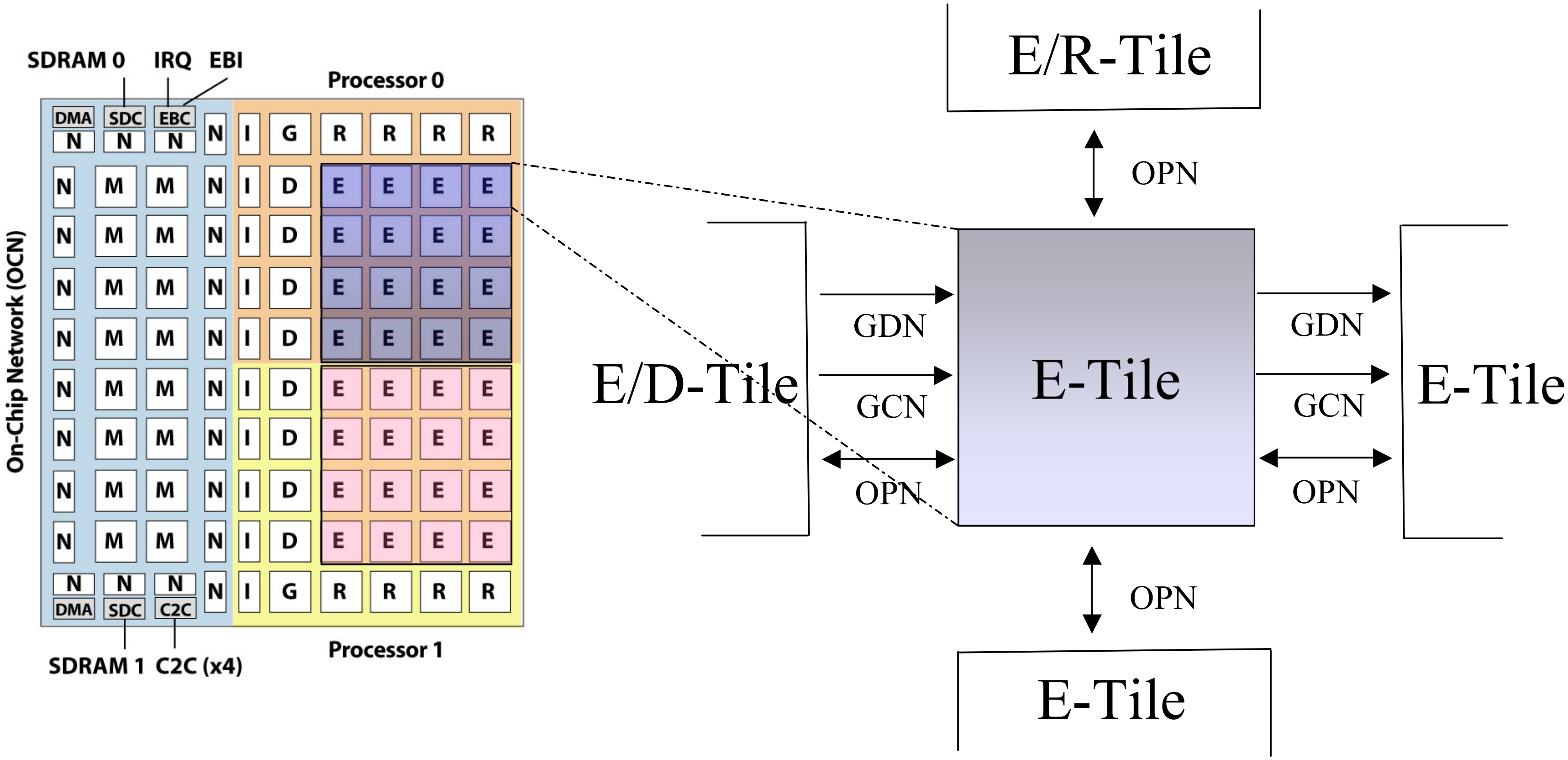
Issue Logic and Execution

ISCA-32 Tutorial

Premkishore Shivakumar

Department of Computer Sciences
The University of Texas at Austin

E-Tile Location and Connections



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

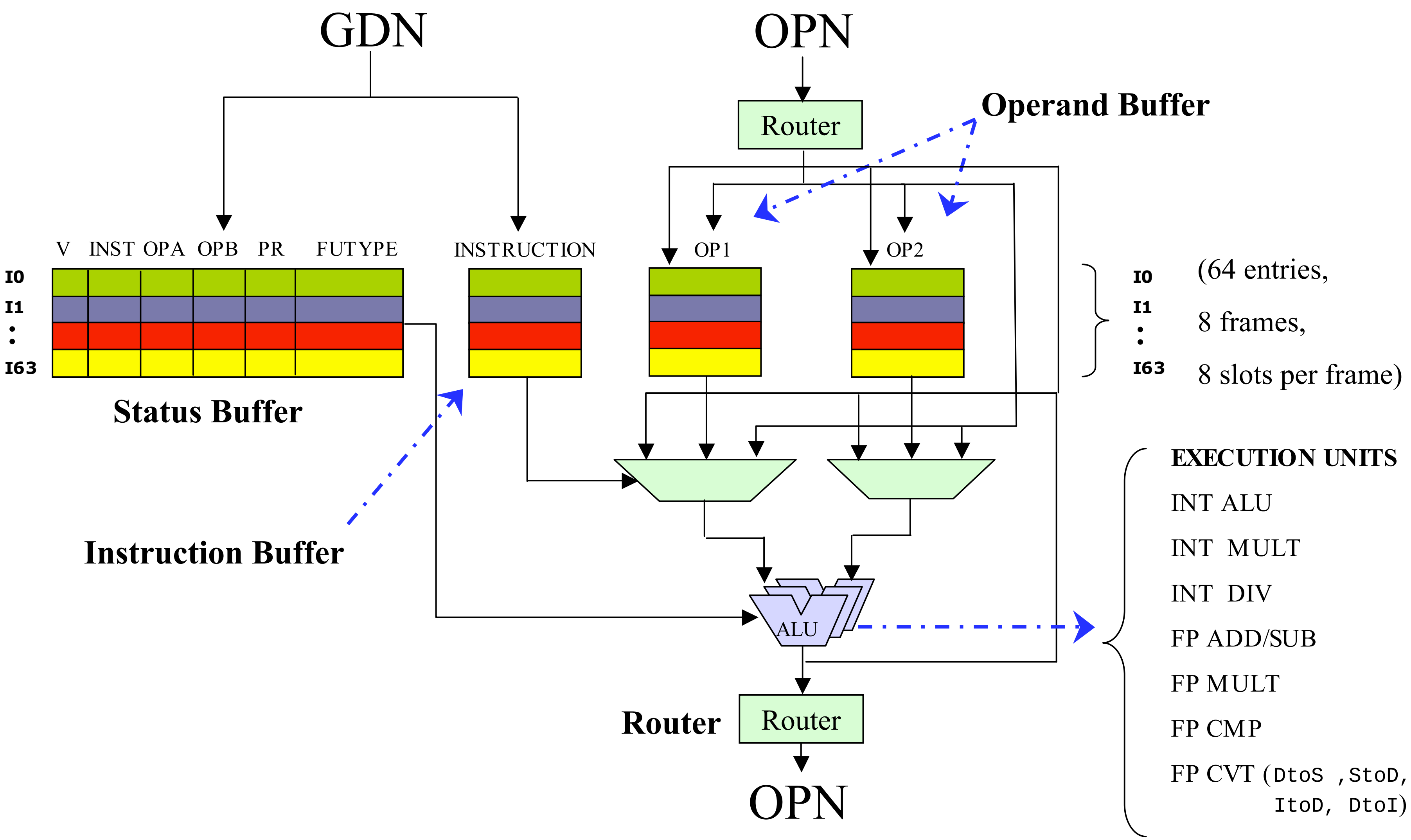
- **GDN** - Global Dispatch Network
  - Receive instructions from I-tile, and the dispatch commands from G-tile
    - Through a D-tile or a neighboring E-tile

- **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



(64 entries,

8 frames,

8 slots per frame)

EXECUTION UNITS

INT ALU

INT MULT

INT DIV

FP ADD/SUB

FP MULT

FP CMP

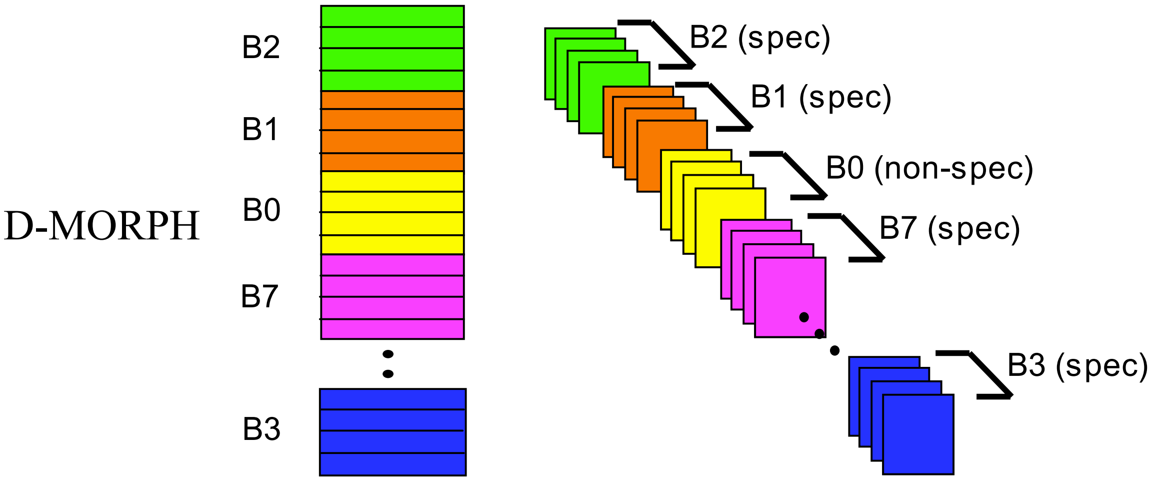
FP CVT (DtoS ,StoD,
ItoD, DtoI)

Functional Unit Latencies in the E-Tile

Functional Unit	**Latency**	**Implementation**
Integer ALU	1	Synopsys IP
Integer Multiplier	3 (Pipelined)	Synopsys IP
Integer Divider	24 (Iterative)	Synopsys IP
FP Add / Sub	4 (Pipelined)	Synopsys IP
FP Multiplier	4 (Pipelined)	Synopsys IP
FP Comparator	2 (Pipelined)	Synopsys IP
FP DtoI Convert	2 (Pipelined)	Synopsys IP
FP ItoD Convert	3 (Pipelined)	Synopsys IP
FP StoD Convert	2 (Pipelined)	UT TRIPS IP
FP DtoS Convert	3 (Pipelined)	UT TRIPS IP

Issue Prioritization Among Ready Instructions

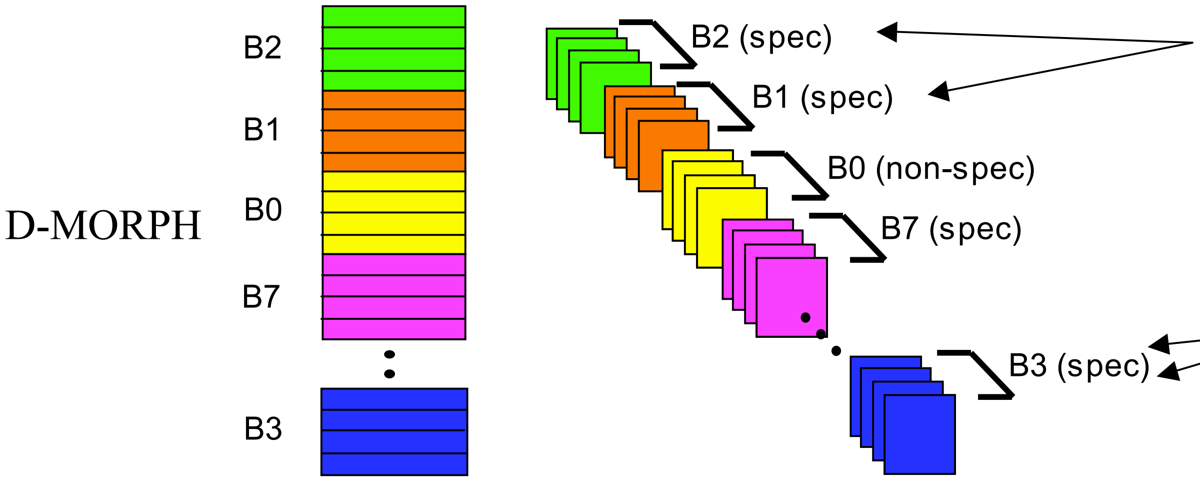
Reservation stations at one E-Tile



- 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

Reservation stations at one E-Tile



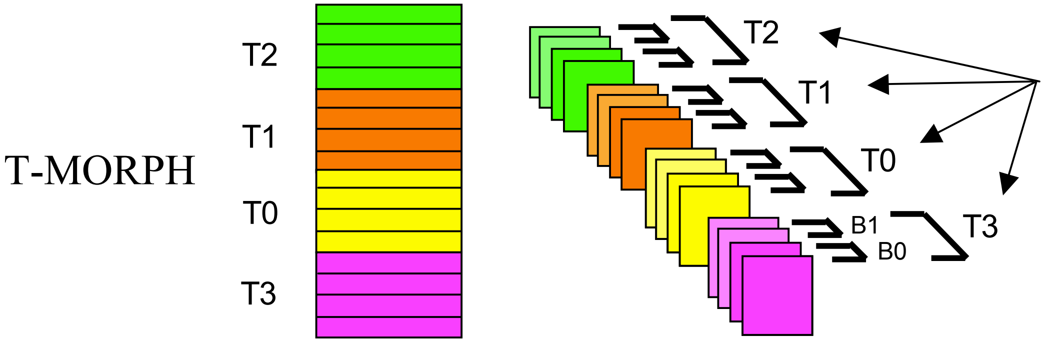
Older Block First

B1 before B2

Smaller Instruction

Slot First

B3.I0 before B3.I2

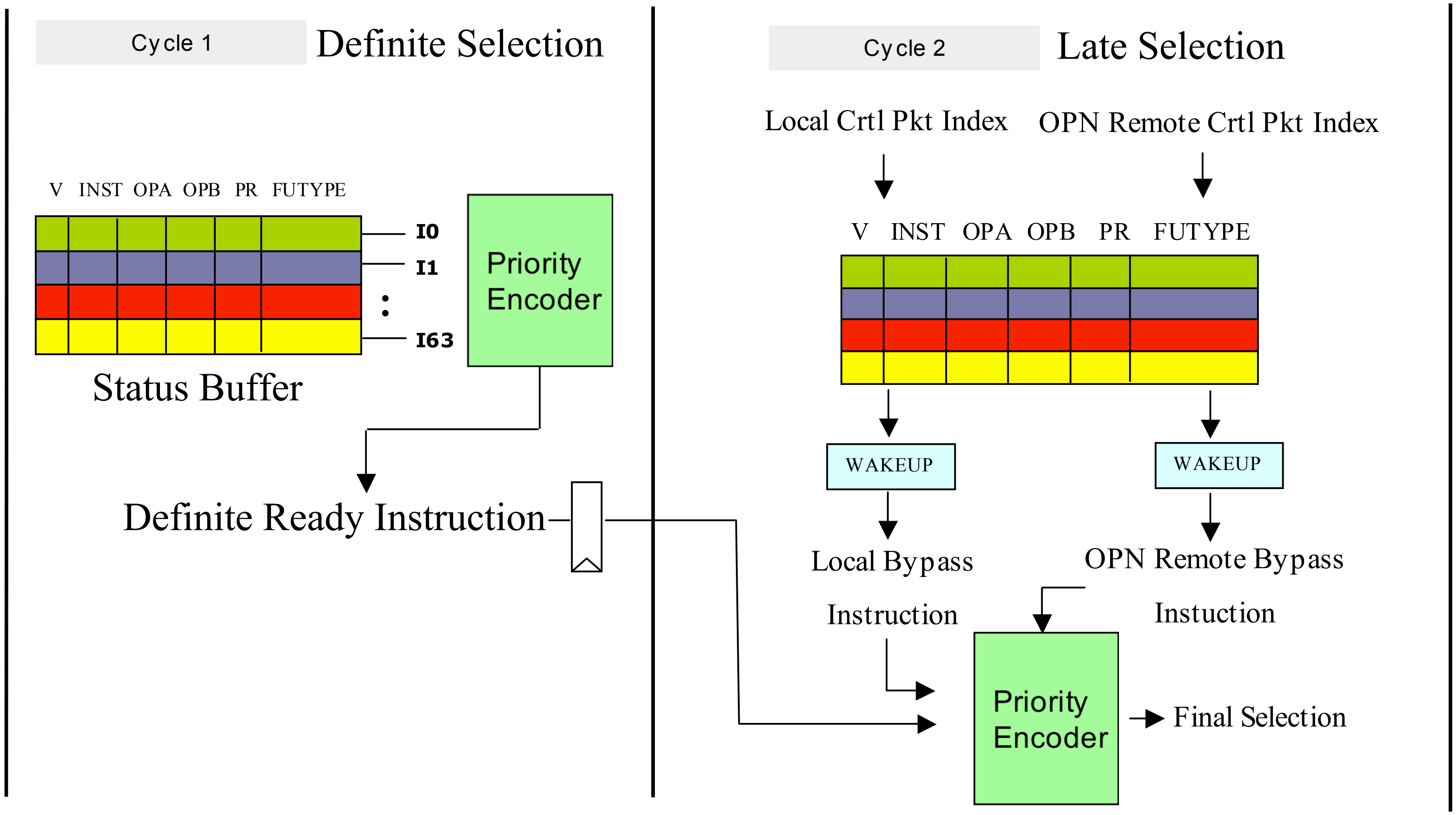


Three-level Select Priority:

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

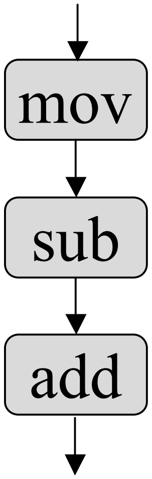
Two Phase Instruction Selection

Two Phase Instruction Selection

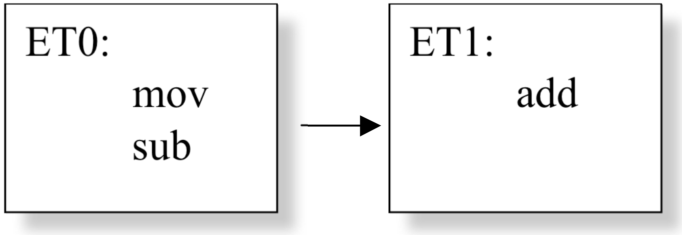


- Priority Encoder uses *Issue Prioritization Policy*
- Wakeup: Back-to-back operand delivery, and two bypassed operands in the same cycle

Basic Operand Local & Remote Bypass



Sample Dependence Graph



Instruction Physical Placement

Local Bypass **mov ⇨ sub** Remote Bypass **sub ⇨ add**	Cycle 1	Cycle 2	Cycle 3	Cycle 4
	• Execute *mov* • Send *mov* Local Bypass Ctrl Pkt • Wakeup, Select *sub*	• Receive *mov* Local Bypass Data Pkt • Execute *sub* • Send *sub* OPN Ctrl Pkt from ET0 to ET1	• Send *sub* OPN Data Pkt from ET0 to ET1 • Wakeup, Select *add*	• Receive *sub* OPN Data Pkt • Execute *add*

Local Bypass

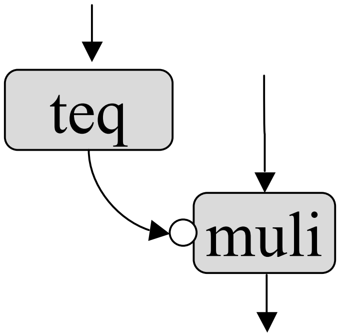
Remote Bypass (extra OPN cycle)

Bypass: Sources of Pipeline Bubbles

- OPN stall due to network congestion

	Cycle 1	Cycle 2		Cycle X	Cycle X+1
Local Bypass **mov ⇨ sub** Remote Bypass **sub ⇨ add**	• Execute *mov* • Send *mov* Local Bypass Ctrl Pkt • Wakeup, Select *sub*	• Receive *mov* Local Bypass Data Pkt • Execute *sub* • Send *sub* OPN Ctrl Pkt from ET0 to ET1	• • • OPN Stall	• Send *sub* OPN Data Pkt from ET0 to ET1 • Wakeup, Select *add*	• Receive *sub* OPN Data Pkt • Execute *add*

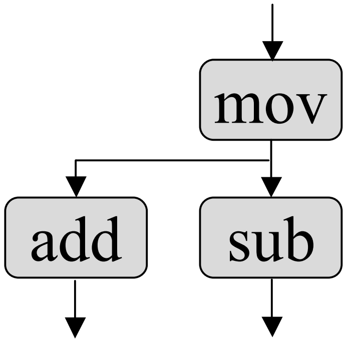
- Bypassed operand data is a *non-enabling* predicate
  - –Detected in execute stage after receiving bypassed data => Issue slot wasted
  - –Similar pipeline bubble due to invalid bypassed OPN data packet



ET0:
teq
muli

	Cycle 1	Cycle 2
Local Bypass **teq ⇨ muli**	• Execute *teq* • Send *teq* Local Bypass Ctrl Pkt • Wakeup, Select *muli*	• Receive *teq* Local Bypass Data Pkt • Data is an *Non-enabling* Predicate Pkt • Issue slot wasted

Target Delivery: Multiple Local Targets

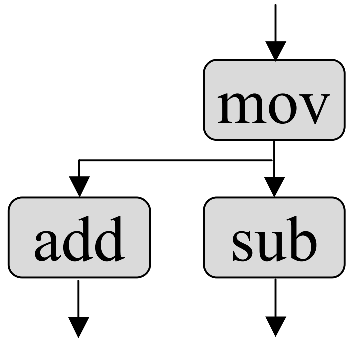


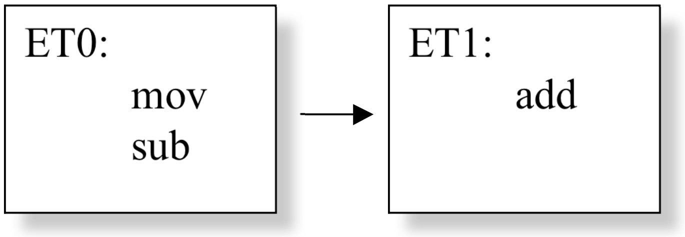
ET0:
mov
sub
add

	Cycle 1	Cycle 2	Cycle 3
Local Bypass **mov ⇨ sub** Local Bypass **mov ⇨ add**	• Execute *mov* • Send *mov* Local Bypass Ctrl Pkt1 • Wakeup, Select *sub*	• Receive *mov* Local Bypass Data Pkt1 • Execute *sub* • Send *mov* Local Bypass Ctrl Pkt2 • Wakeup, Select *add*	• Receive *mov* Local Bypass Data Pkt2 • Execute *add*

- Successive local targets delivered in consecutive cycles
- Successive remote targets also delivered in consecutive cycles (extra OPN forward cycle)

Concurrent Local & Remote Target Delivery





Local Bypass

mov ⇨ sub

Remote Bypass

mov ⇨ add

Cycle 1	Cycle 2	Cycle 3
• Execute *mov* • Send *mov* Local Bypass Ctrl Pkt • Wakeup, Select *sub* • Send *mov* OPN Ctrl Pkt from ET0 to ET1	• Receive *mov* Local Bypass Data Pkt1 • Execute *sub* • Send *mov* OPN Data Pkt2 from ET0 to ET1 • Wakeup, Select *add*	• Receive *mov* OPN Data Pkt2 • Execute *add*

# cycles to emit targets (Local/Remote)

LL	**2**
LR	**1**
RR	**2**
LLL	**3**
LLR	**2**
LRR	**2**
RRR	**3**
LLLL	**4**
LLLR	**3**
LLRR	**2**
LRRR	**3**
RRRR	**4**

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

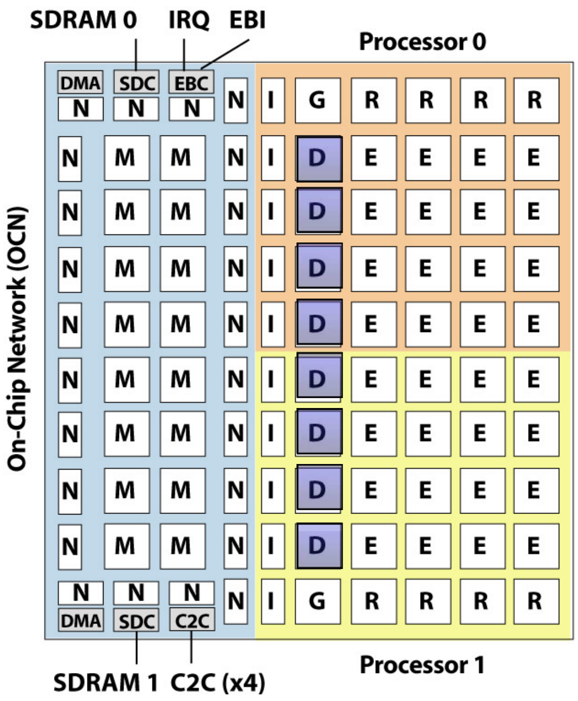
Primary Memory System

ISCA-32 Tutorial

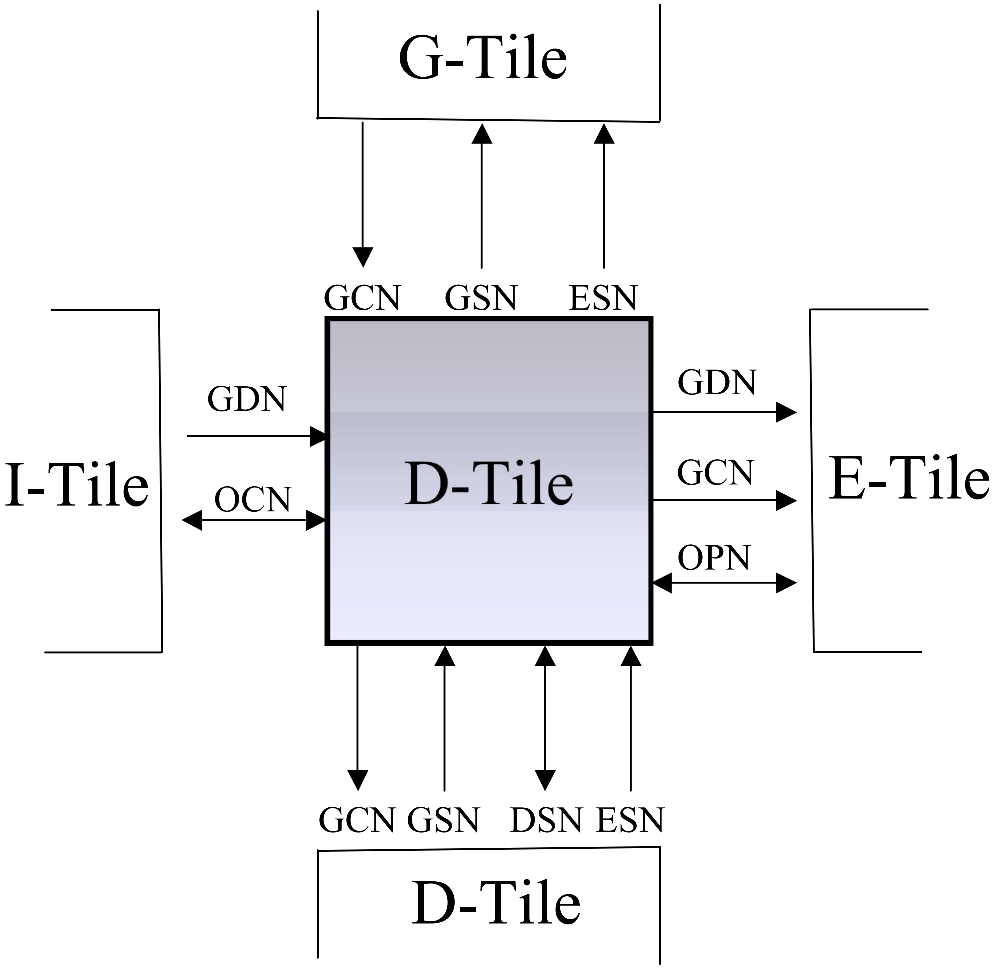
Simha Sethumadhavan

Department of Computer Sciences
The University of Texas at Austin

D-Tile Location and Connections



DMA N	SDC N	EBC N	N	I	G	R	R	R	R
N	M	M	N	I	D	E	E	E	E
N	M	M	N	I	D	E	E	E	E
N	M	M	N	I	D	E	E	E	E
N	M	M	N	I	D	E	E	E	E
N	M	M	N	I	D	E	E	E	E
N	M	M	N	I	D	E	E	E	E
N	M	M	N	I	D	E	E	E	E
N	M	M	N	I	D	E	E	E	E
N DMA	N SDC	N C2C	N	I	G	R	R	R	R



Major D-Tile Responsibilities

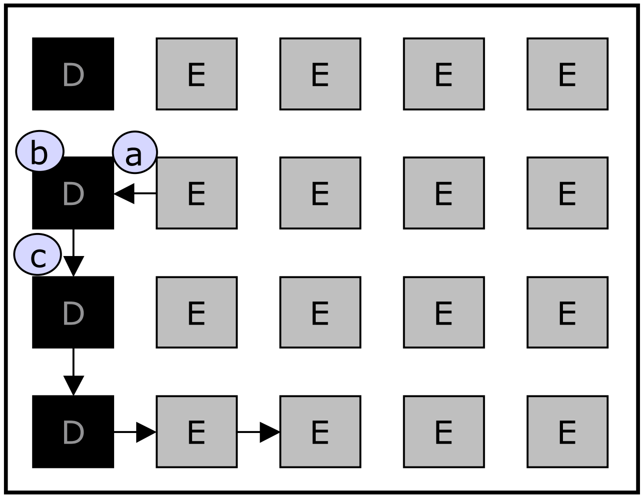
- Provide D-cache access for arriving loads and stores
- Perform address translation with DTLB
- Handle cache misses with MSHRs
- Perform dynamic memory disambiguation with load/store queues
- Perform dependence prediction for aggressive load/store issue
- Detect per-block store completion
- Write stores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

Primary Memory Latencies

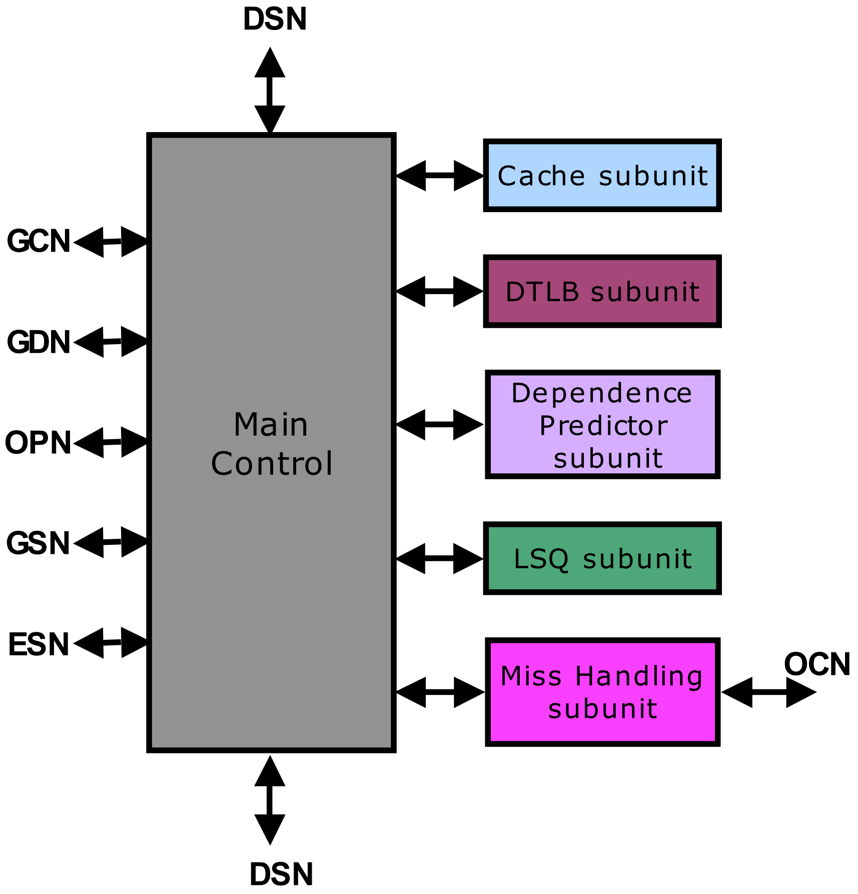
- Data cache is interleaved on a cache line basis (64 bytes)

- 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



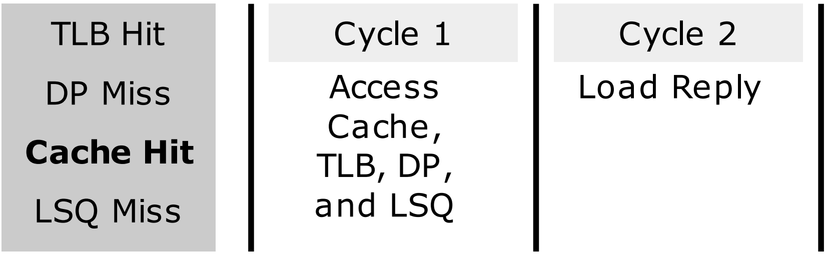
- Caches
  - 8KB, 2-way, 1R and 1W port
  - 64 byte lines, LRU, Virtually indexed, physically tagged,
  - Write-back, Write-no-allocate
- DTLB
  - 16 entries, 64KB to 1TB pages
- Dependence Predictor
  - 1024 bits, PC based hash function
- Load Store Queues
  - 256 entries, 32 loads/stores per block, 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

TLB	Dep. Pr	LSQ	Cache	Response
Miss	-	-	-	Report TLB Exception
Hit	**Wait (Hit)**	-	-	Defer load until all prior stores are received. Non deterministic latency
Hit	Miss	Miss	**Hit**	Forward data from L1 Cache
Hit	Miss	Miss	**Miss**	Forward data from L2 Cache (or memory) Issue cache fill request (if cacheable)
Hit	Miss	**Hit**	Hit	Forward data from LSQ
Hit	Miss	**Hit**	Miss	Forward data from LSQ Issue cache fill request (if cacheable)

Load Pipeline

- Common case: 2 cycle load hit latency



TLB Hit

DP Miss

Cache Hit

LSQ Miss

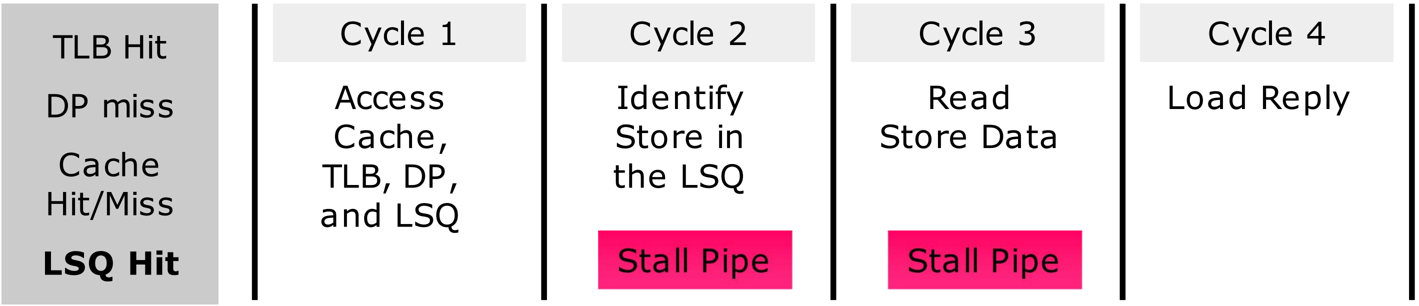
Cycle 1

Access Cache, TLB, DP, and LSQ

Cycle 2

Load Reply

- 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)



TLB Hit

DP miss

Cache Hit/Miss

LSQ Hit

Cycle 1

Access Cache, TLB, DP, and LSQ

Cycle 2

Identify Store in the LSQ

Stall Pipe

Cycle 3

Read Store Data

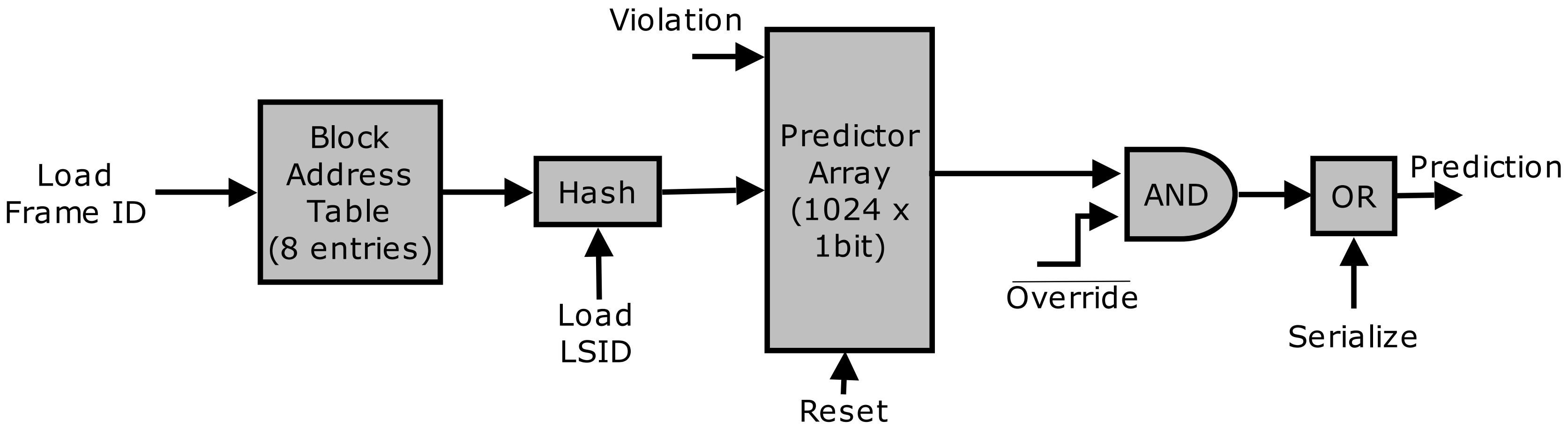
Stall Pipe

Cycle 4

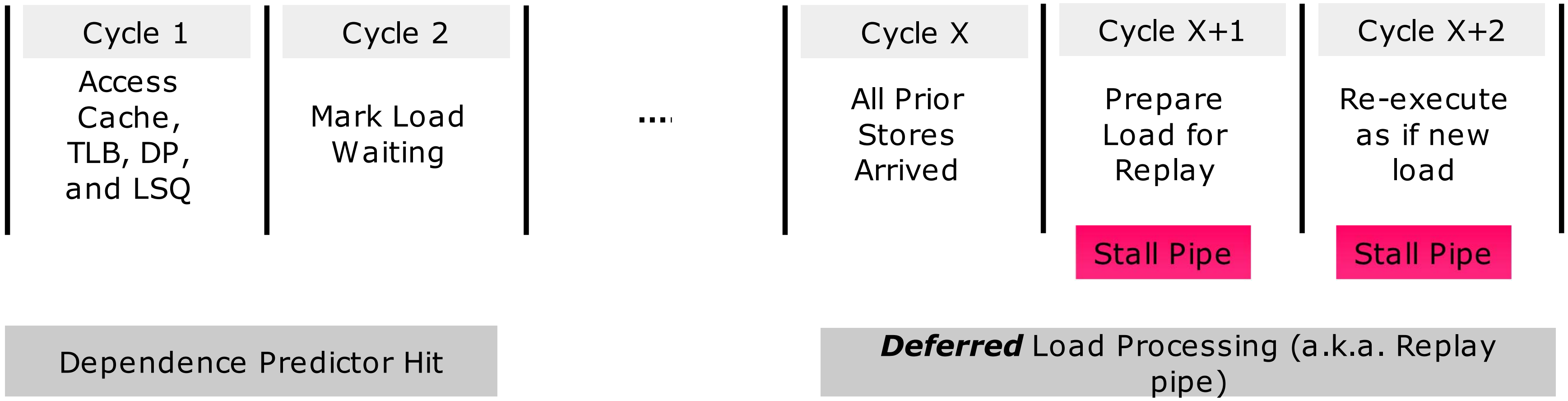
Load Reply

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


Violation
Load Frame ID
Block Address Table (8 entries)
Hash
Load LSID
Predictor Array (1024 x 1bit)
Reset
AND
$\overline{\text{Override}}$
OR
Serialize
Prediction

Deferred Load Pipeline



Cycle 1

Access Cache, TLB, DP, and LSQ

Cycle 2

Mark Load Waiting

....

Cycle X

All Prior Stores Arrived

Cycle X+1

Prepare Load for Replay

Stall Pipe

Cycle X+2

Re-execute as if new load

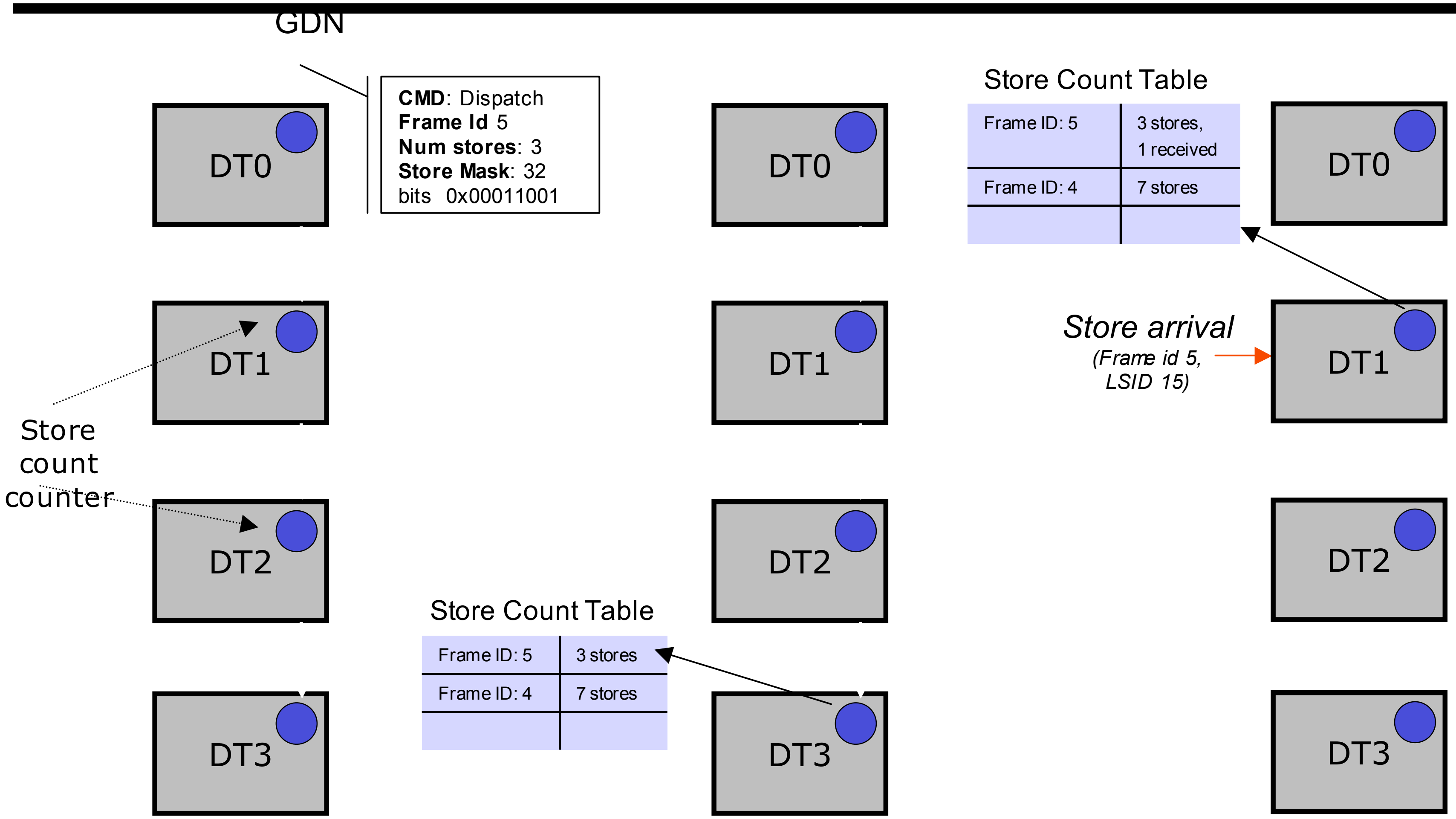
Stall Pipe

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

Store Counting



CMD: Dispatch
Frame Id 5
Num stores: 3
Store Mask: 32 bits 0x00011001

Store Count Table

Frame ID: 5	3 stores
Frame ID: 4	7 stores

Store Count Table

Frame ID: 5	3 stores, 1 received
Frame ID: 4	7 stores

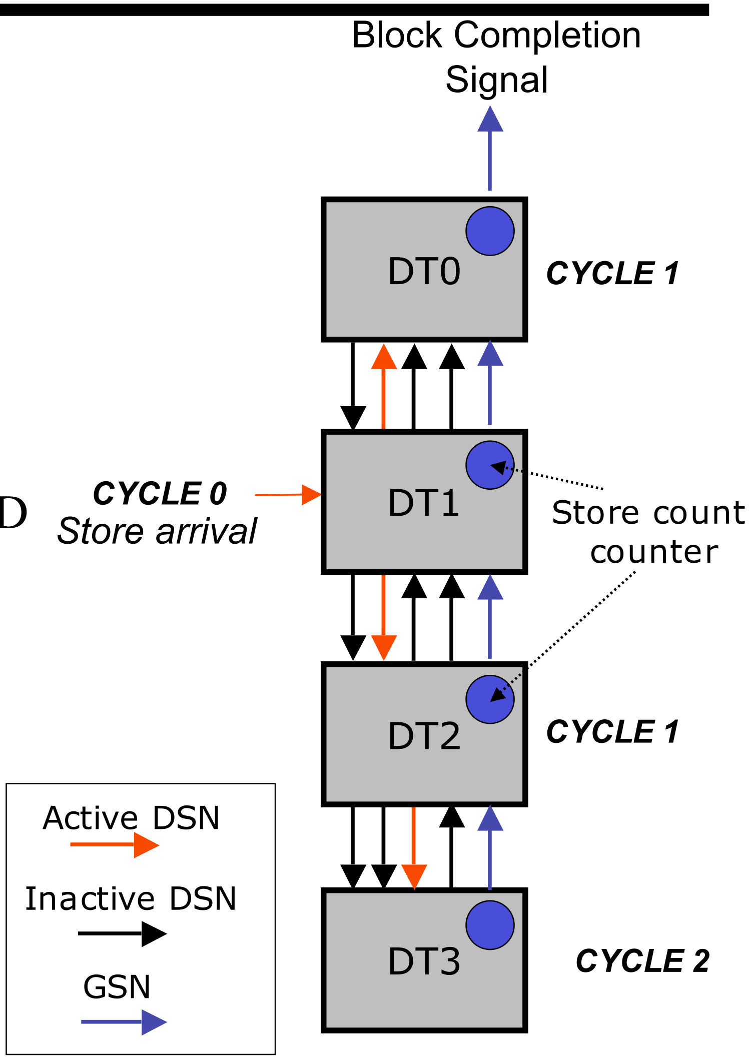
At block dispatch, each DT receives and records information on stores in the dispatched block

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)

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 in done in D-tile0 before sending it to G-tile
  - Messages sent on GSN



Store Commit Pipeline

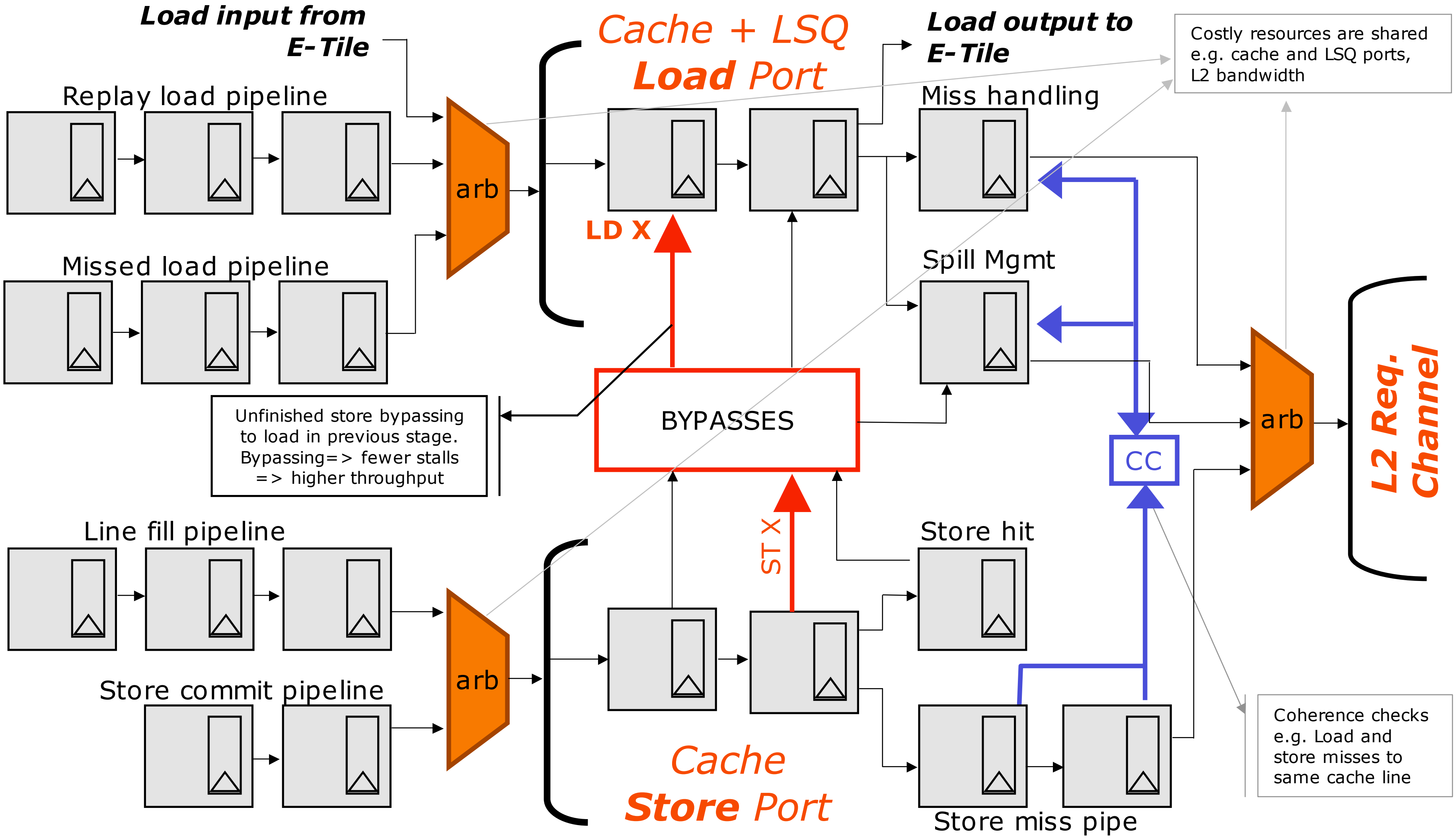
Store Commit	Cycle 1	Cycle 2	Cycle 3	Cycle 4	Cycle 5
	Pick Store to commit	Read Store data from LSQ	Access cache tags	Check for hit or miss	Write to Cache or Merge buffer
	Stall Pipe	Stall Pipe			

- 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

D-Tile Pipelines Block Diagram



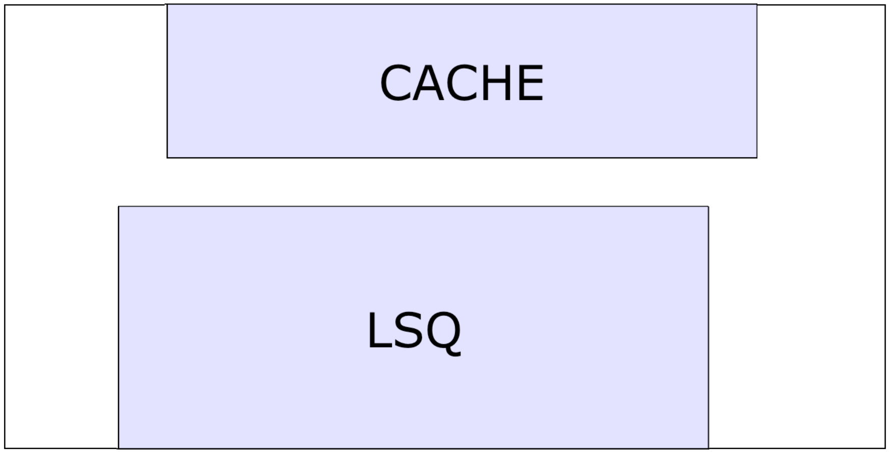
Costly resources are shared e.g. cache and LSQ ports, L2 bandwidth

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

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

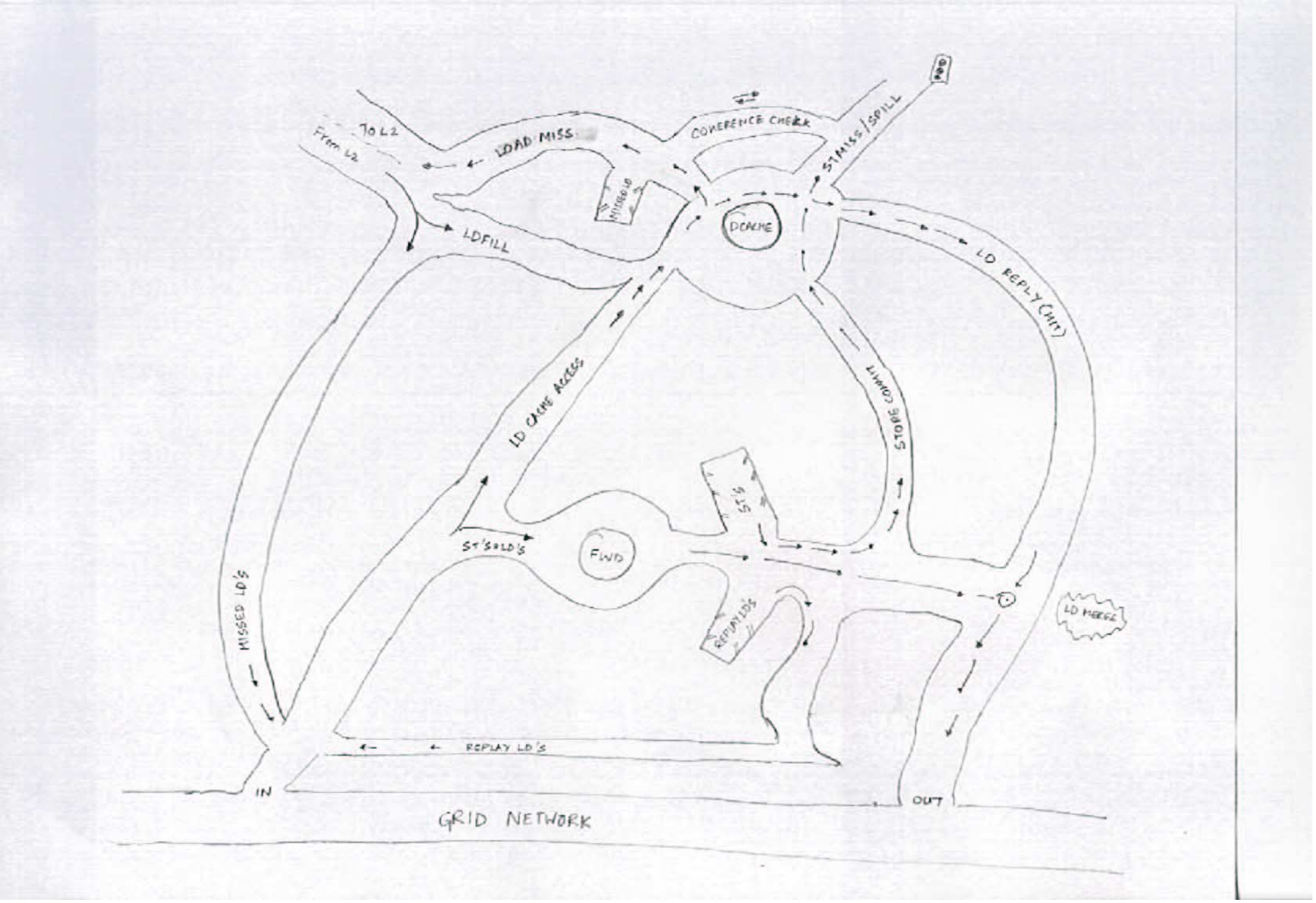
Memory System Design Challenges

- 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 ☺



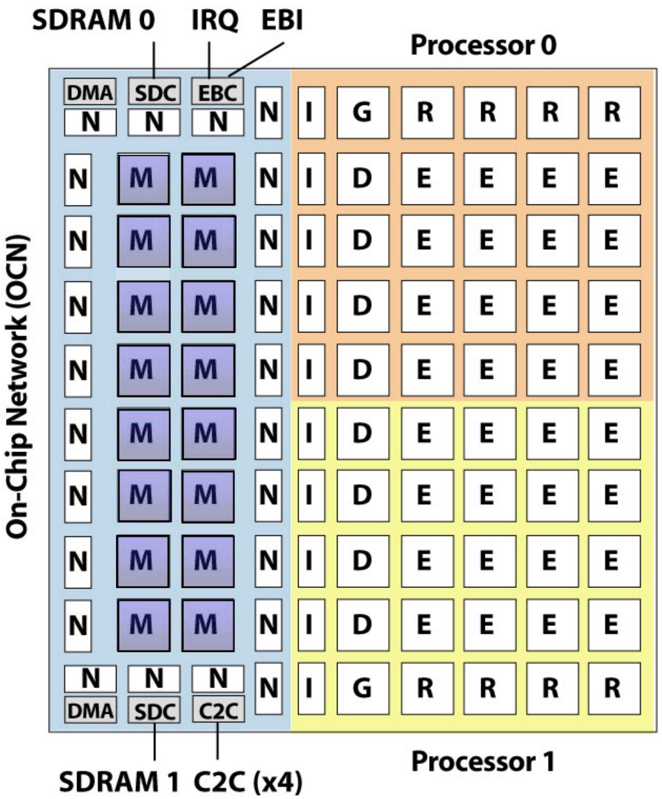
Secondary Memory System

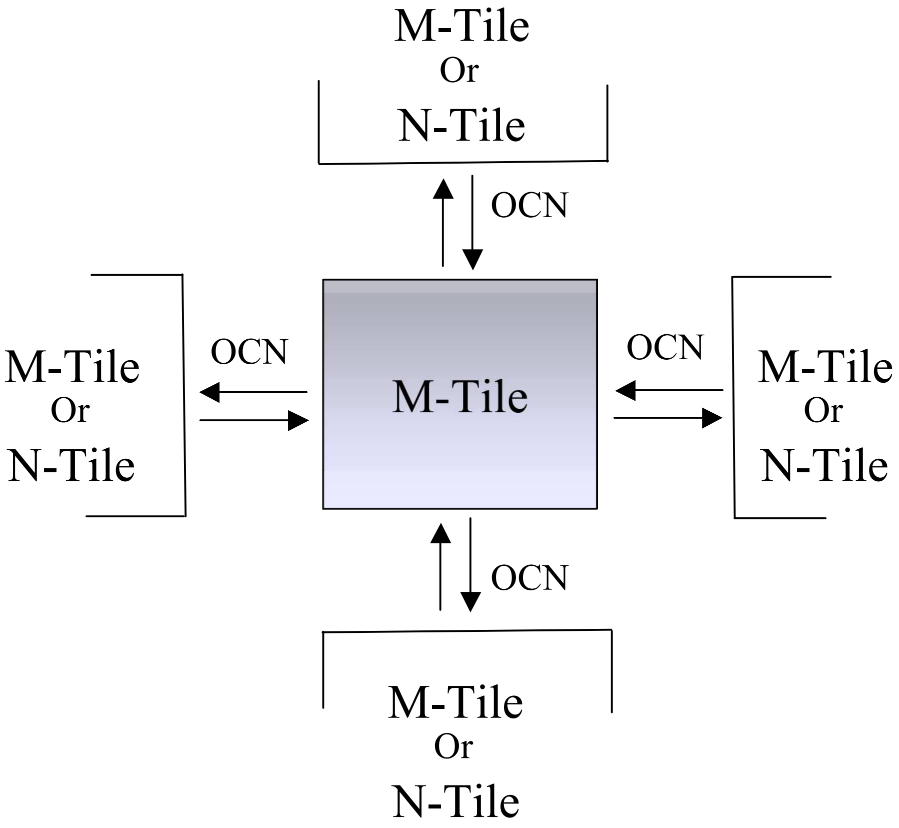
ISCA-32 Tutorial

Changkyu Kim

Department of Computer Sciences
The University of Texas at Austin

M-Tile Location and Connections

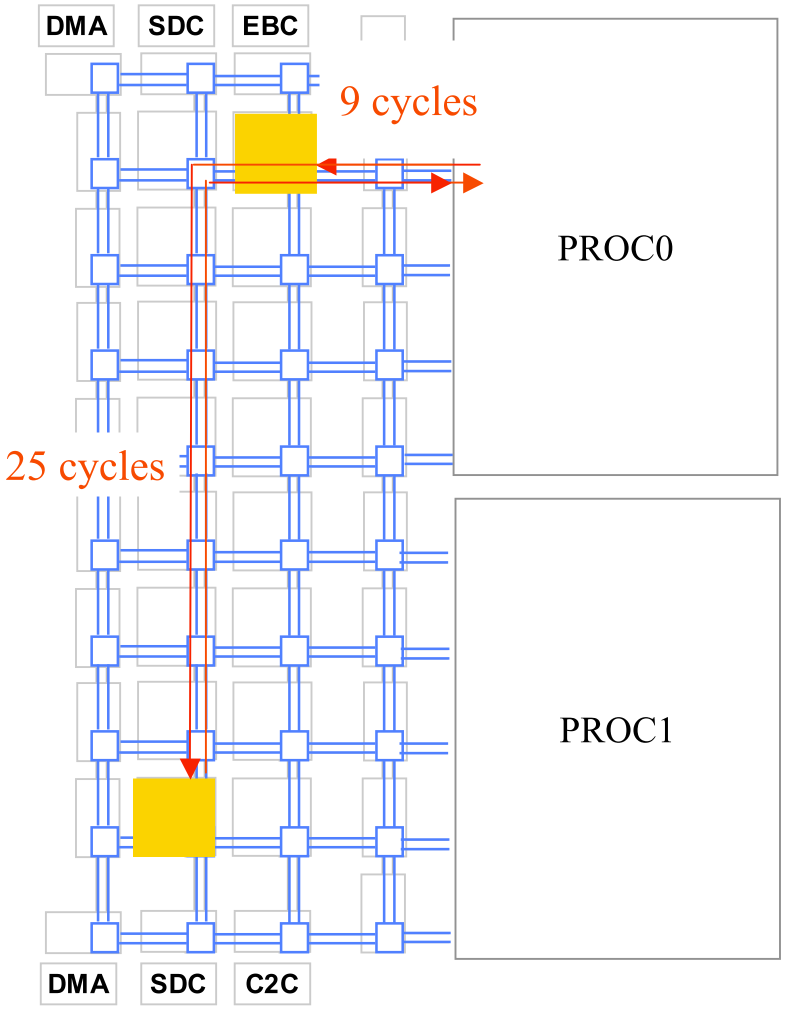




M-Tile Network Attributes

- 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)

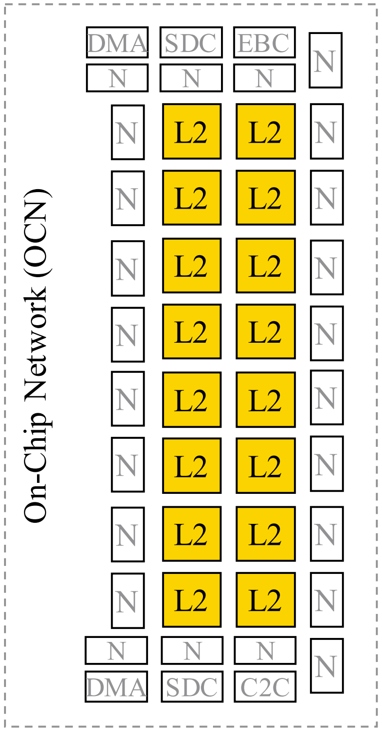


- Non-uniform access latency
  - Closer bank can be accessed faster
- Designed for wire-dominated future technologies
  - No global interconnect
- Connected by switched network
  - Can fit large number of banks with small area overhead
- Allows thousands of in-flight transactions
  - Network capacity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

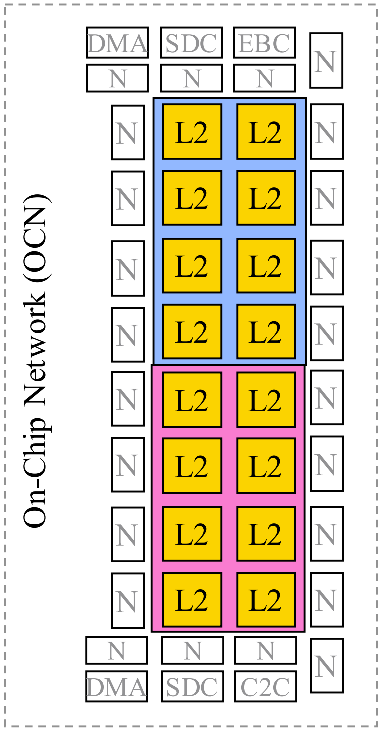
Multiple Modes in TRIPS Secondary Memory

L2 : M-Tile configured as a level-2 Cache

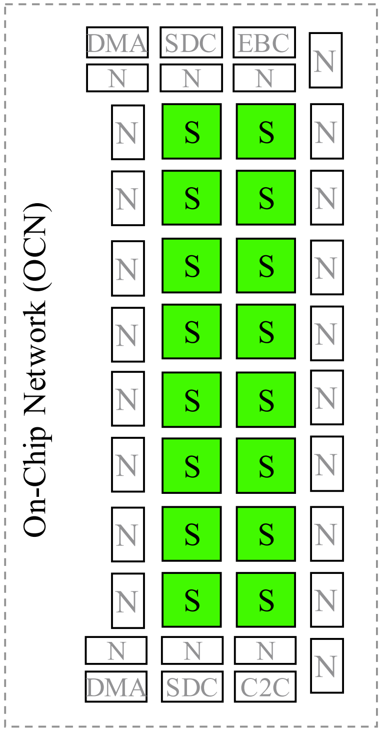
S : M-Tile configured as a scratchpad memory



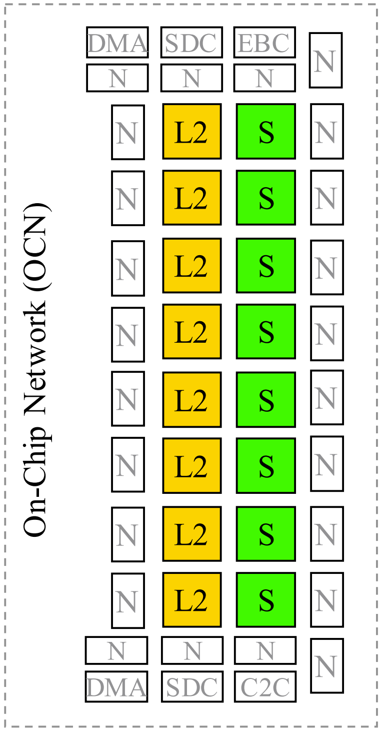
All shared
level-2 caches



Per-processor
level-2 caches

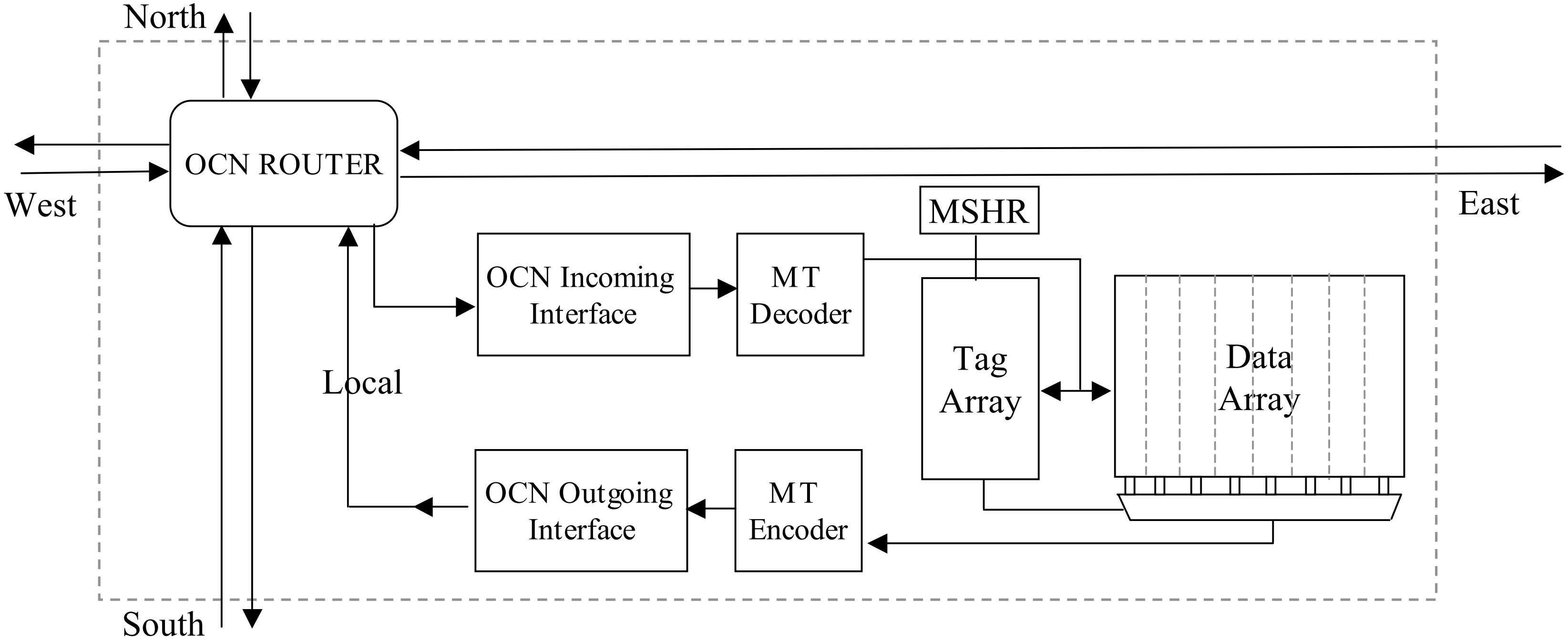


All scratchpad
memories



Half level-2 caches, half
scratchpad memories

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

Back-to-Back Read Pipeline

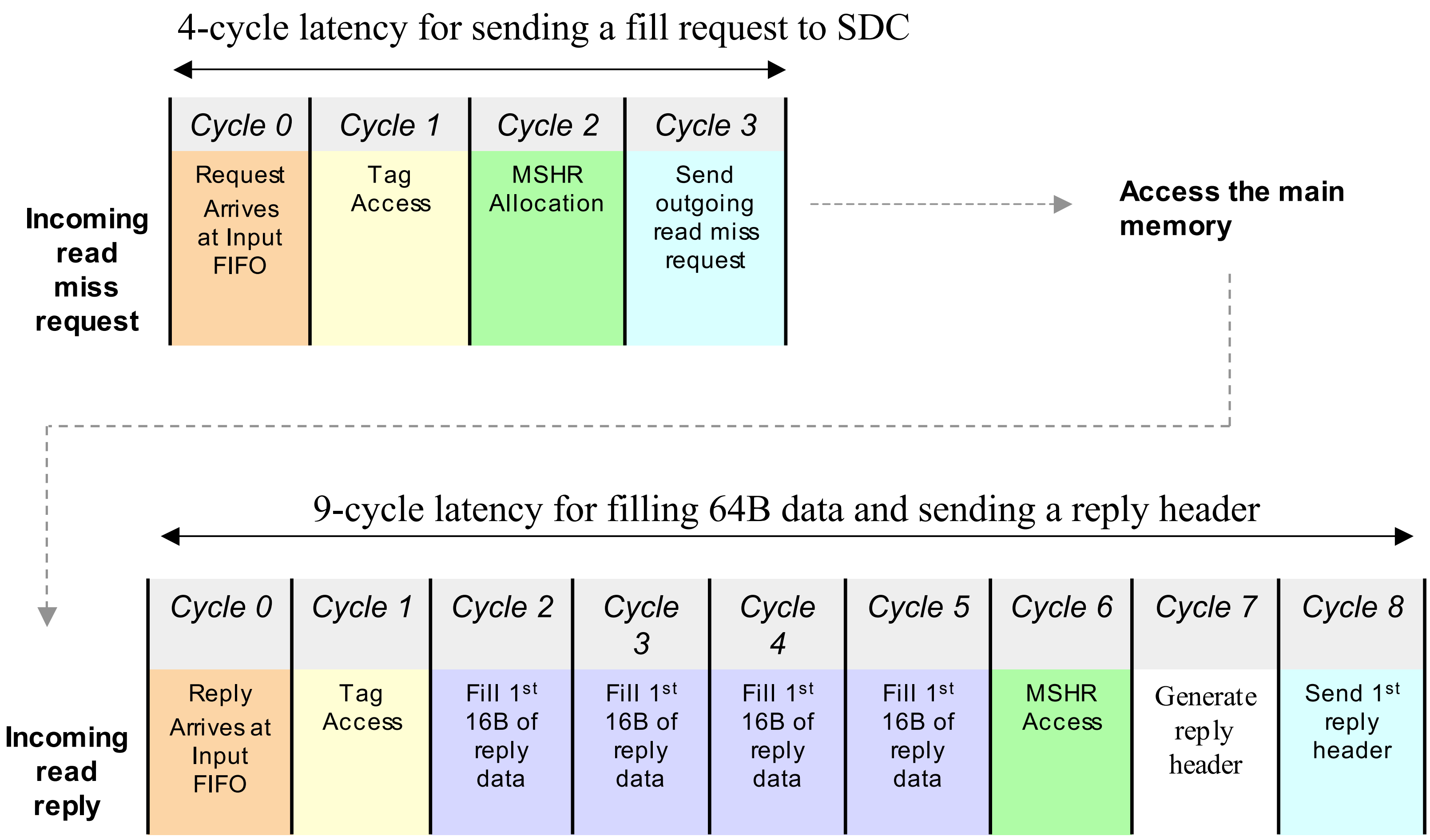
7-cycle latency for transferring all 64B data

4-cycle latency for sending a reply header

	Cycle 0	Cycle 1	Cycle 2	Cycle 3	Cycle 4	Cycle 5	Cycle 6	Cycle 7	Cycle 8	Cycle9
First Read	Request Arrives at Input FIFO	Tag Access Data Access 1	Data Access 2	Send reply header	Send 1st 16B of reply data	Send 2nd 16B of reply data	Send 3rd 16B of reply data	Send 4th 16B of reply data		
Second Read		Request Arrives at Input FIFO					Tag Access Data Access 1	Data Access 2	Send reply header	Send 1st 16B of reply data

Seamless Data Transfer (6.8GB/sec)

Read Miss Pipeline



4-cycle latency for sending a fill request to SDC

	Cycle 0	*Cycle 1*	*Cycle 2*	*Cycle 3*
Incoming read miss request	Request Arrives at Input FIFO	Tag Access	MSHR Allocation	Send outgoing read miss request

Access the main memory

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

	Cycle 0	*Cycle 1*	*Cycle 2*	*Cycle 3*	*Cycle 4*	*Cycle 5*	*Cycle 6*	*Cycle 7*	*Cycle 8*
Incoming read reply	Reply Arrives at Input FIFO	Tag Access	Fill 1st 16B of reply data	Fill 1st 16B of reply data	Fill 1st 16B of reply data	Fill 1st 16B of reply data	MSHR Access	Generate reply header	Send 1st reply header

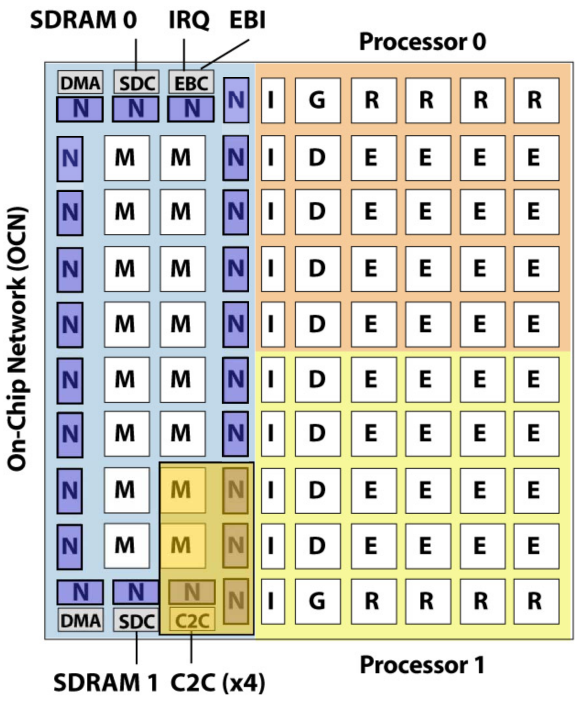
Chip- and System-Level Networks and I/O

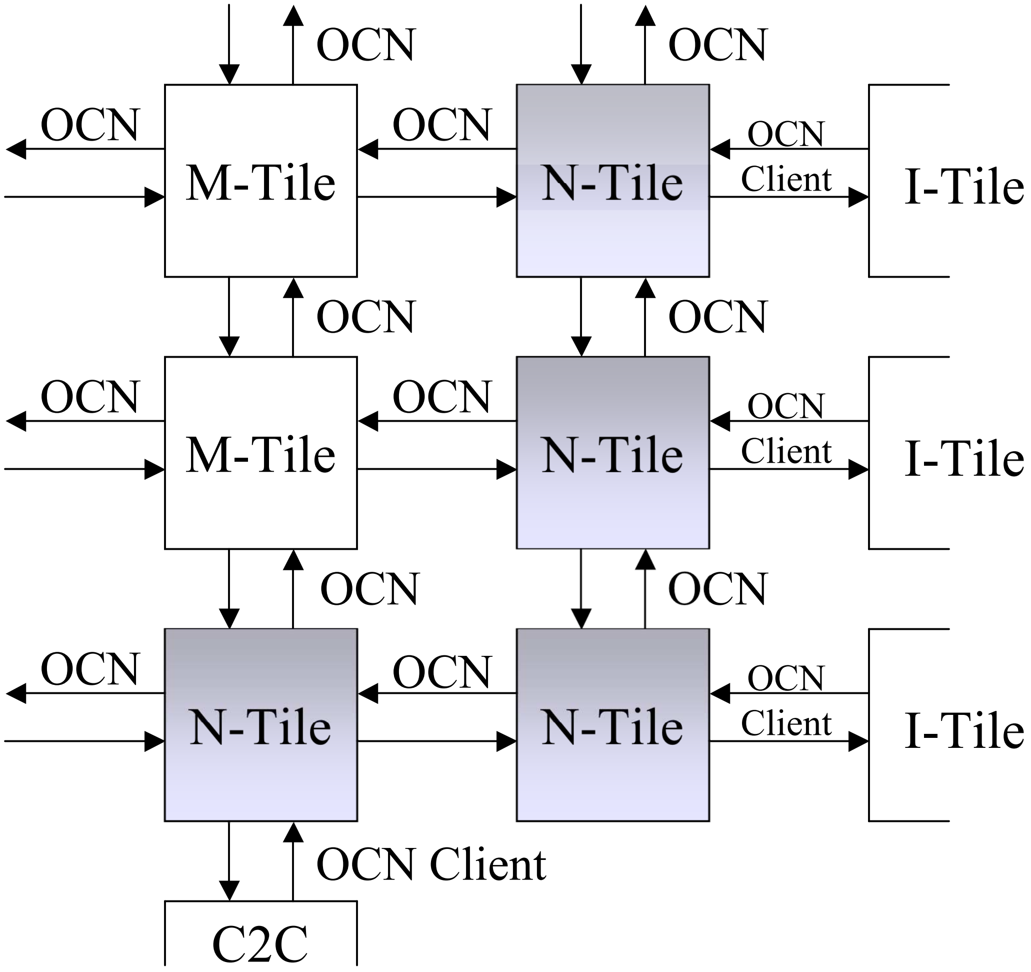
ISCA-32 Tutorial

Paul Gratz

Department of Computer Sciences
The University of Texas at Austin

OCN Location and Connections

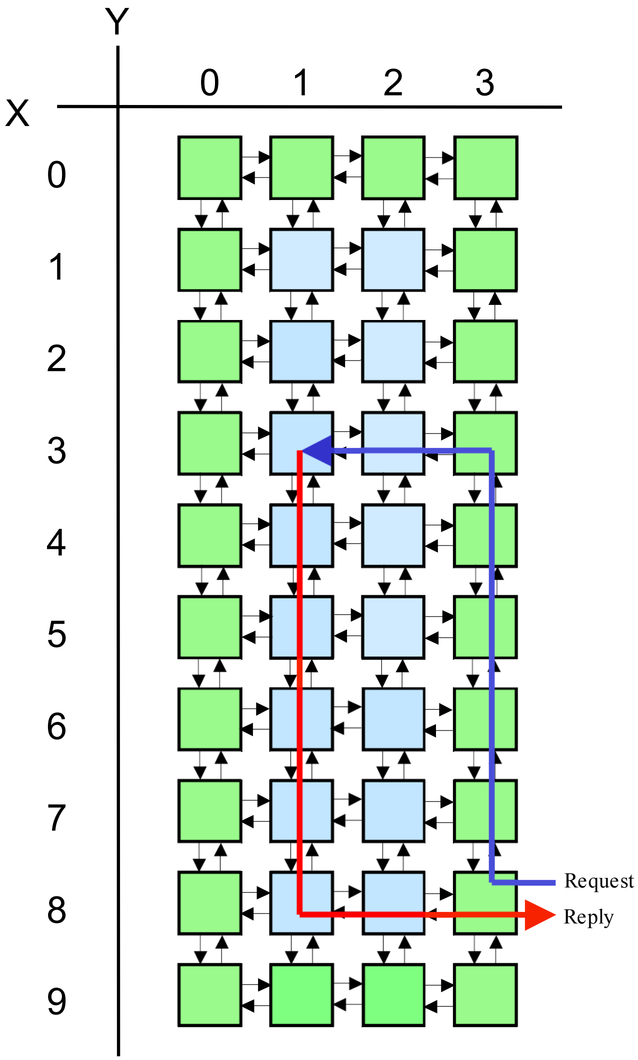




Goals and Requirements of the OCN

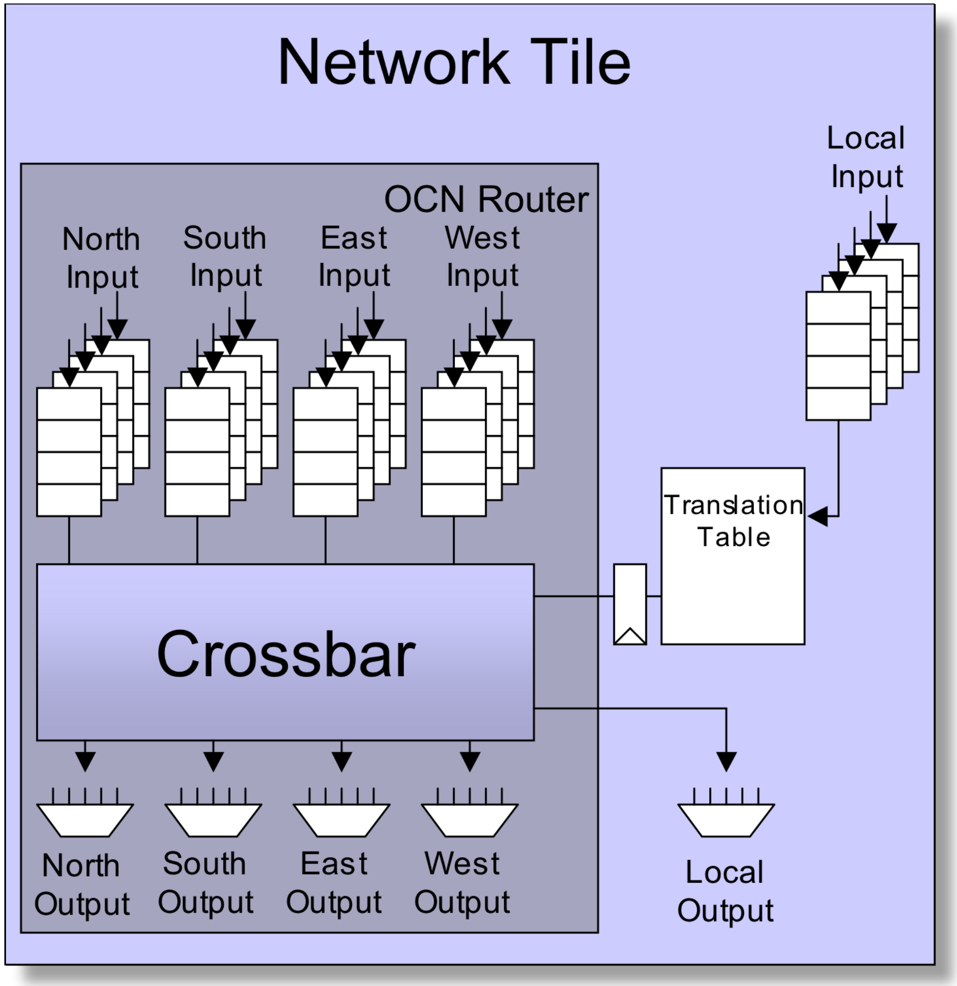
- Fabric for interconnection of processors, memory and I/O
- Handle multiple types of traffic:
  - Routing of L1 and L2 cache misses
  - Memory requests
  - Configuration and data register reads/writes by external host
  - Inter-chip communication
- OCN Network Attributes
  - Flexible mapping of resources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

OCN Protocol



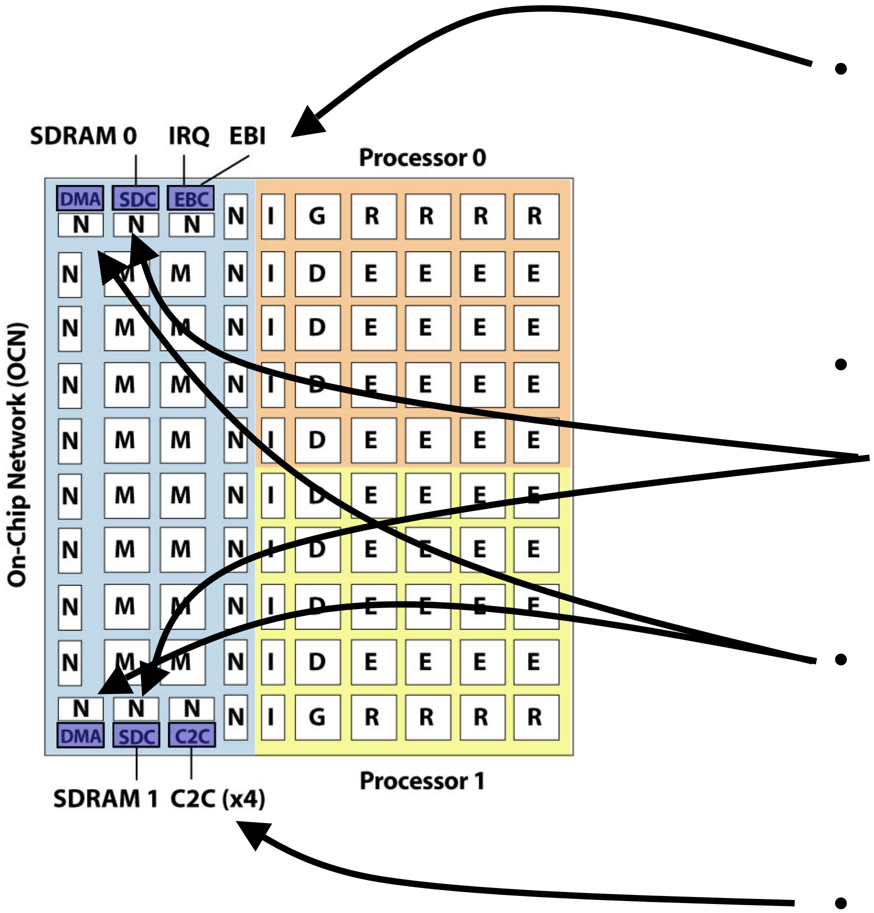
- Flexible mapping of resources
  - System address mapping tables are runtime re-configurable to allow for runtime re-mapping between L2 cache banks and scratchpad memory banks
- Deadlock Avoidance
  - Dimension order routed network (Y-X)
  - Four virtual channels used to avoid deadlock
- Performance
  - 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



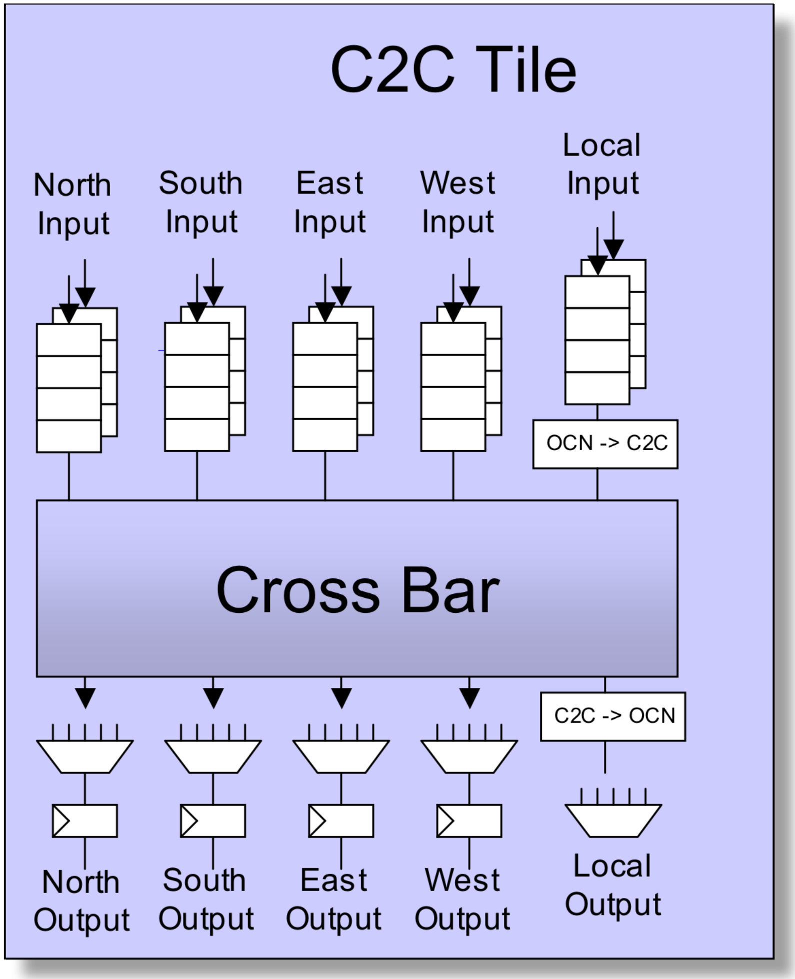
- Goals
  - Router for OCN packets
  - System address to network address translation
    - Configurable L2 network addresses
  - One cycle per hop latency
- Attributes
  - Composed of OCN router with translation tables on local input
  - Portions of the system address index into translation tables
 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

I/O Tiles



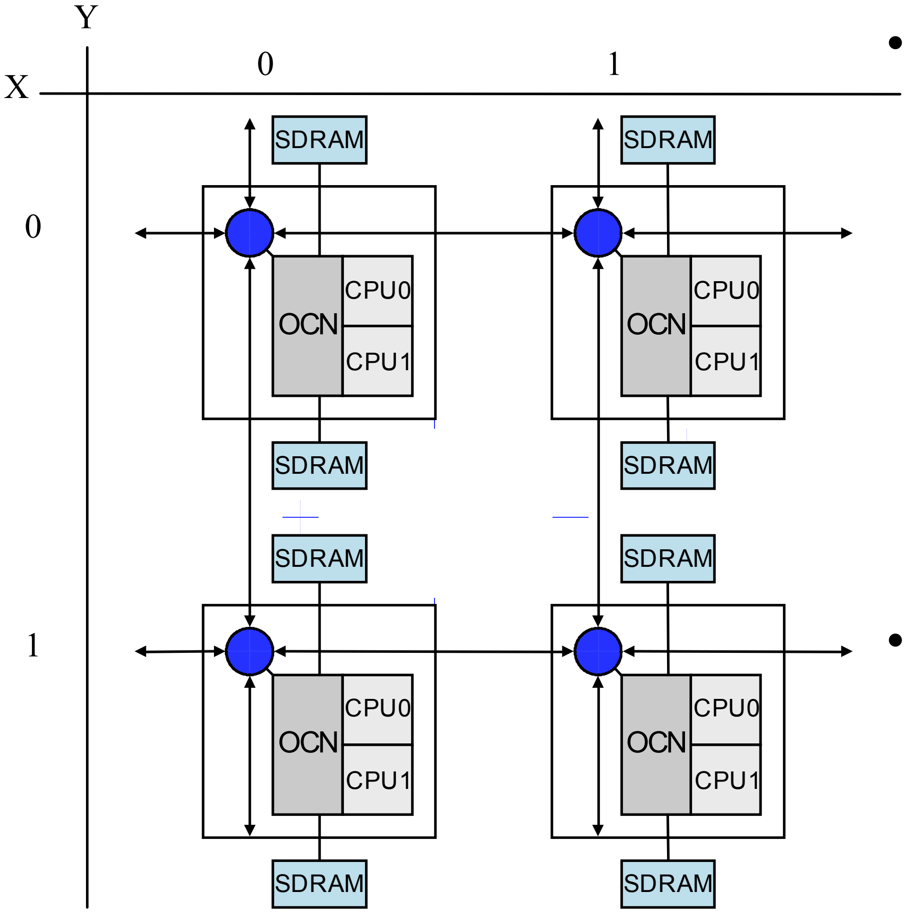
- EBC Tile
  - Provides interface between board-level PowerPC 440 GP chip and the TRIPS chip
  - Forwards interrupts to the 440 GP’s service routines
- SDC Tile
  - Contains IBM SDRAM Memory Controller
  - Error detection, swap support and address remapping added
- DMA Tile
  - Supports multi-block and 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

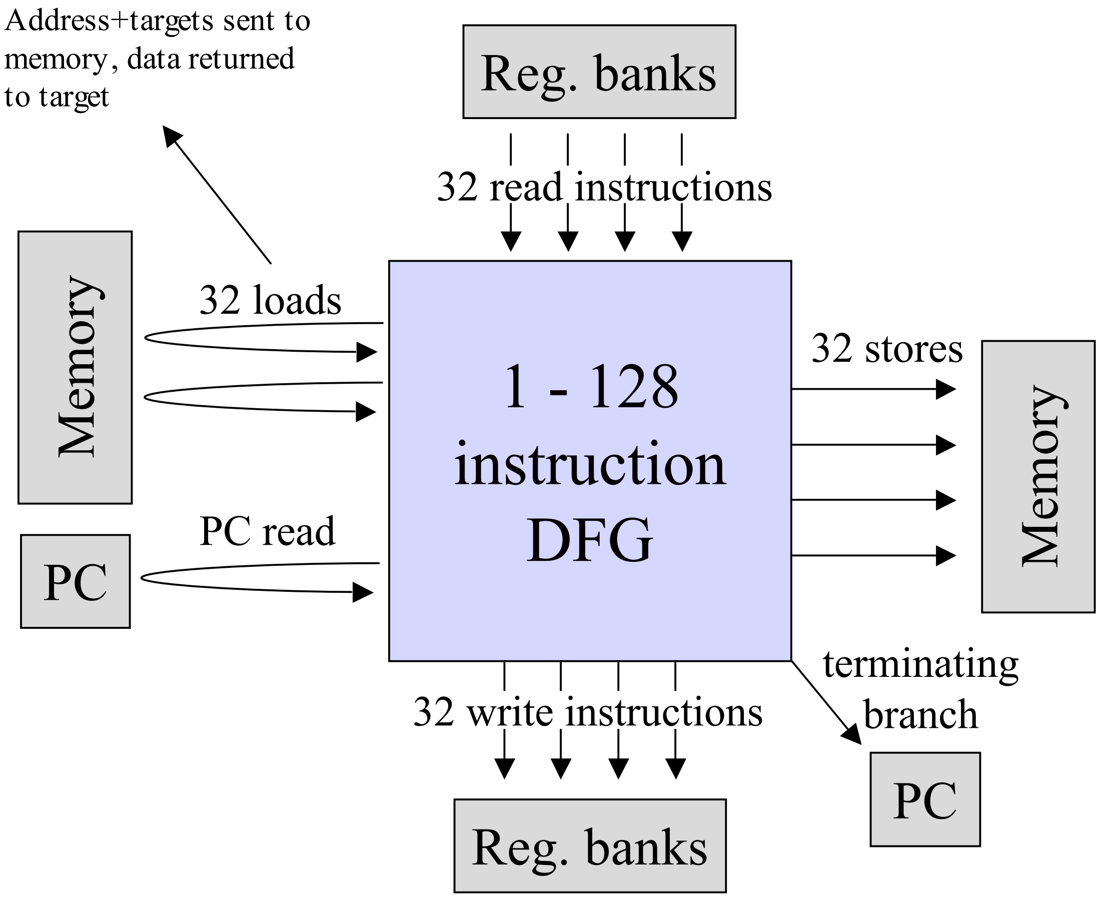
“Don’t put off to runtime what you can do at compile time.” Norm Jouppi, May 2005.

- 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



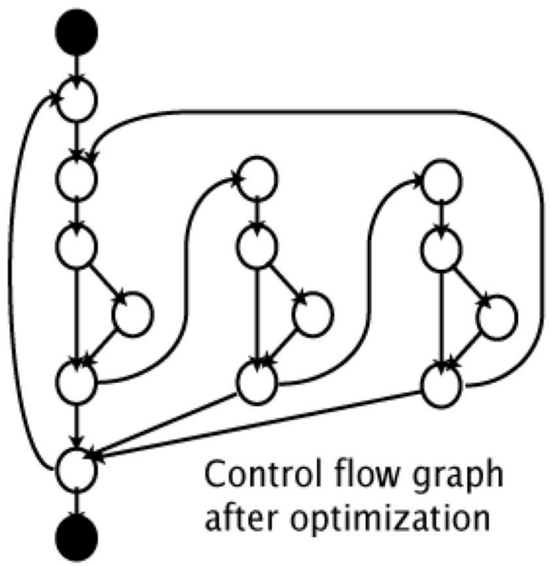
TRIPS Block Constraints

- **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

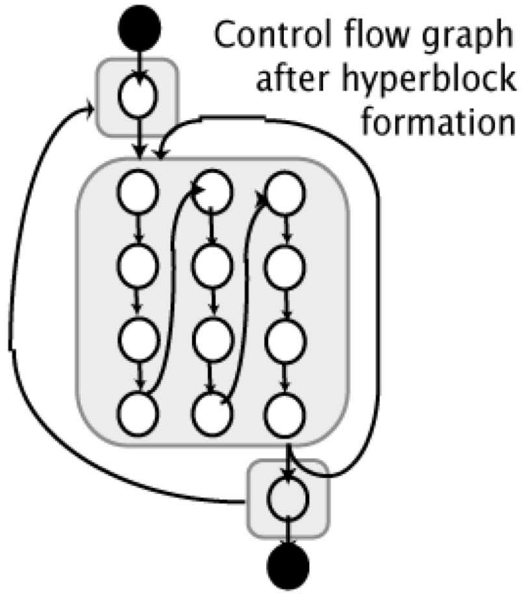
Outline

- 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



Control flow graph after optimization



Control flow graph after hyperblock formation

C/Fortran Frontend

High-level Transformations

- Inlining
- Loop Unrolling and Flattening

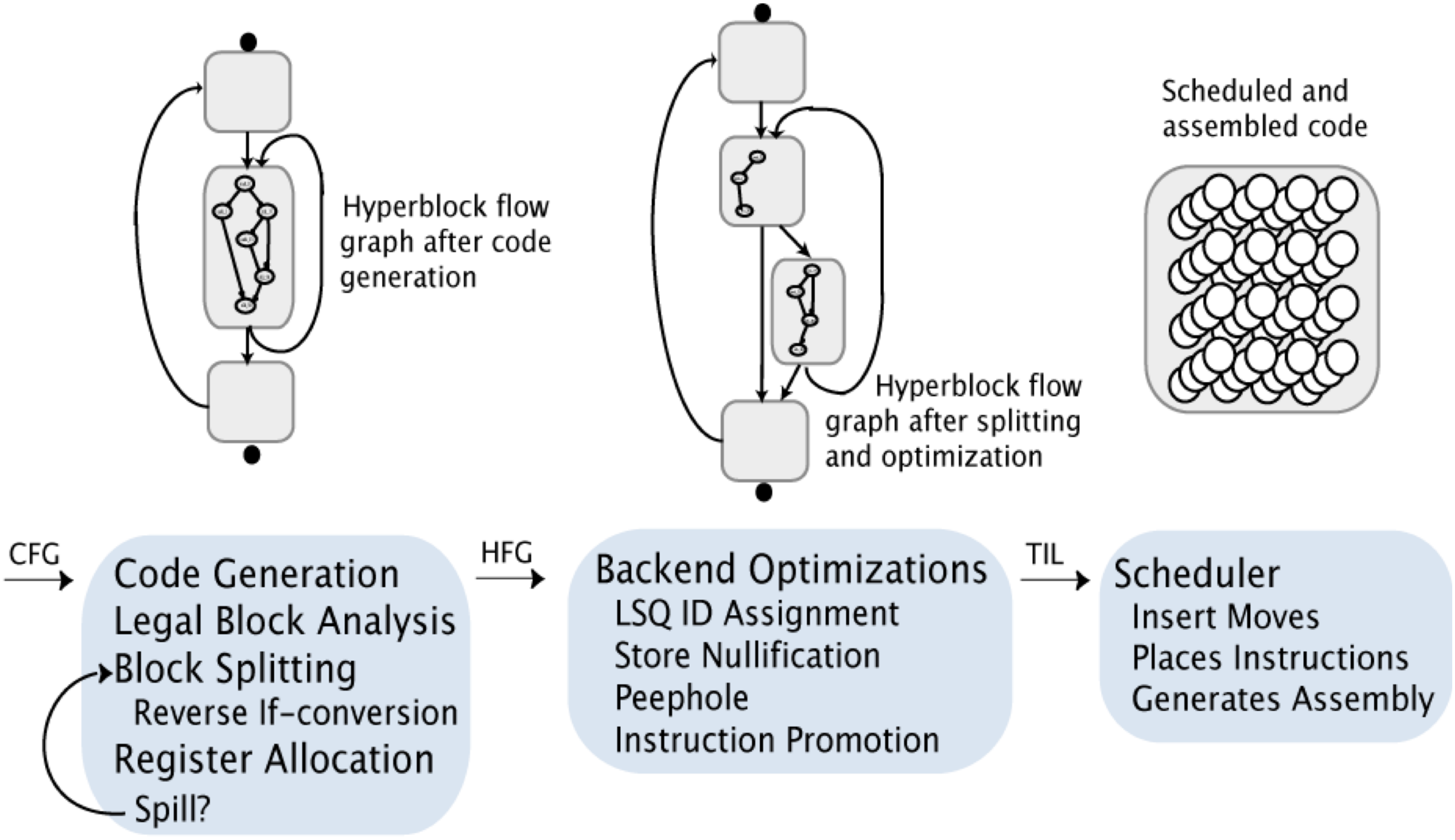
Scalar Optimizations

CFG →

Hyperblock Formation

- Region Selection
- If-conversion

Compiler Phases (2)



Hyperblock flow graph after code generation

Hyperblock flow graph after splitting and optimization

Scheduled and assembled code

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

Backend Optimizations
  LSQ ID Assignment
  Store Nullification
  Peephole
  Instruction Promotion

Scheduler
  Insert Moves
  Places Instructions
  Generates Assembly

Forming TRIPS Blocks

ISCA-32 Tutorial

Aaron Smith

Department of Computer Sciences
The University of Texas at Austin

High Quality TRIPS Blocks

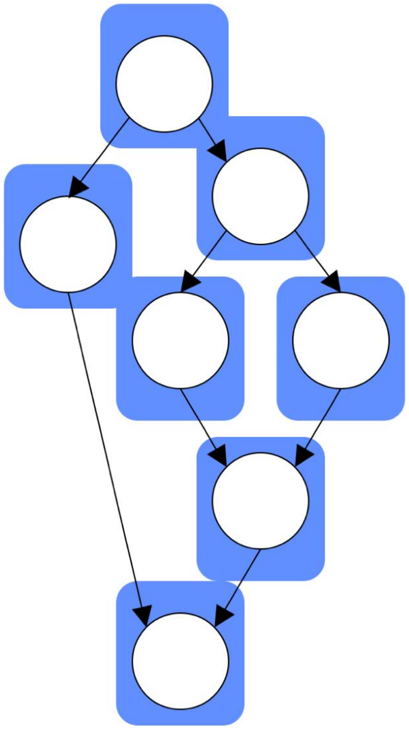
- Full
- High ratio of committed to total instructions
  - Dependence height is not an issue
- Minimize branch misprediction
- Meet correctness constraints

TRIPS Block Constraints

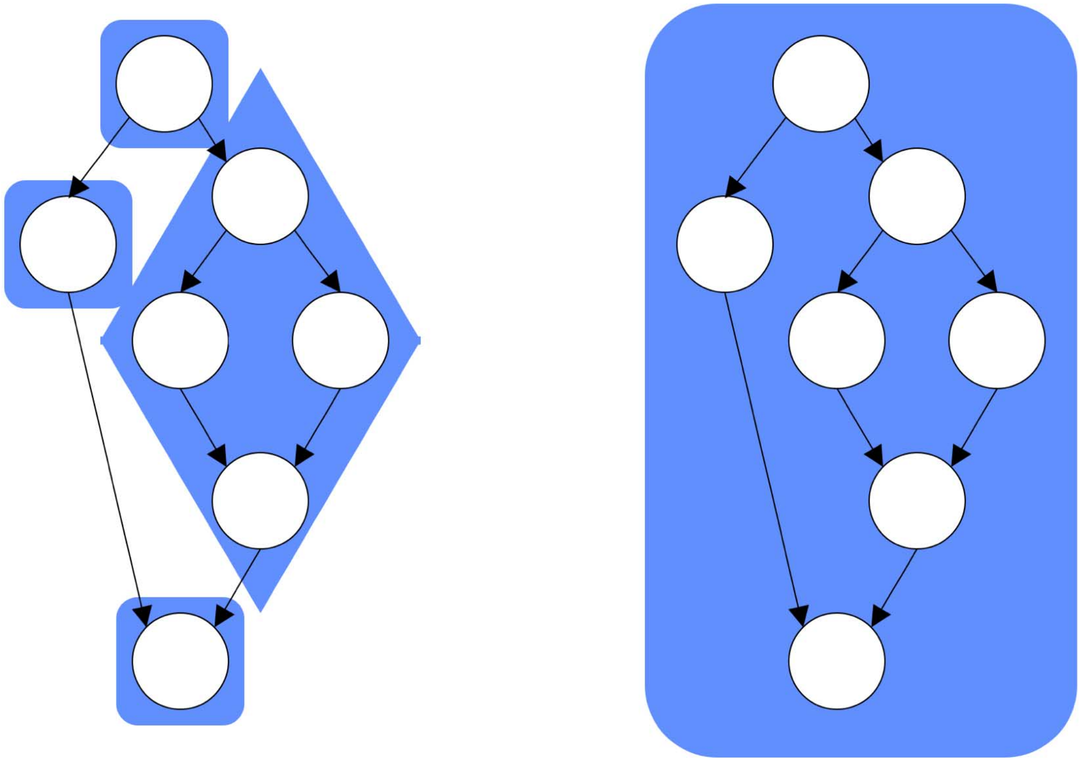
- **Fixed size**: at most, 128 instructions
  - Easy to satisfy by the compiler, but….
  - Good performance requires blocks full of useful instructions
- Registers: 8 reads and writes to each of 4 banks (plus 128 instructions)
- Block Termination
- Fixed output: at most, 32 load or store queue IDs

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



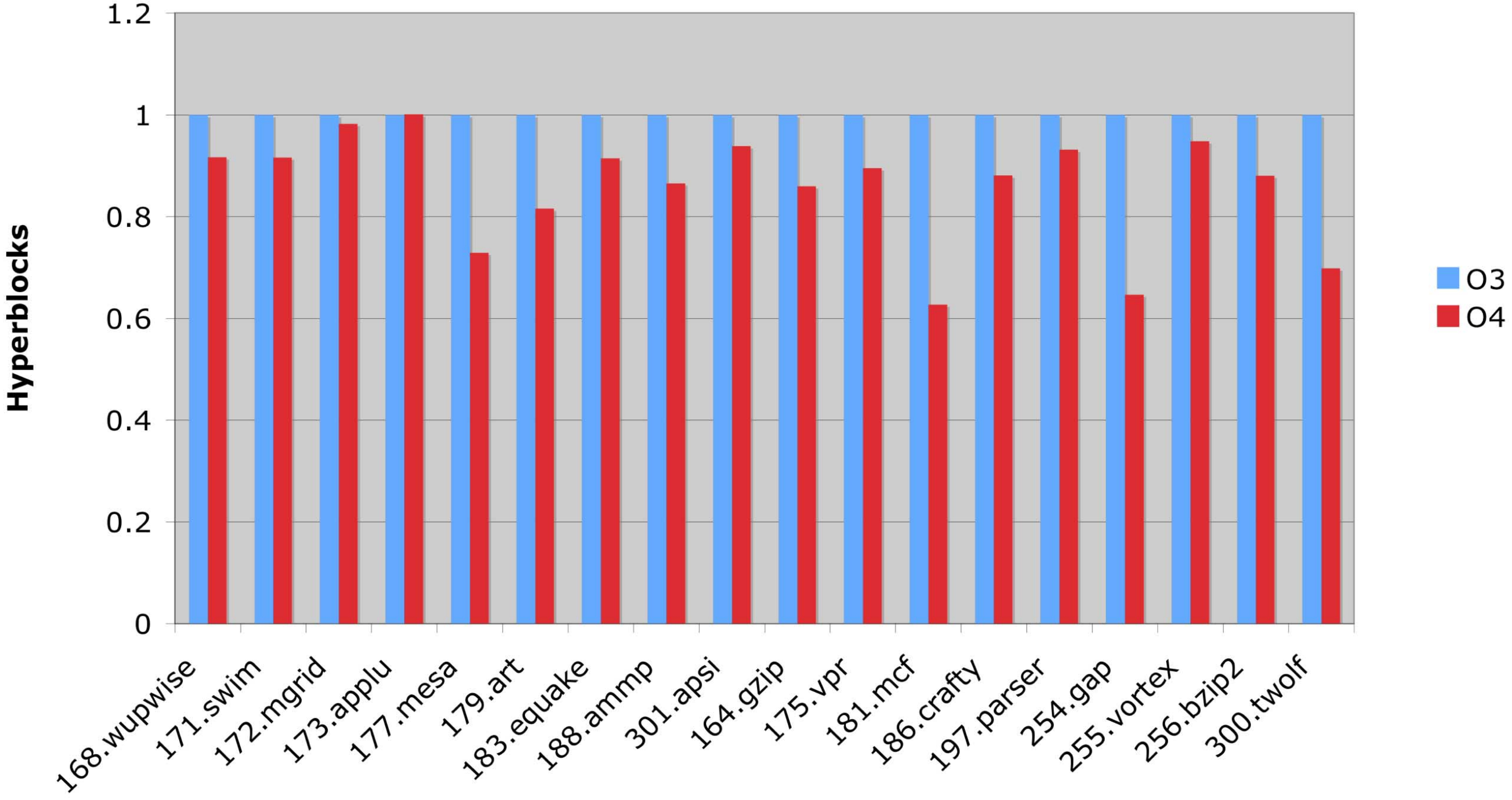
Two possible hammock hyperblocks for this graph

Static TRIPS Block Instruction Counts

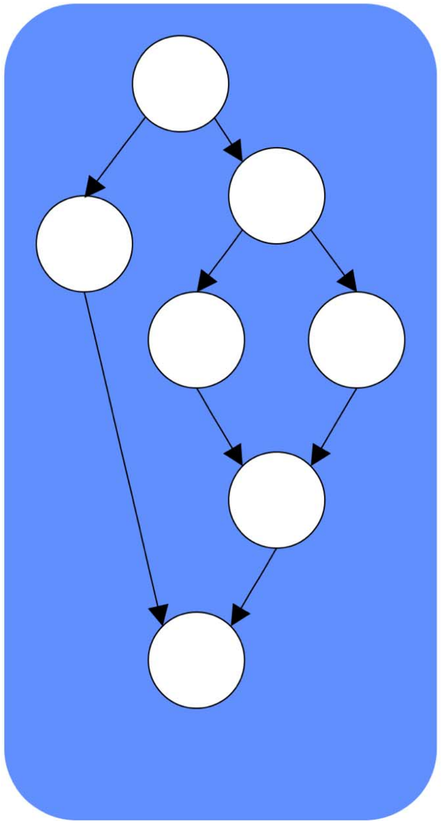
	Static Averages					
	03			04		
	min	**max**	**mean**	**min**	**max**	**mean**
SPEC2000	**7**	**42**	**16**	**8**	**42**	**18**
EEMBC	**7**	**31**	**16**	**9**	**67**	**22**

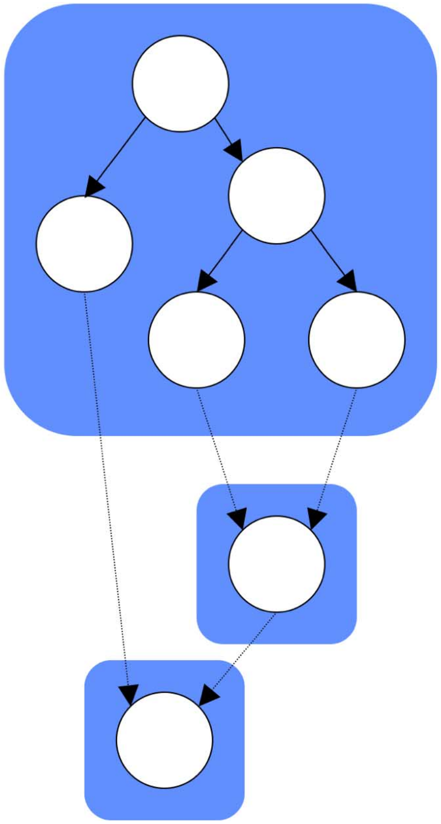
SPEC Total TRIPS Blocks

- **15% average reduction in number of TRIPS blocks between O3 and O4**
- **Results missing sixtrack, perl, and gcc**



Enforcing Block Size





Initial Hammock Hyperblock **After Reverse If-Conversion**

Too Big?

1. **Unpredicated**

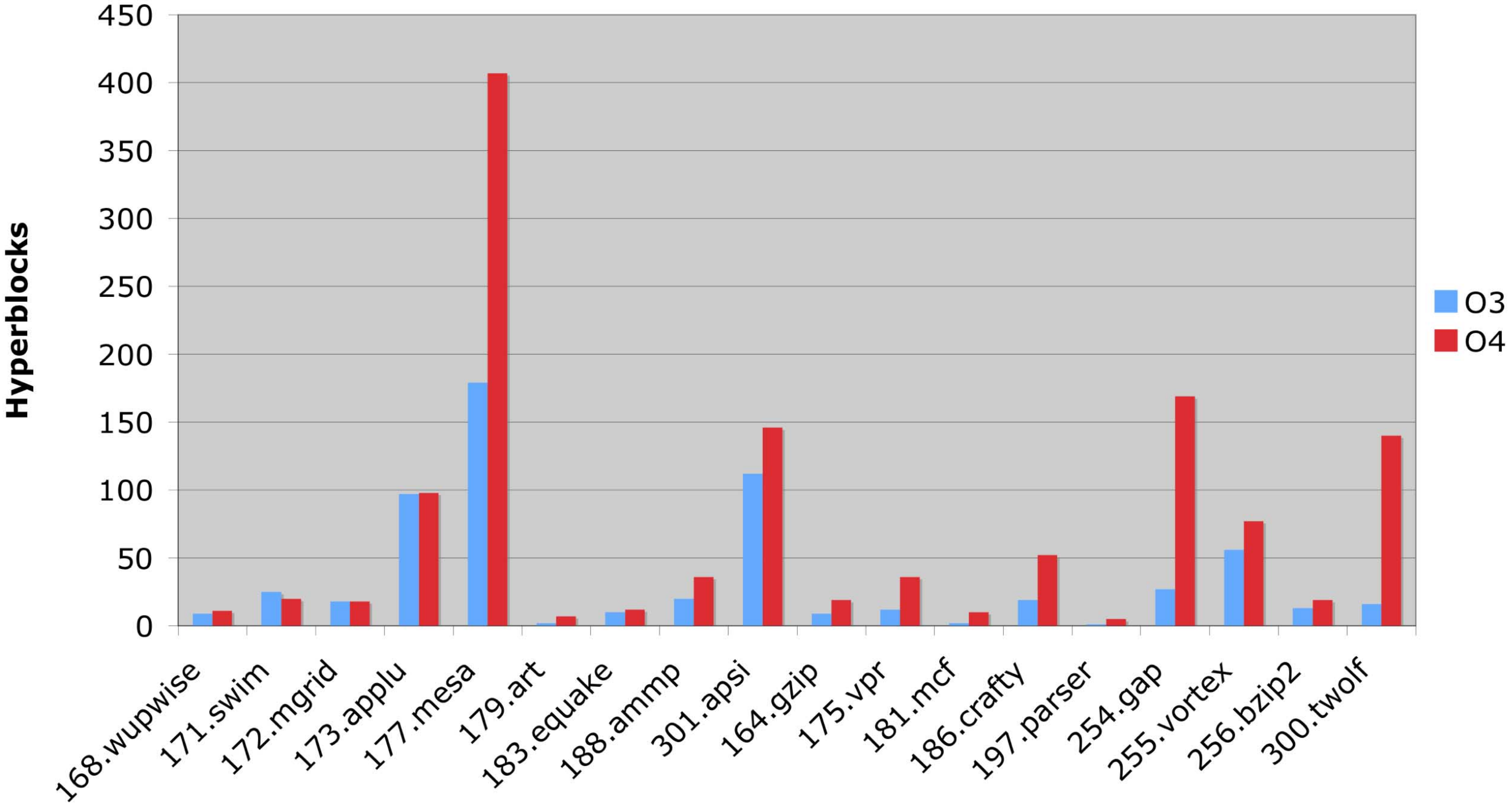
   - cut

2. **Predicated**

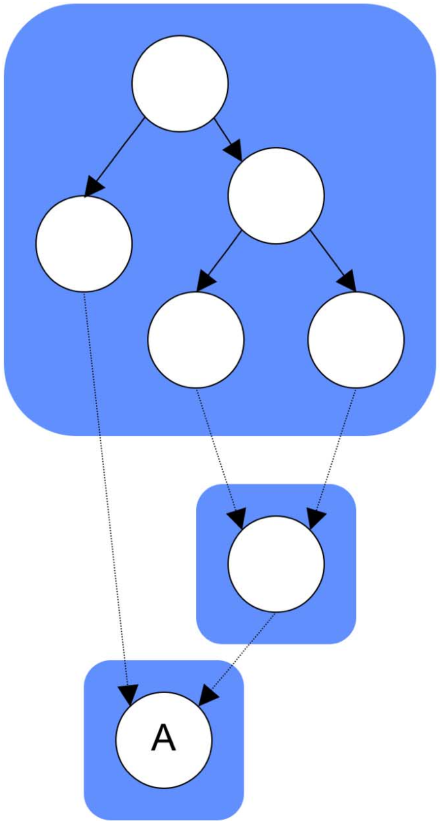
   - Reverse if-conversion

SPEC > 128 Instructions

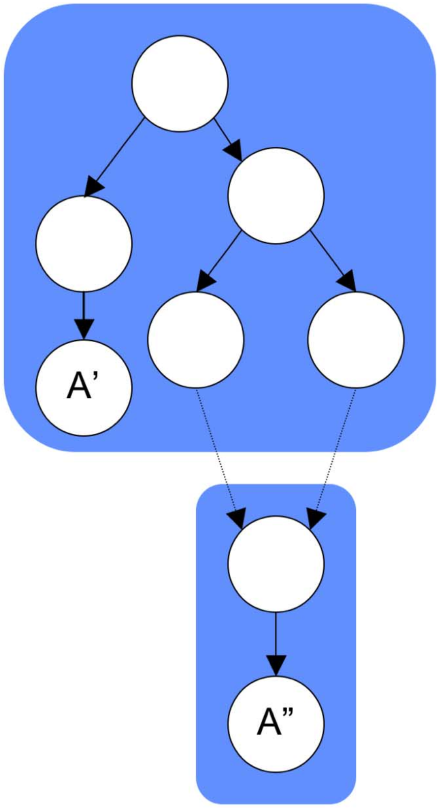
- **At O4 an average of 0.9% TRIPS blocks > 128 insts. (max 6.6%)**



In Progress: Tail Duplication



After Reverse If-Conversion

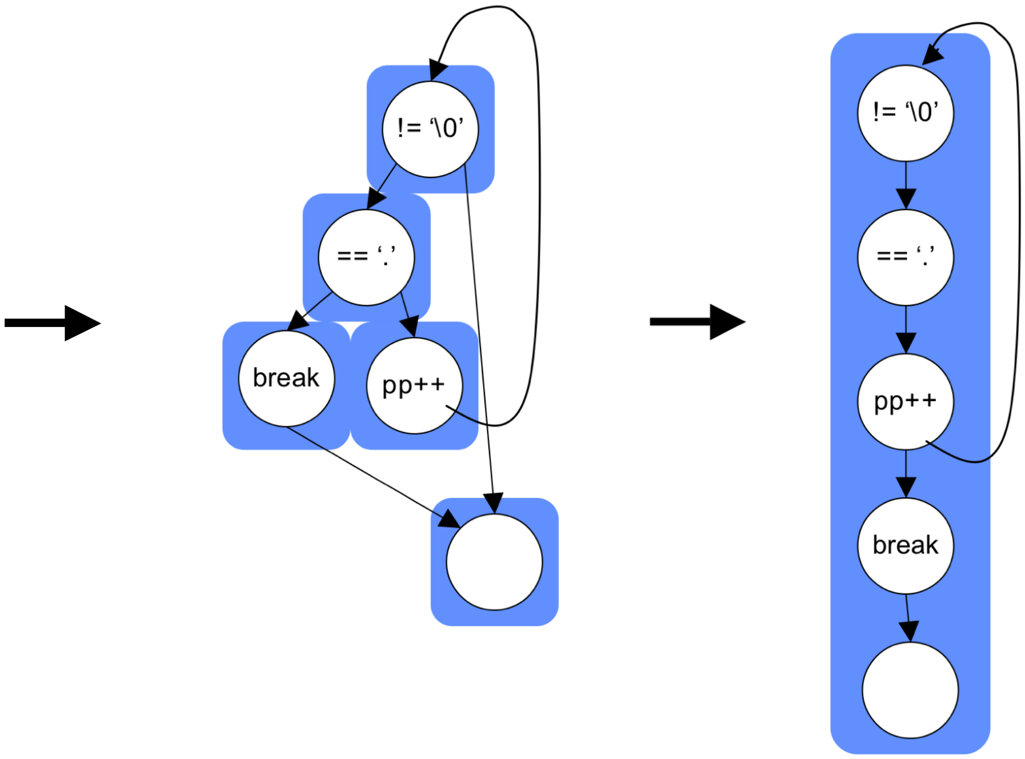


After Tail Duplication

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

In Progress: More Aggressive Hyperblocks

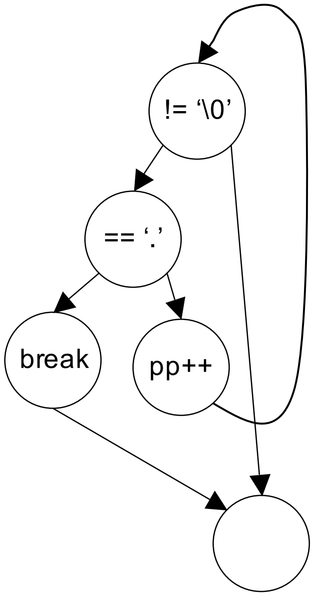
```
while (*pp != '\0')
{
   if (*pp == '.') {
       break;
   }
   pp++;
}
```

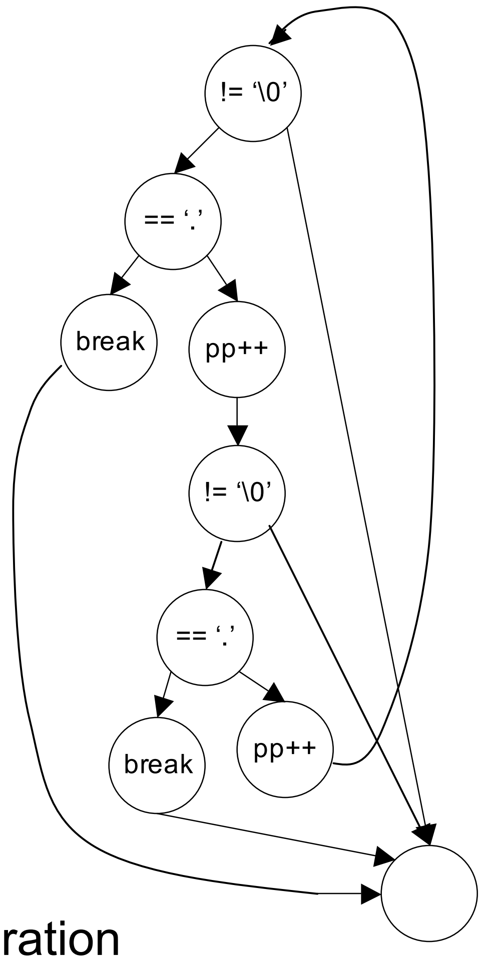


- Add support for complex control flow (i.e. breaks/continues)
- An example: the number of TRIPS blocks in ammp_1 microbenchmark
    - 38 / 27 / 12 with O3 / 04 (current) / new

While Loop Unrolling

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



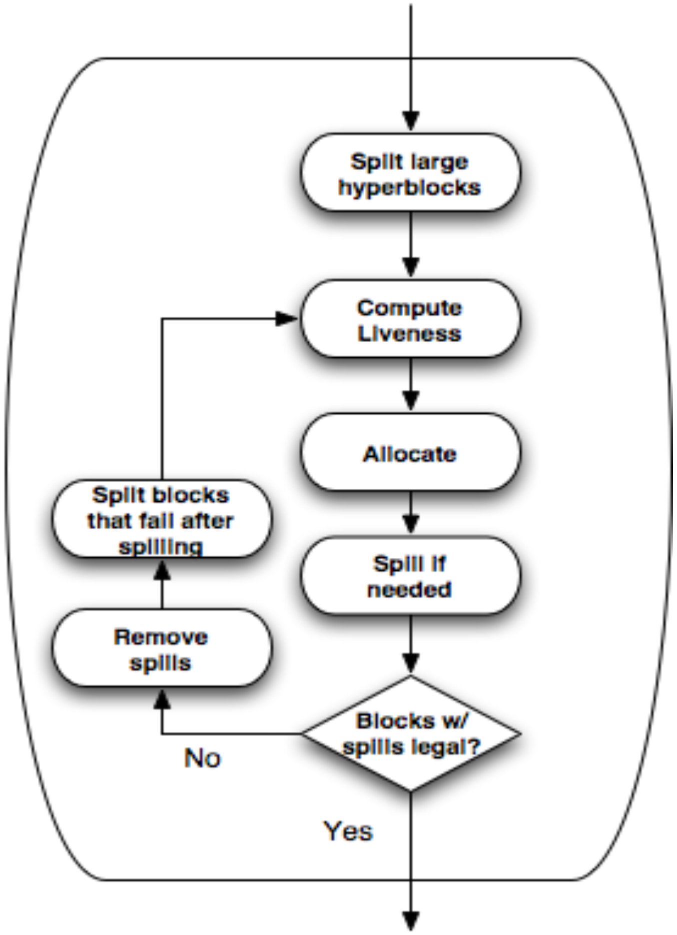


- Supported on architectures with predication
- Implemented, but needs improved hyperblock generation

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

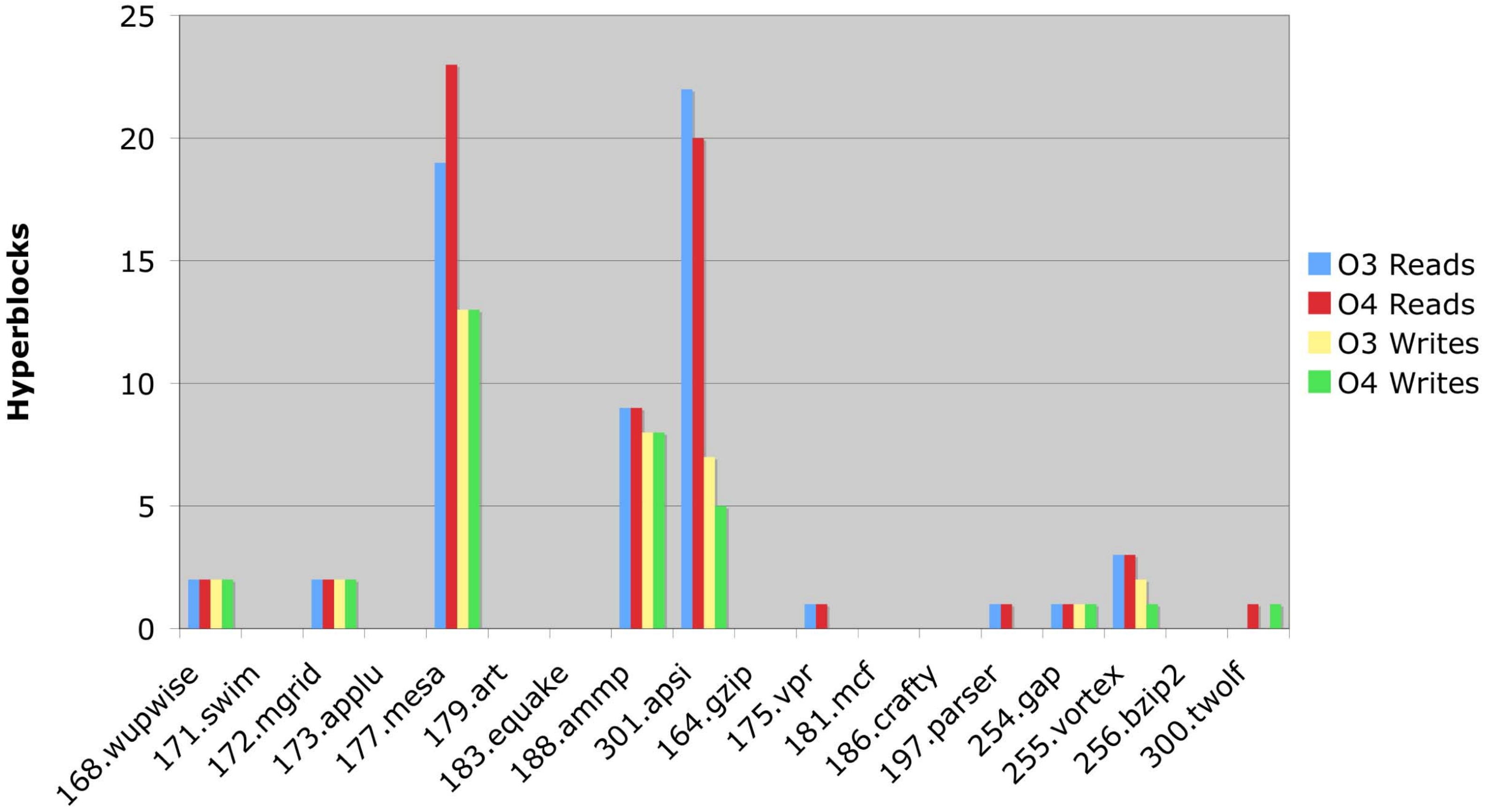
Linear Scan Register Allocation



- 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

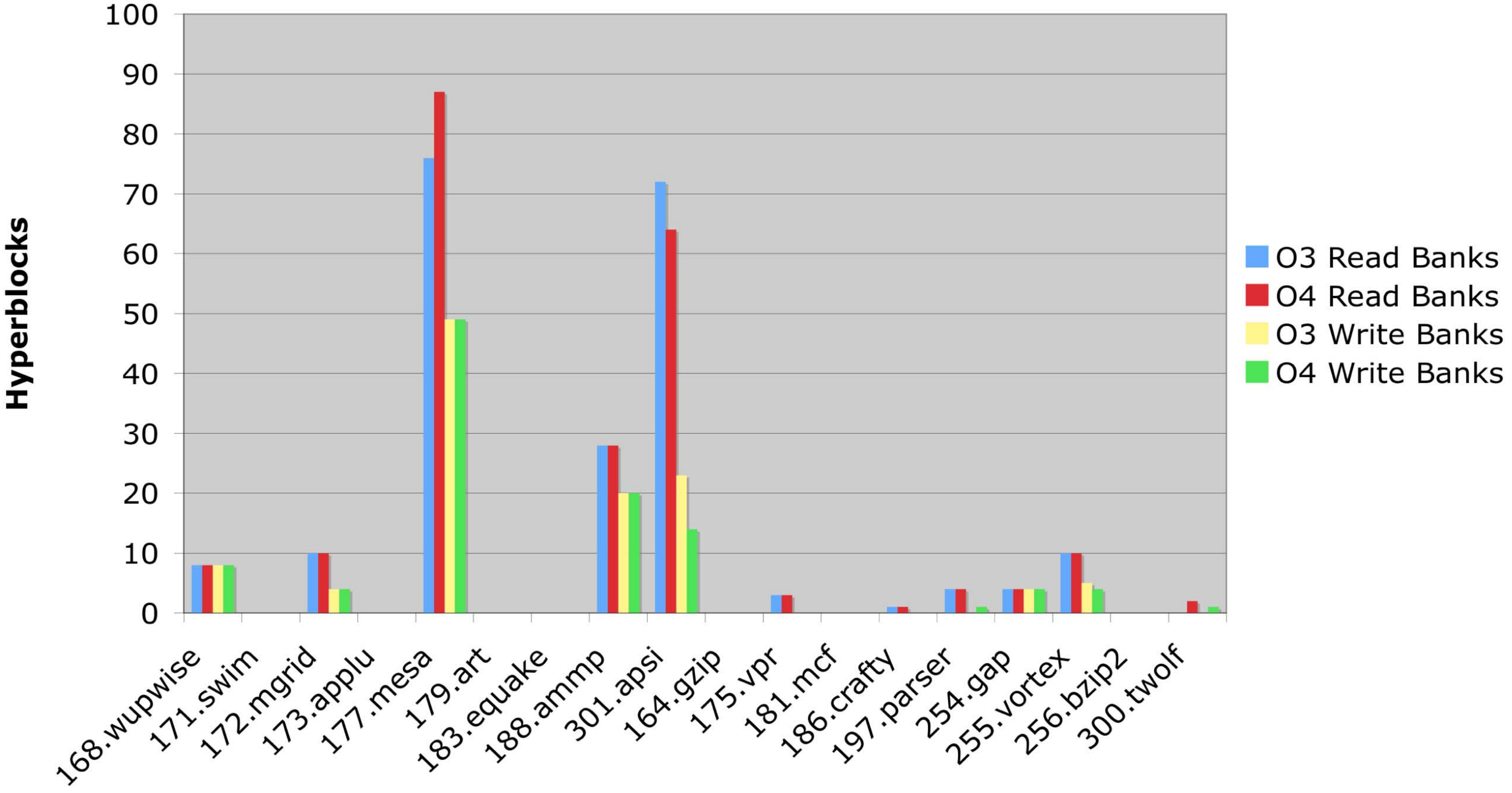
SPEC > 32 (8x4 banks) Registers

- **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**



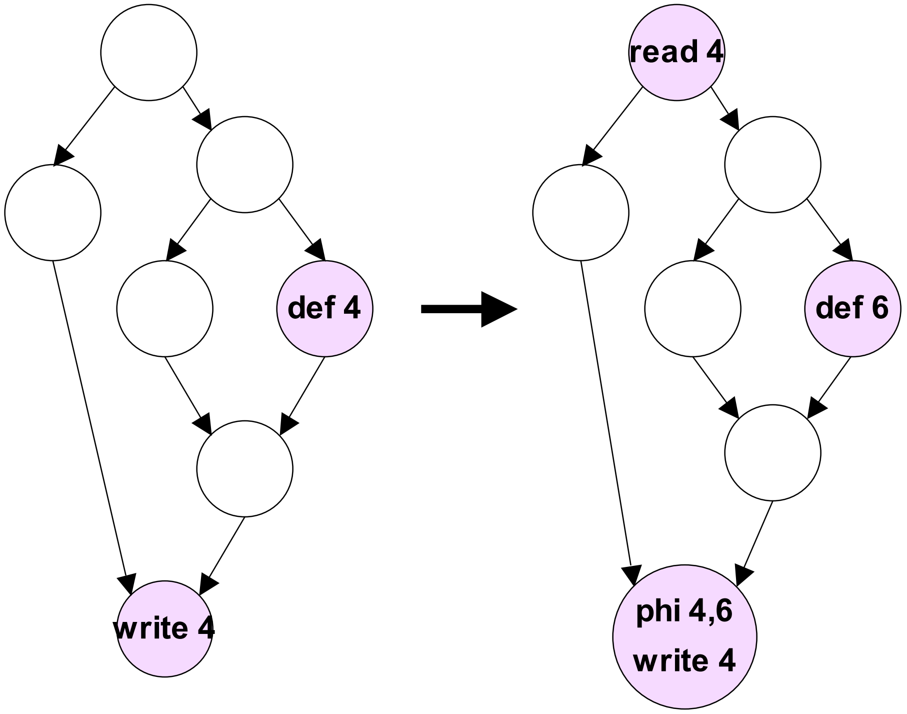
TRIPS Block Constraints

- 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

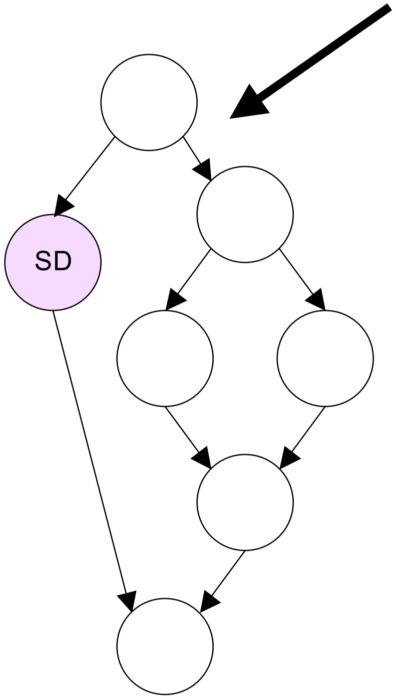


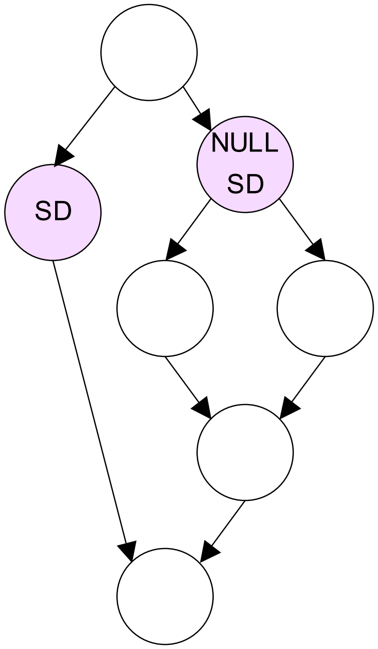
Before SSA **After SSA**

- 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

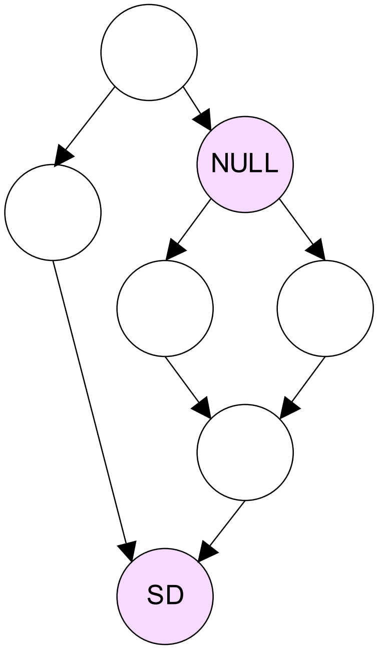
Store Nullification

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





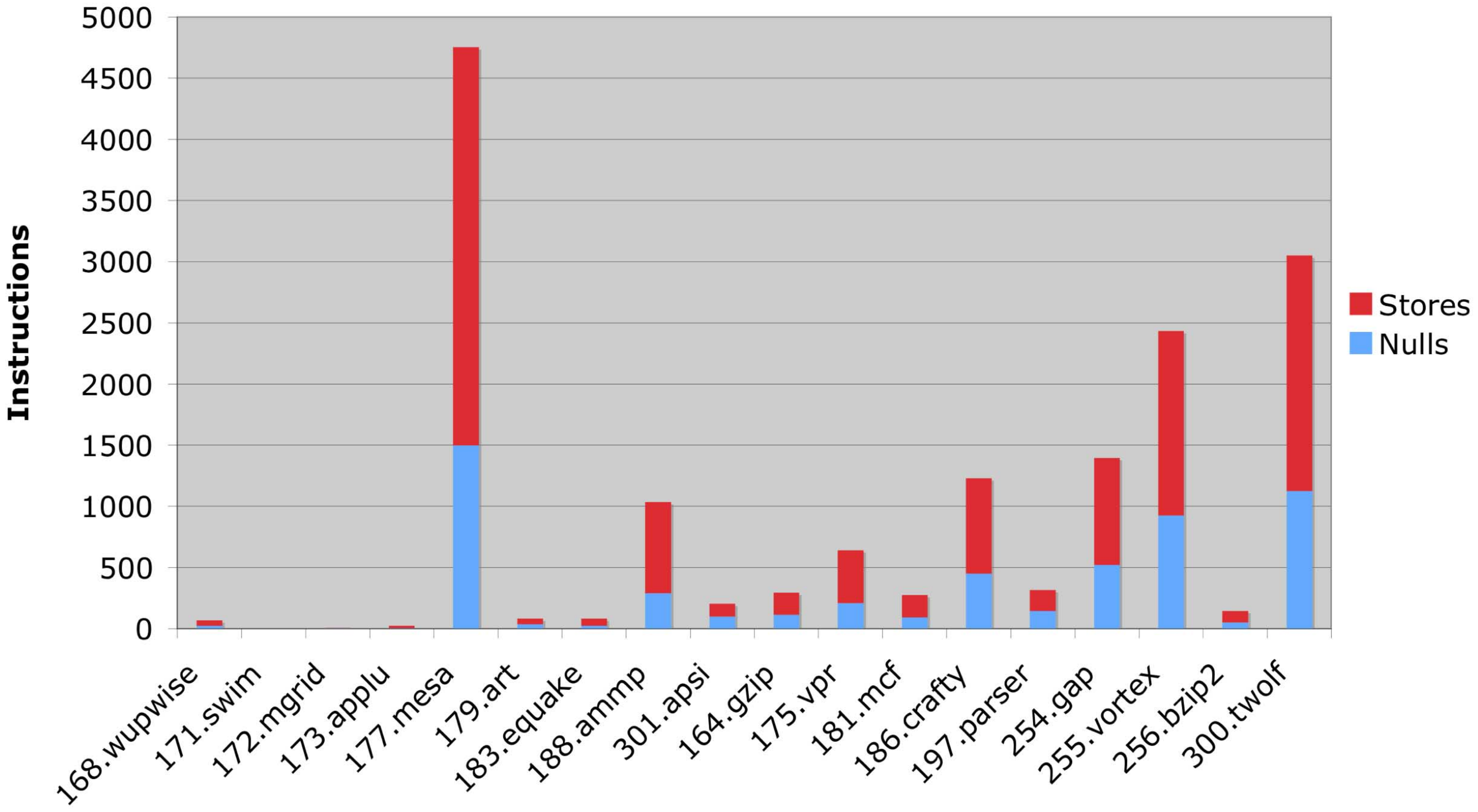
- Dummy stores
  - Wastes space
  - Easier to schedule?



- Code motion
  - Better resource utilization

SPEC Store Nullification

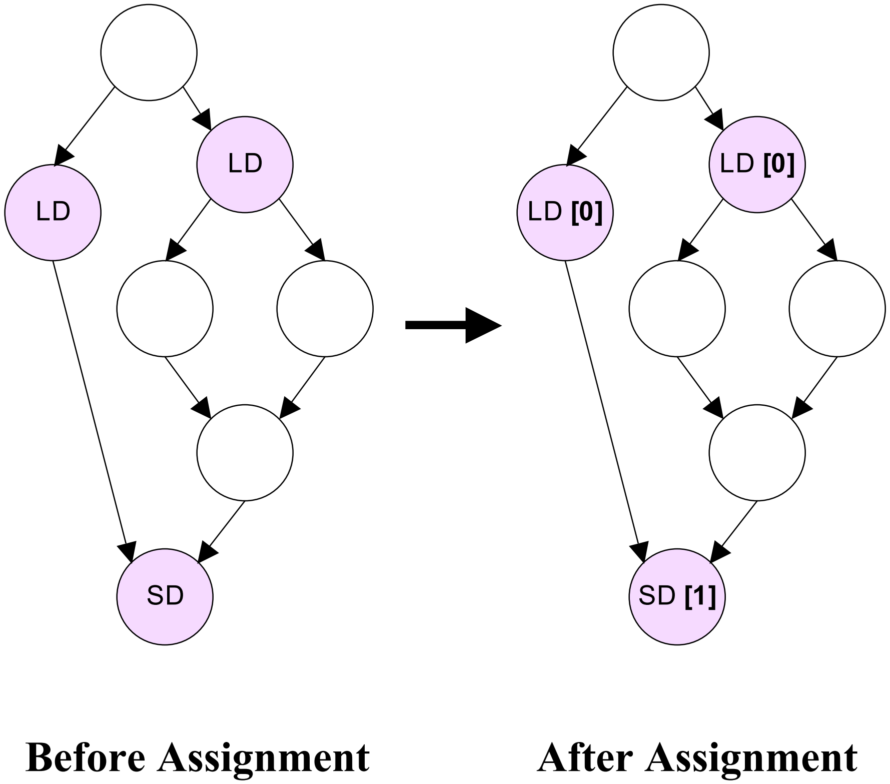
- **Nullification is 1.2% of instructions for 177.mesa (0.8% dummy stores)**
- **Is it significant?**



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

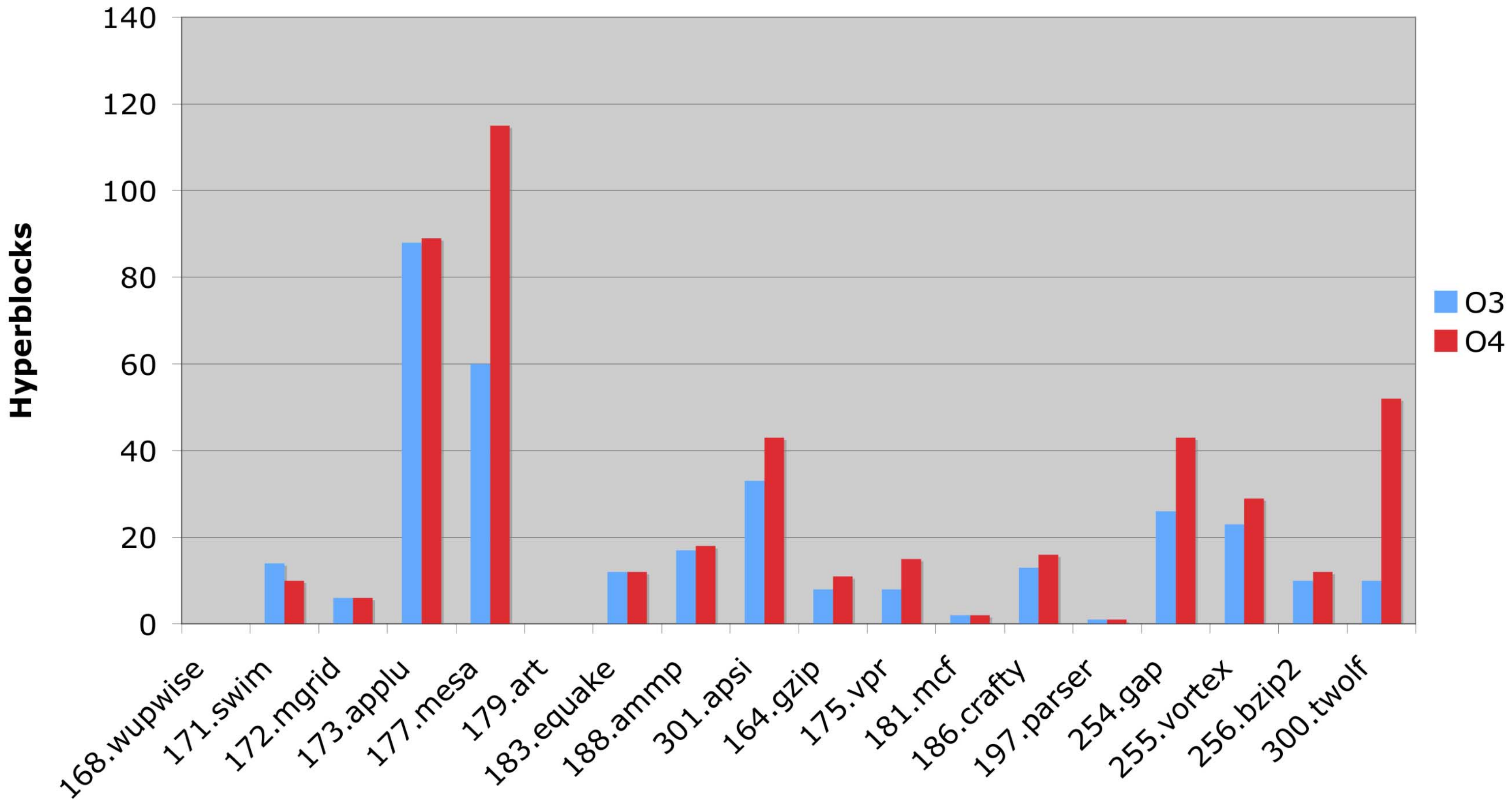
Load/Store ID Assignment



- 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

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

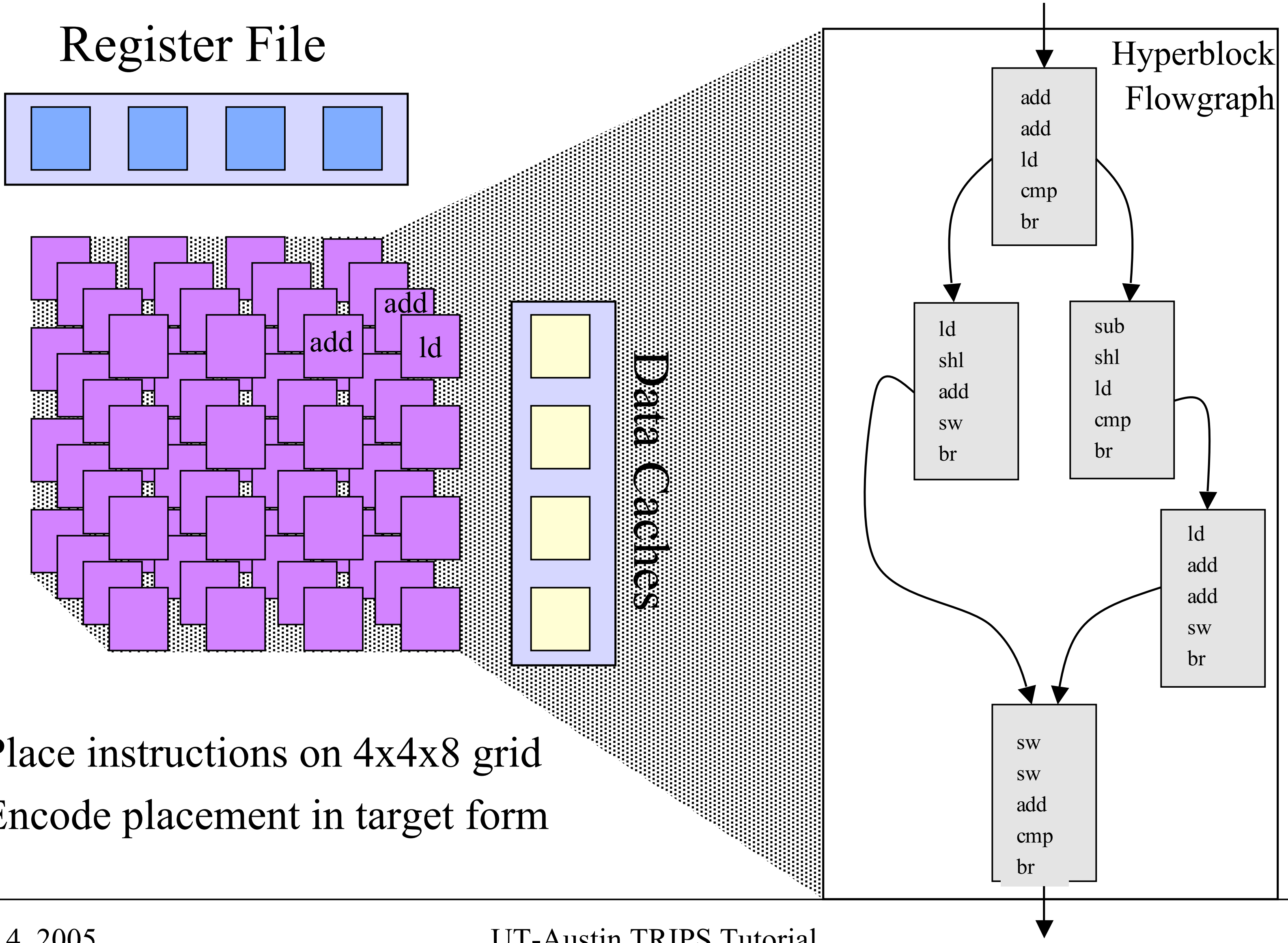
Outline

- 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*

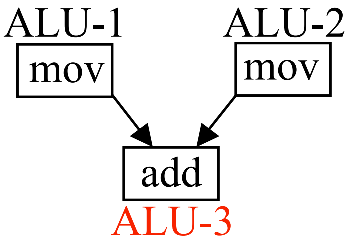
- VLIW represents one extreme
  - Static Placement and Static Issue (SPSI)
  - + Static Placement works well for partitioned architectures
  - – Static Issue causes problems with unknown latencies

- 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	EDGE
i1: movi r1, #10	ALU-1: movi #10, ALU-3
i2: movi r2, #20	ALU-2: movi #20, ALU-3
i3: add r3, r2, r1	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*

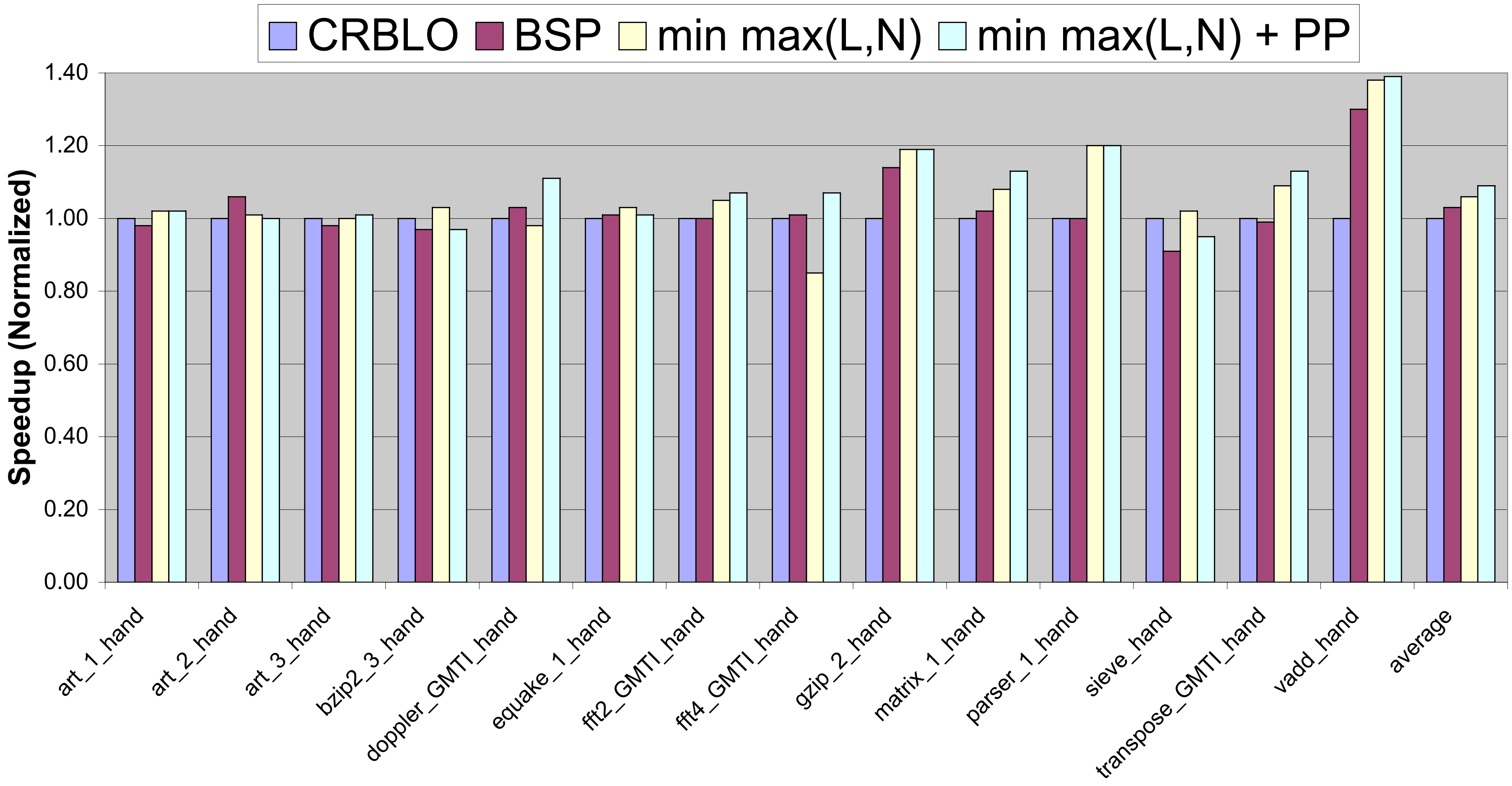
Scheduling Algorithms (1 & 2)

- 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

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

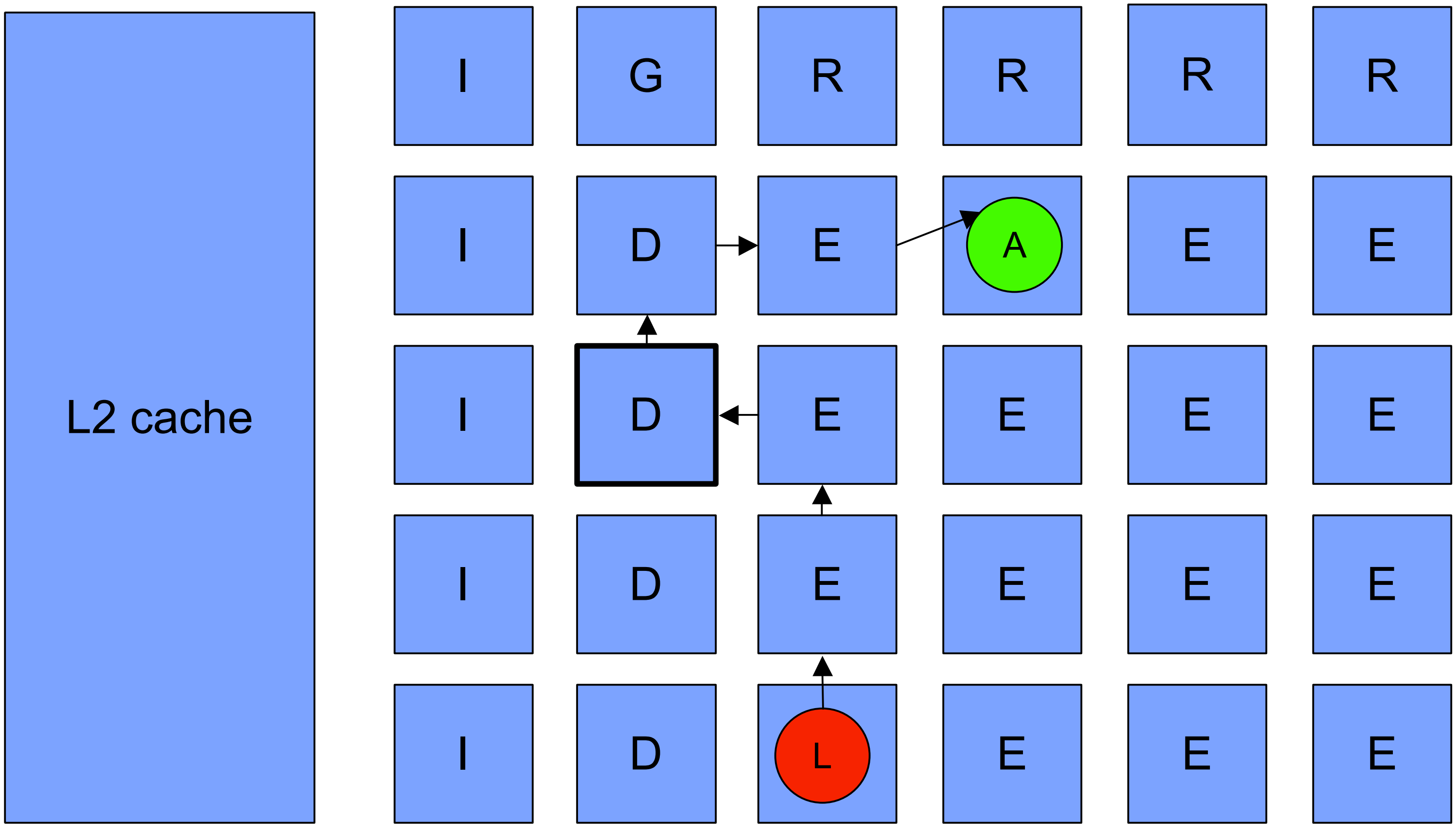


Spatial L1 Bank D-Cache Load/Store Alignment

- Problem
  - Load/store mapping to data L1 cache bank unknown to compiler
- Example

```
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

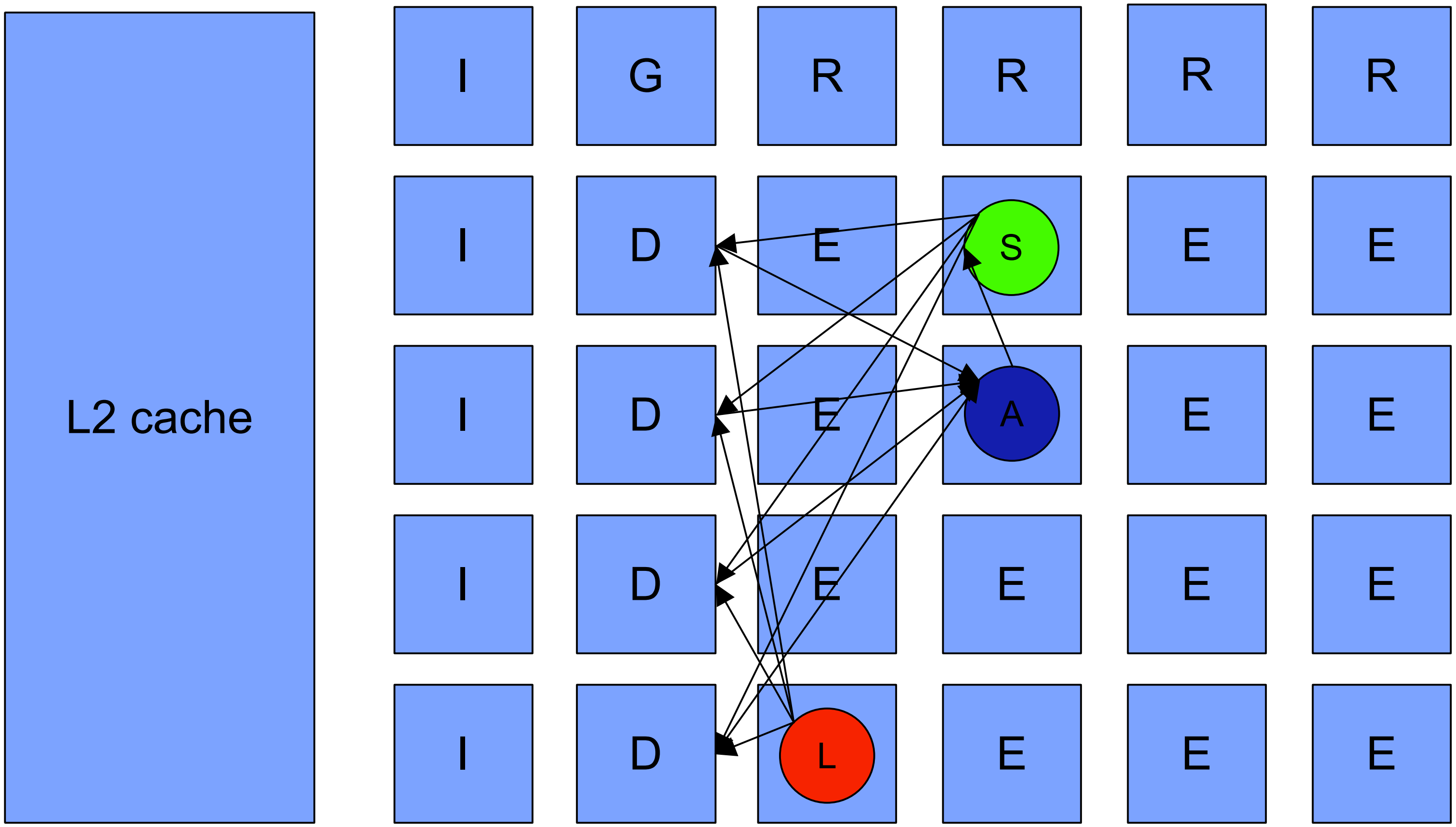
I D E A E E

I D E E E E

I D E E E E

I D L E E E

Spatial L1 D-cache Communication

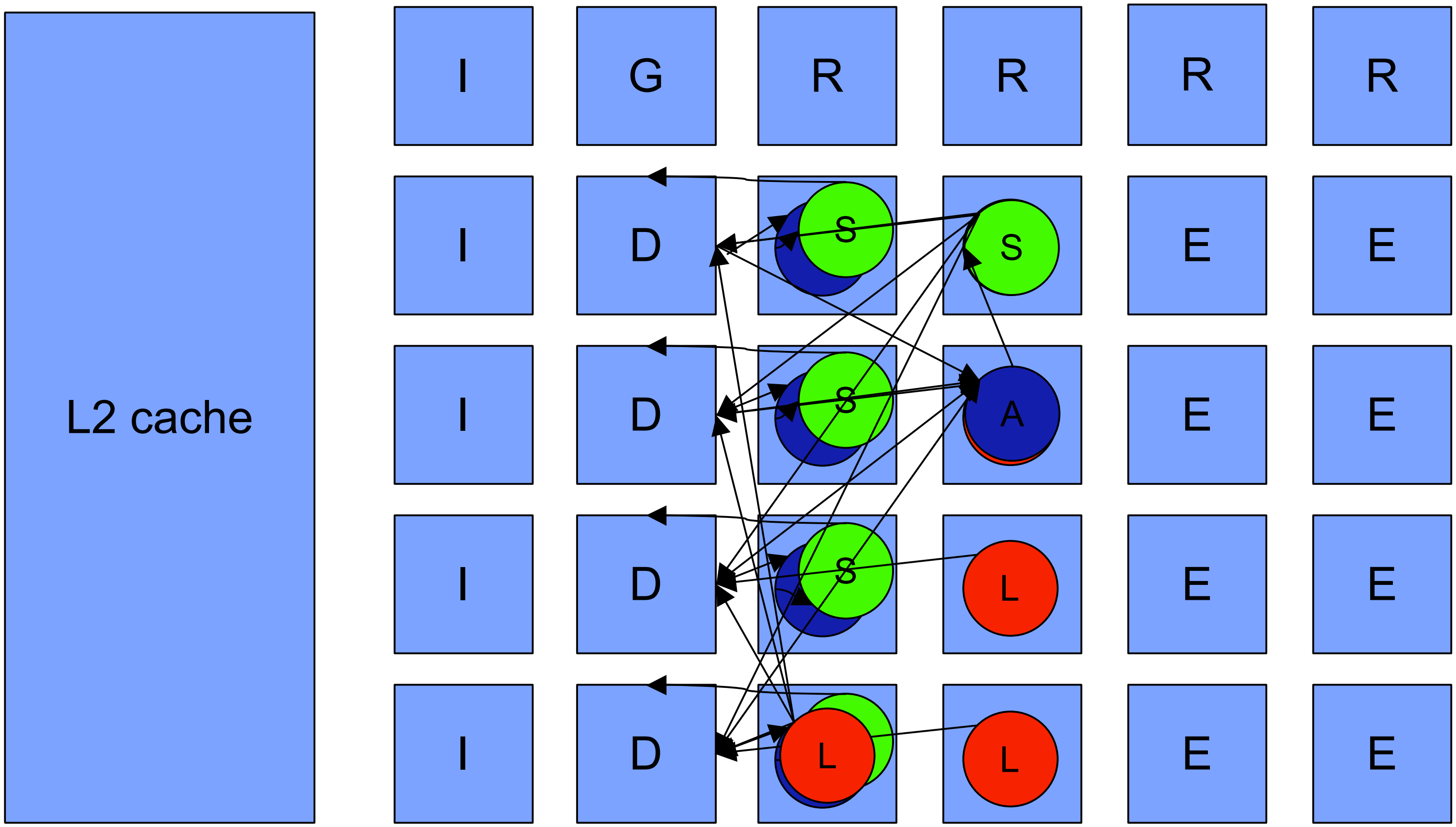


Spatial Load/Store Alignment

- 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

```
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
    C[i+16] += (A[i+16] + B[i+16]); // Bank 2
    C[i+24] += (A[i+24] + B[i+24]); // Bank 3
  }
}
```

Spatial L1 D-cache Communication



Unrolling for Larger Hyperblocks

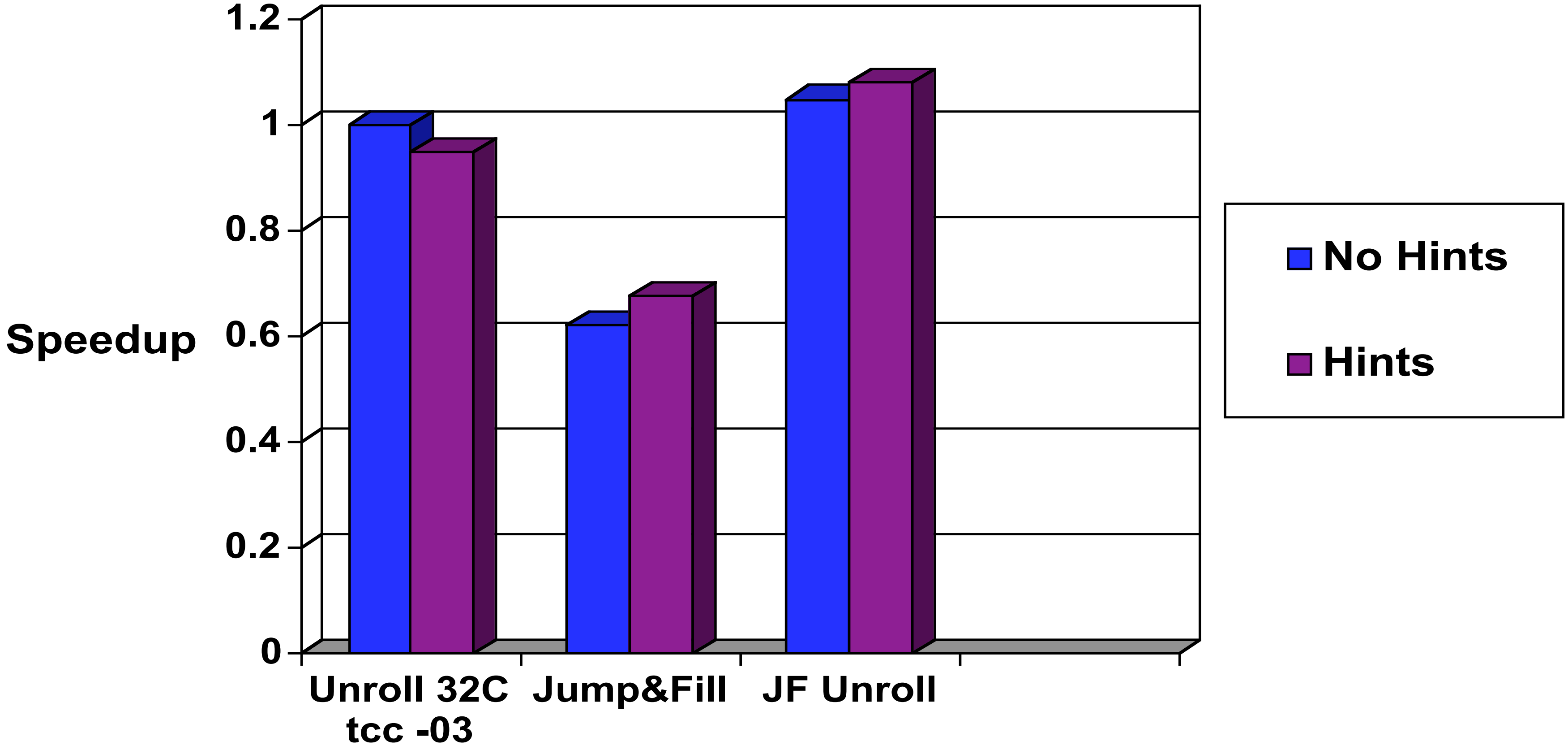
- **Vector Add Unrolled Continuously (tcc 03)**

```
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
       . . .
  C[i+31] += (A[i+31] + B[i+31]); // Bank 3
}
```

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

```
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+16] += (A[i+16] + B[i+16]); // Bank 2
  C[i+24] += (A[i+24] + B[i+24]); // Bank 3
  C[i+ 1] += (A[i+ 1] + B[i+ 1]); // Bank 0
         . . .
  C[i+31] += (A[i+31] + B[i+31]); // Bank 3
}
```

Spatial Load/Store Alignment Results

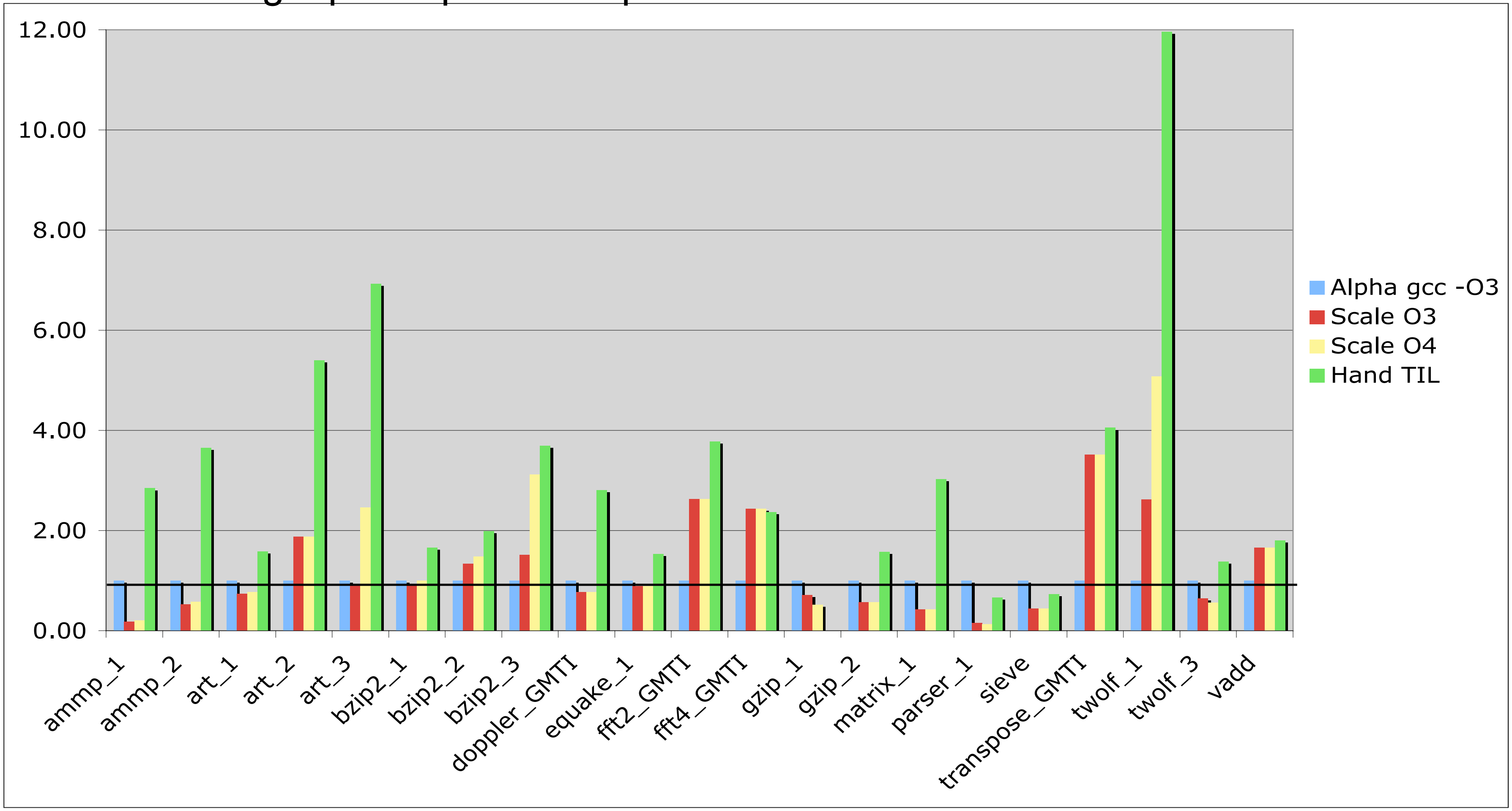


Performance Goals

- 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

- 🟥 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 parallal 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

Bit Offsets

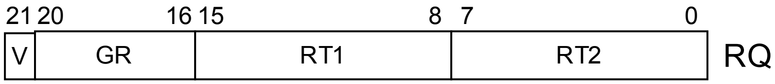
	31	27 6	5 0
0	H0	Read 0.0	Write 0.0
4	H1	Read 1.0	Write 1.0
8	H2	Read 2.0	Write 2.0
12	H3	Read 3.0	Write 3.0
16	H4	Read 0.1	Write 0.1
20	H5	Read 1.1	Write 1.1
24	H6	Read 2.1	Write 2.1
28	H7	Read 3.1	Write 3.1
	…	…	…
96	H24	Read 0.6	Write 0.6
100	H25	Read 1.6	Write 1.6
104	H26	Read 2.6	Write 2.6
108	H27	Read 3.6	Write 3.6
112	H28	Read 0.7	Write 0.7
116	H29	Read 1.7	Write 1.7
120	H30	Read 2.7	Write 2.7
124	H31	Read 3.7	Write 3.7

Byte Offsets

- 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

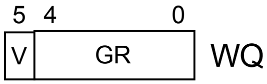
ISA: Read and Write Formats

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 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

8 7	6 5	4 3	2 1 0	
00	00	00	000	No Target
00	01	WID		Write Slot (WQ)
01	Inst. ID			Predicate Slot (IQ)
10	Inst. ID			OP0 Slot (IQ)
11	Inst. ID			OP1 Slot (IQ)

- 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

PR	Description
00	Not Predicated
01	(Reserved)
10	Predicated Upon False
11	Predicated Upon True

- 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)

ISA: Loads and Stores

Mnemonic	Name	Format	Sources	Targets
LB	Load Byte	L	OP0, IMM	T0
LBS	Load Byte Signed	L	OP0, IMM	T0
LH	Load Halfword	L	OP0, IMM	T0
LH	Load Halfword Signed	L	OP0, IMM	T0
LW	Load Word	L	OP0, IMM	T0
LWS	Load Word Signed	L	OP0, IMM	T0
LD	Load Doubleword	L	OP0, IMM	T0
SB	Store Byte	S	OP0, OP1, IMM	None
SH	Store Halfword	S	OP0, OP1, IMM	None
SW	Store Word	S	OP0, OP1, IMM	None
SD	Store Doubleword	S	OP0, OP1, IMM	None

- `LB:  T0 = ZEXT( MEM_B[OP0 + SEXT(IMM)] )`
- `LBS: T0 = SEXT( MEM_B[OP0 + SEXT(IMM)] )`
- `LH:  T0 = ZEXT( MEM_H[OP0 + SEXT(IMM)] )`
- `LHS: T0 = SEXT( MEM_H[OP0 + SEXT(IMM)] )`
- `LW:  T0 = ZEXT( MEM_W[OP0 + SEXT(IMM)] )`
- `LWS: T0 = SEXT( MEM_W[OP0 + SEXT(IMM)] )`
- `LD:  T0 = ZEXT( MEM_D[OP0 + SEXT(IMM)] )`

- `SH:  MEM_H[OP0 + SEXT(IMM)] = OP1[15:0]`
- `SW:  MEM_W[OP0 + SEXT(IMM)] = OP1[31:0]`
- `SD:  MEM_D[OP0 + SEXT(IMM)] = OP1[63:0]`

ISA: Integer Arithmetic

Mnemonic	Name	Format	Sources	Targets
ADD	Add	G	OP0, OP1	T0, T1
ADDI	Add Immediate	I	OP0, IMM	T0
SUB	Subtract	G	OP0, OP1	T0, T1
SUBI	Subtract Immediate	I	OP0, IMM	T0
MUL	Multiply	G	OP0, OP1	T0, T1
MULI	Multiply Immediate	I	OP0, IMM	T0
DIVS	Divide Signed	G	OP0, OP1	T0, T1
DIVSI	Divide Signed Immediate	I	OP0, IMM	T0
DIVU	Divide Unsigned	G	OP0, OP1	T0, T1
DIVUI	Divide Unsigned Immediate	I	OP0, IMM	T0

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

ISA: Integer Logical

Mnemonic	Name	Format	Sources	Targets
AND	Bitwise AND	G	OP0, OP1	T0, T1
ANDI	Bitwise AND Immediate	I	OP0, IMM	T0
OR	Bitwise OR	G	OP0, OP1	T0, T1
ORI	Bitwise OR Immediate	I	OP0, IMM	T0
XOR	Bitwise XOR	G	OP0, OP1	T0, T1
XORI	Bitwise XOR Immediate	I	OP0, IMM	T0

- XORI may be used to perform a bitwise complement – XORI(A, -1)

ISA: Integer Shift

Mnemonic	Name	Format	Sources	Targets
SLL	Shift Left Logical	G	OP0, OP1	T0, T1
SLLI	Shift Left Logical Immediate	I	OP0, IMM	T0
SRL	Shift Right Logical	G	OP0, OP1	T0, T1
SRLI	Shift Right Logical Immediate	I	OP0, IMM	T0
SRA	Shift Right Arithmetic	G	OP0, OP1	T0, T1
SRAI	Shift Right Arithmetic Immediate	I	OP0, IMM	T0

- 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

Mnemonic	Name	Format	Sources	Targets
EXTSB	Extend Signed Byte	G	OP0	T0, T1
EXTSH	Extend Signed Halfword	G	OP0	T0, T1
EXTSW	Extend Signed Word	G	OP0	T0, T1
EXTUB	Extend Unsigned Byte	G	OP0	T0, T1
EXTUH	Extend Unsigned Halfword	G	OP0	T0, T1
EXTUW	Extend Unsigned Word	G	OP0	T0, T1

- These are better than doing a left shift followed by a right shift

ISA: Integer Test

Mnemonic	Name	Format	Sources	Targets
TEQ	Test EQ	G	OP0, OP1	T0, T1
TNE	Test NE	G	OP0, OP1	T0, T1
TLE	Test LE	G	OP0, OP1	T0, T1
TLEU	Test LE Unsigned	G	OP0, OP1	T0, T1
TLT	Test LT	G	OP0, OP1	T0, T1
TLTU	Test LT Unsigned	G	OP0, OP1	T0, T1
TGE	Test GE	G	OP0, OP1	T0, T1
TGEU	Test GE Unsigned	G	OP0, OP1	T0, T1
TGT	Test GT	G	OP0, OP1	T0, T1
TGTU	Test GT Unsigned	G	OP0, OP1	T0, T1

- Test instructions produce true (1) or false (0)

ISA: Integer Test #2

Mnemonic	Name	Format	Sources	Targets
TEQI	Test EQ Immediate	I	OP0, IMM	T1
TNEI	Test NE Immediate	I	OP0, IMM	T1
TLEI	Test LE Immediate	I	OP0, IMM	T1
TLEUI	Test LE Unsigned Immediate	I	OP0, IMM	T1
TLTI	Test LT Immediate	I	OP0, IMM	T1
TLTUI	Test LT Unsigned Immediate	I	OP0, IMM	T1
TGEI	Test GE Immediate	I	OP0, IMM	T1
TGEUI	Test GE Unsigned Immediate	I	OP0, IMM	T1
TGTI	Test GT Immediate	I	OP0, IMM	T1
TGTUI	Test GT Unsigned Immediate	I	OP0, IMM	T1

- Test instructions produce true (1) or false (0)

ISA: Floating Point Arithmetic

Mnemonic	Name	Format	Sources	Targets
FADD	FP Add	G	OP0, OP1	T0, T1
FSUB	FP Substract	G	OP0, OP1	T0, T1
FMUL	FP Multiply	G	OP0, OP1	T0, T1
FDIV	FP Divide (simulator only!)	G	OP0, OP1	T0, T1

- 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
- The is no support for gradual underflow or denormal representations

ISA: Floating-Point Test

Mnemonic	Name	Format	Sources	Targets
FEQ	FP Test EQ	G	OP0, OP1	T0, T1
FNE	FP Test NE	G	OP0, OP1	T0, T1
FLE	FP Test LE	G	OP0, OP1	T0, T1
FLT	FP Test LT	G	OP0, OP1	T0, T1
FGE	FP Test GE	G	OP0, OP1	T0, T1
FGT	FP Test GT	G	OP0, OP1	T0, T1

- These operations are performed assuming double-precision values
- Test instructions produce true (1) or false (0)

ISA: Floating Point Conversion

Mnemonic	Name	Format	Sources	Targets
FITOD	Convert Integer to Double FP	G	OP0	T0, T1
FDTOI	Convert Double FP to Integer	G	OP0	T0, T1
FSTOD	Convert Single FP to Double FP	G	OP0	T0, T1
FDTOS	Convert Double FP to Single FP	G	OP0	T0, T1

- 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

ISA: Control Flow

Mnemonic	Name	Format	Sources	Targets
BR	Branch	B	OP0	PC
BRO	Branch with Offset	B	OFFSET	PC +=
CALL	Call	B	OP0	PC
CALLO	Call with Offset	B	OFFSET	PC +=
RET	Return	B	OP0	PC
SCALL	System Call	B	None	PC

- 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

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

- 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

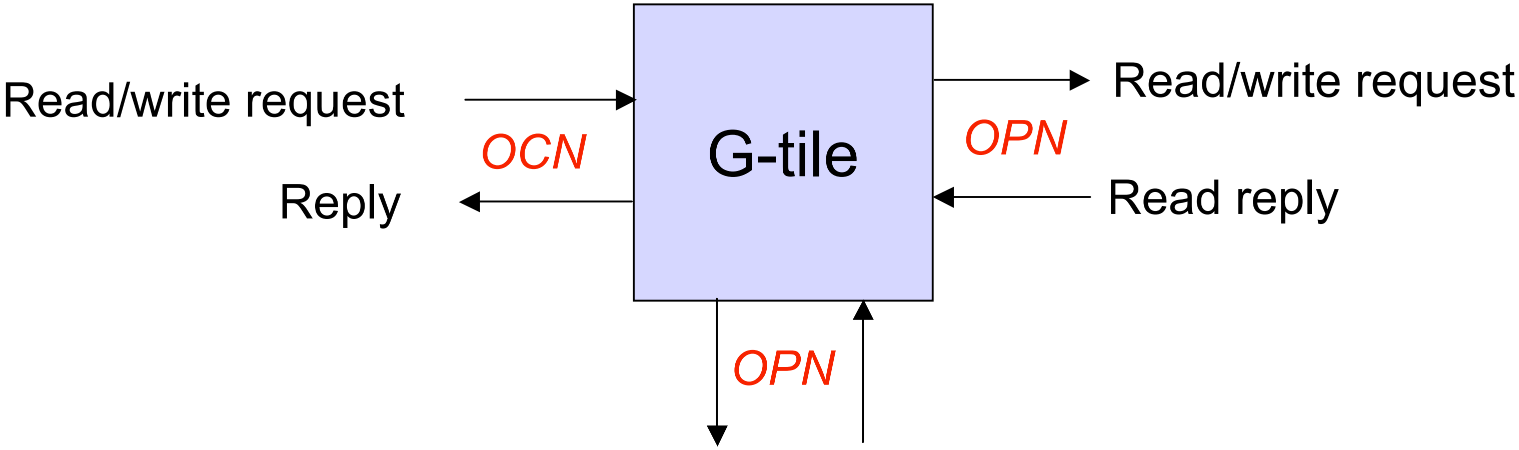
Bandwidth Summary

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

(Maximum bandwidth calculations)

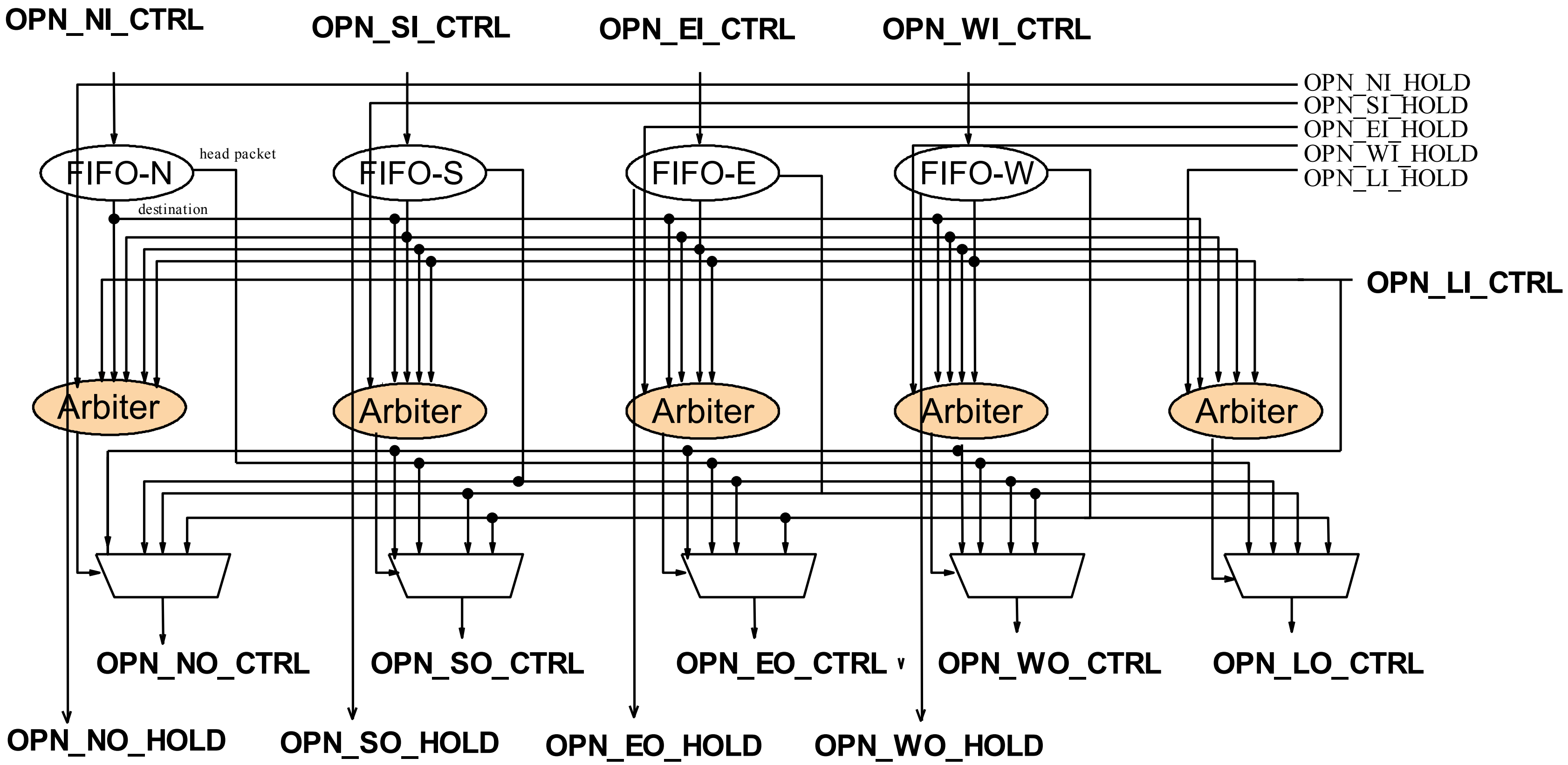
G-tile: Access to Architected Registers

Access to architected registers allowed when processor is halted



Only one read/write request handled at a time

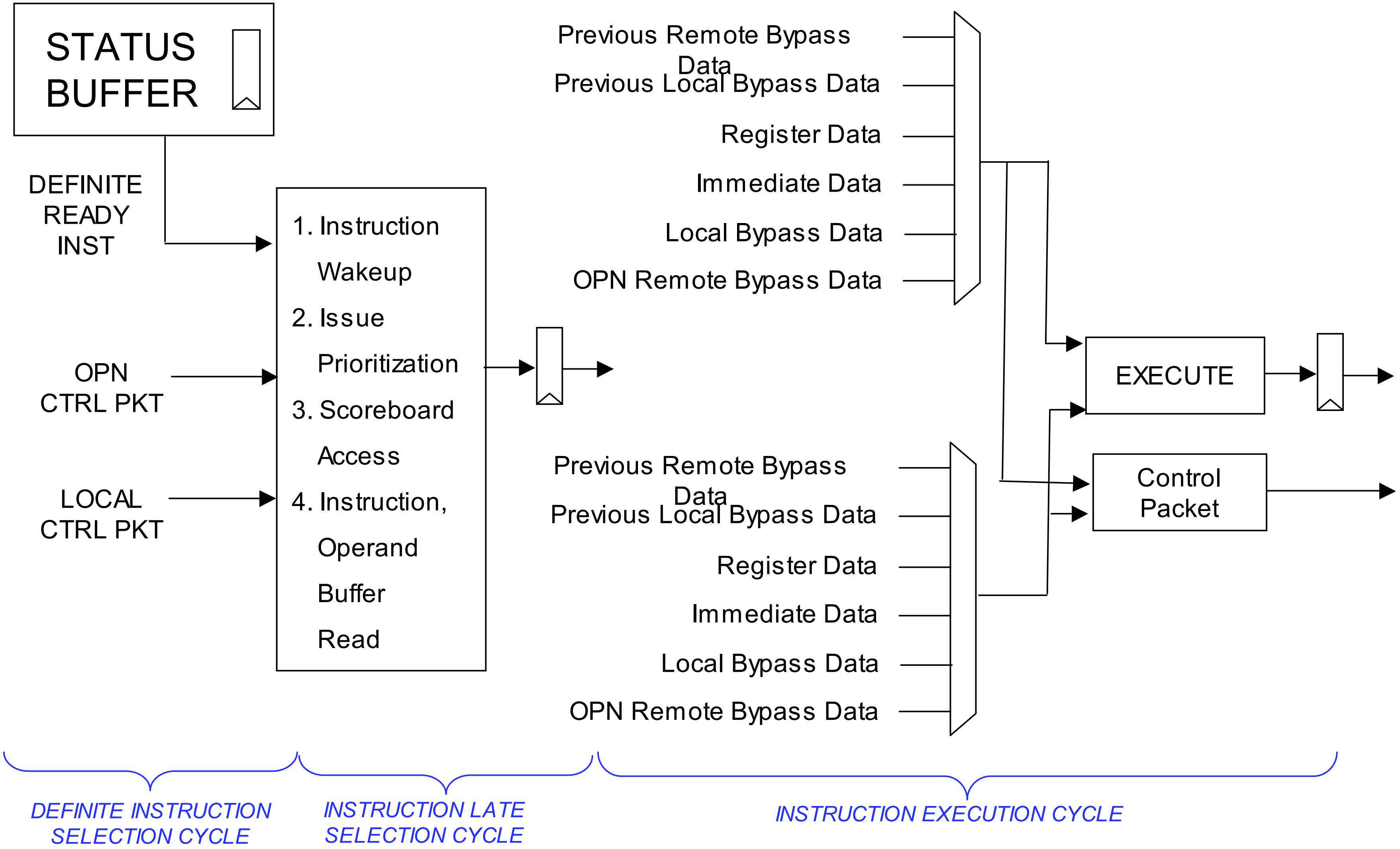
OPN: Detailed Router Schematic



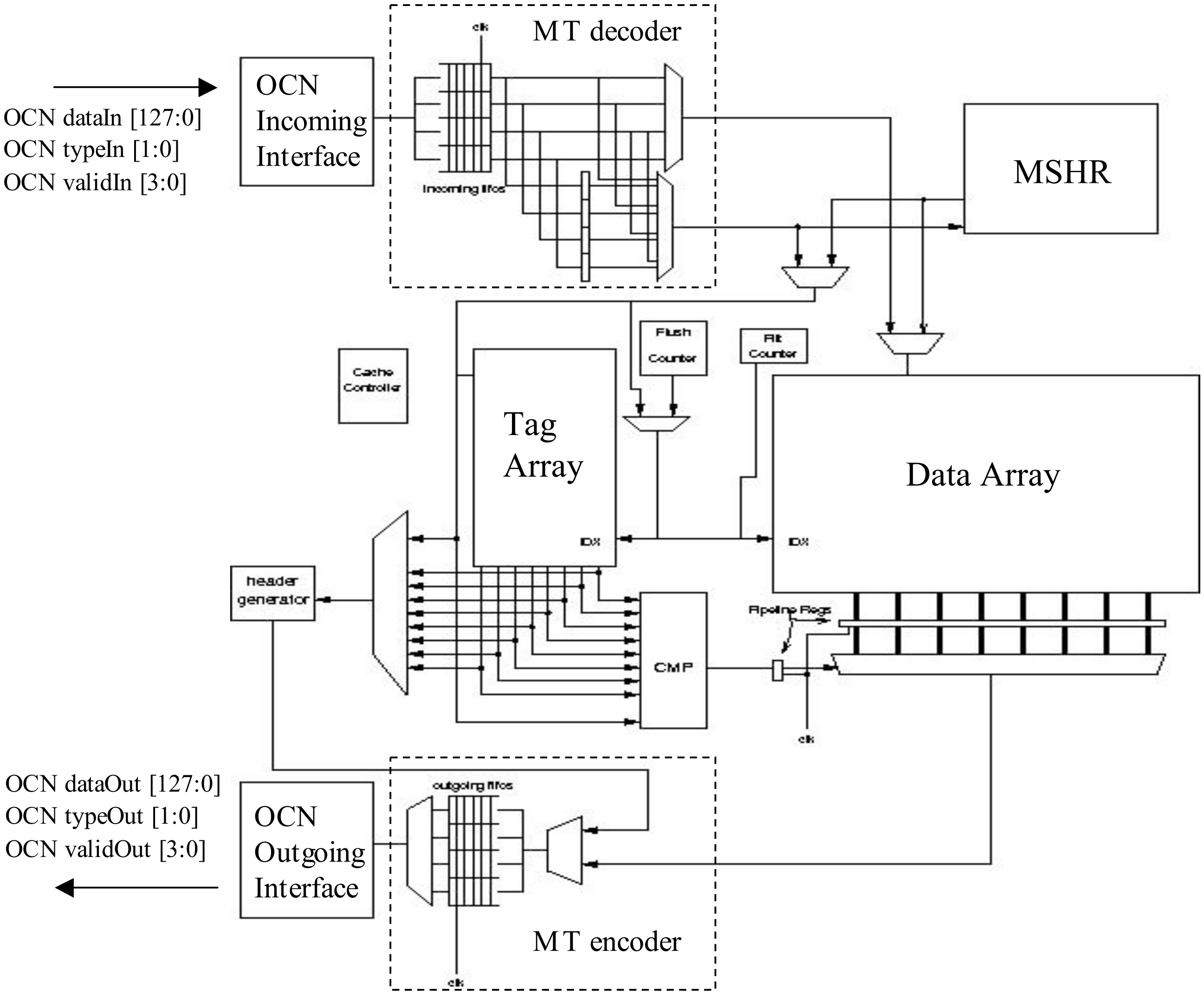
E-Tile: List of Pipelines

- 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



M-Tile: Internal Structure



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

OCN Incoming Interface

MT decoder

MSHR

Tag Array

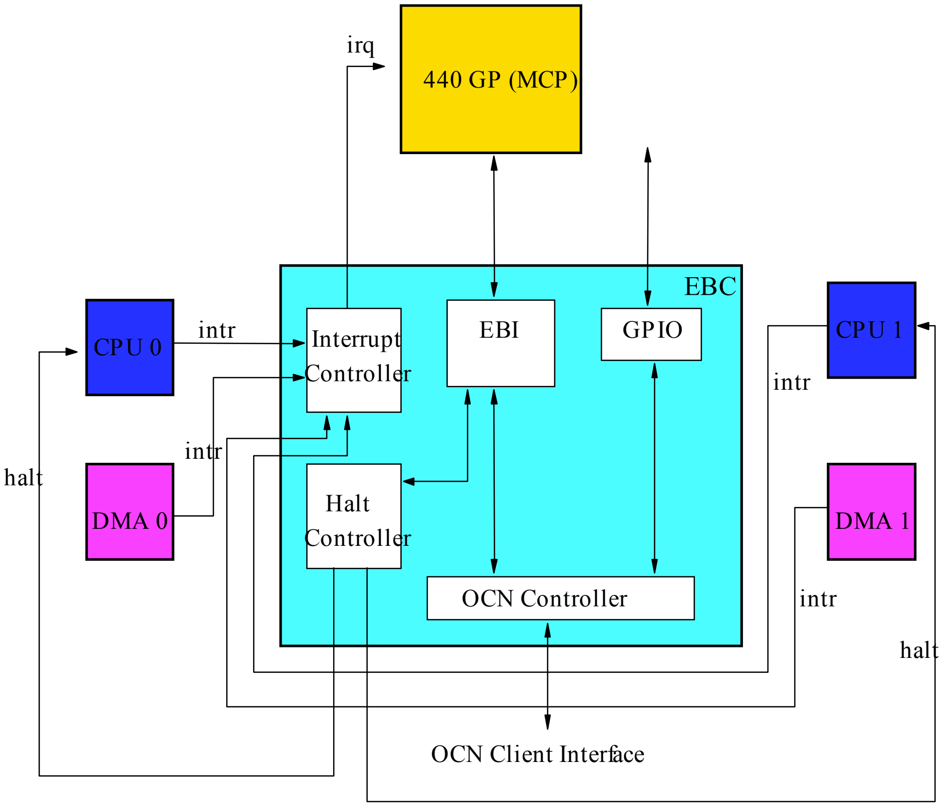
Data Array

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

OCN Outgoing Interface

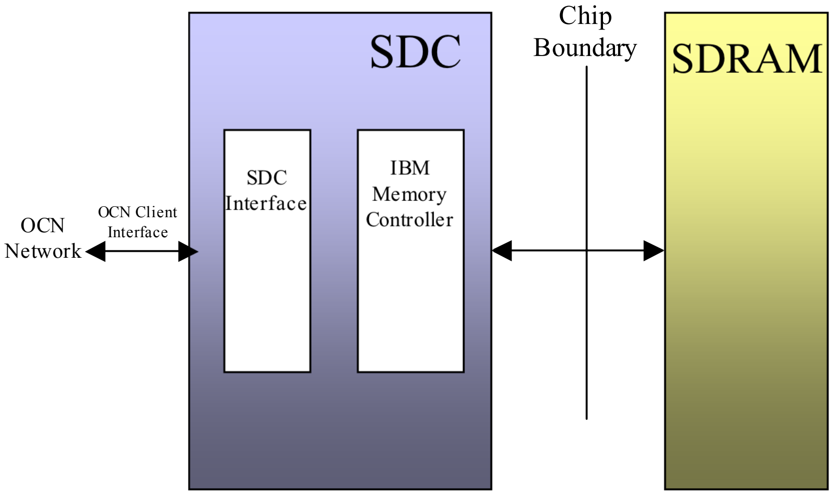
MT encoder

EBC: External Bus Controller Datapath



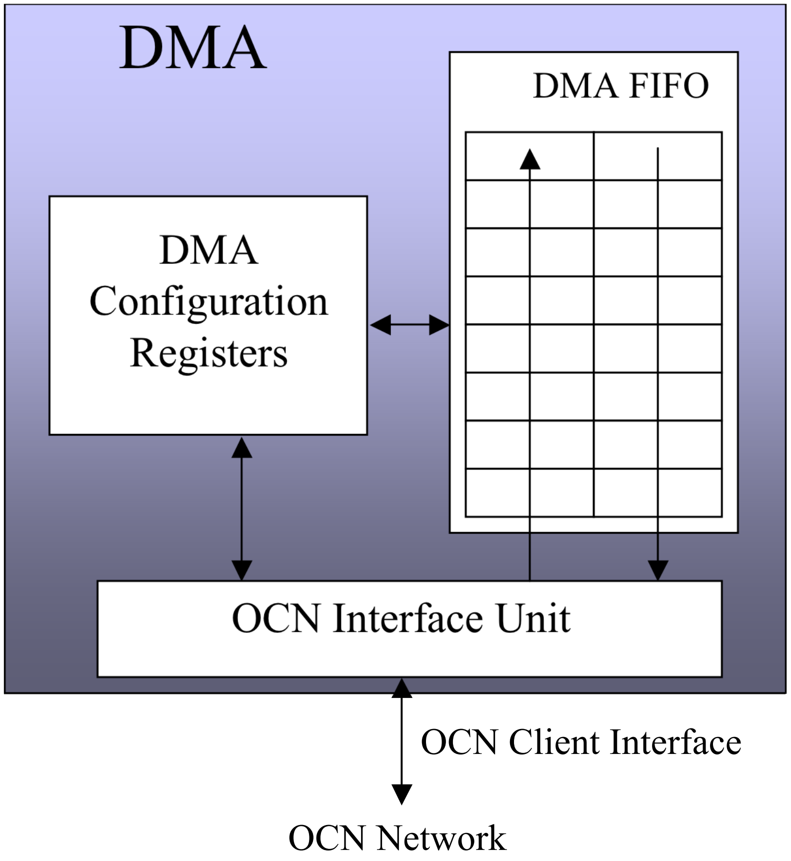
- Provides an interface between the 440 GP and TRIPS chips
- GPIOs for general purpose I/O
- Synchronizes between 66MHz 440GP bus and system clock
- Interrupt process:
  - CPU's and DMAs notify EBC of an exception through the "intr" interface
  - Interrupt controller calls service routines in 440 GP via irq line
  - 440 GP writes to the Halt Controller's registers to stop the processors
  - 400 GP services the interrupt

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