

HW2 review / GPUs

To read more...

This day's papers:

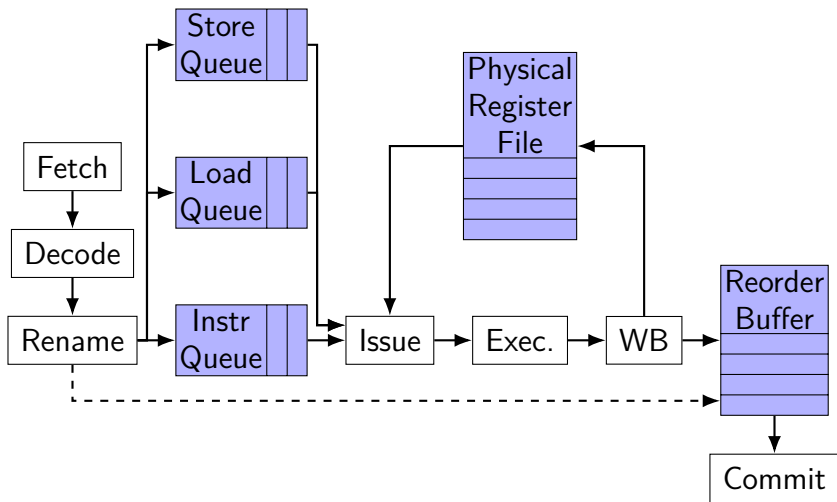
Lee et al, "Debunking the 100X GPU vs. CPU myth: an evaluation of throughput computing on CPU and GPU"

Lee et al, "Exploring the Design Space of SPMD Divergence Management on Data-Parallel Architectures"

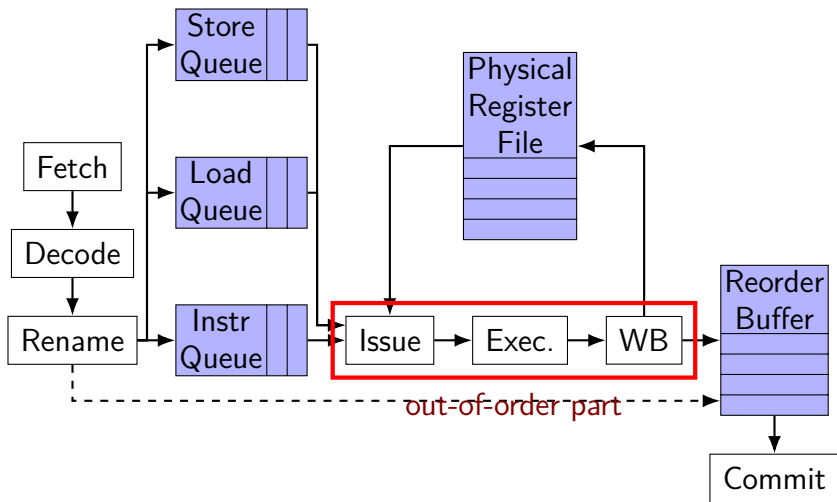
Supplementary readings:

Volokv and Demmel, "Benchmarking GPUs to Tune Dense Linear Algebra"

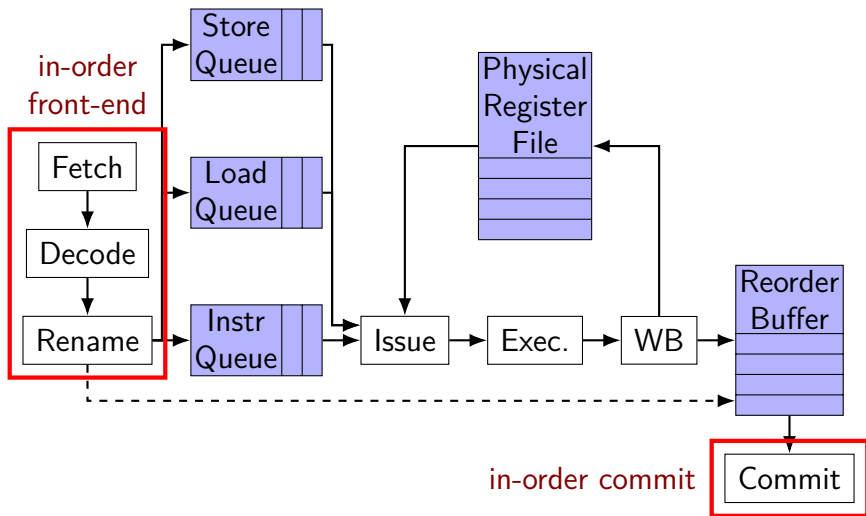
gem5 out-of-order CPU stages



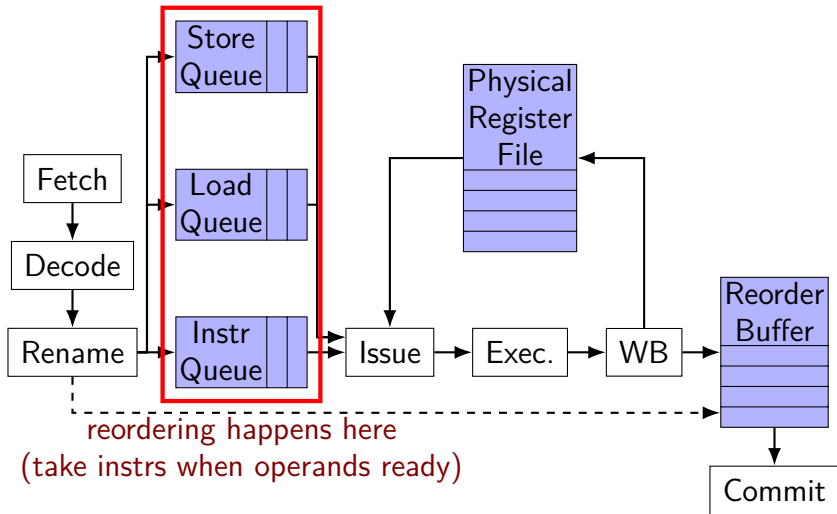
gem5 out-of-order CPU stages



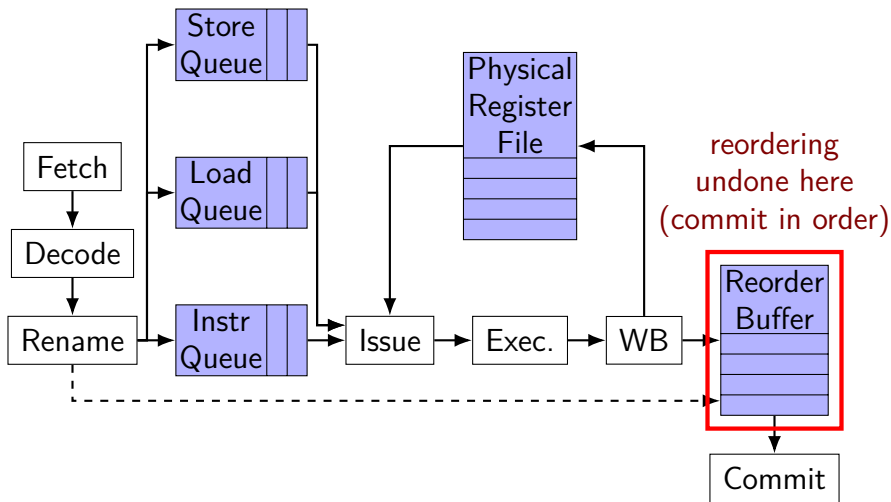
gem5 out-of-order CPU stages



gem5 out-of-order CPU stages

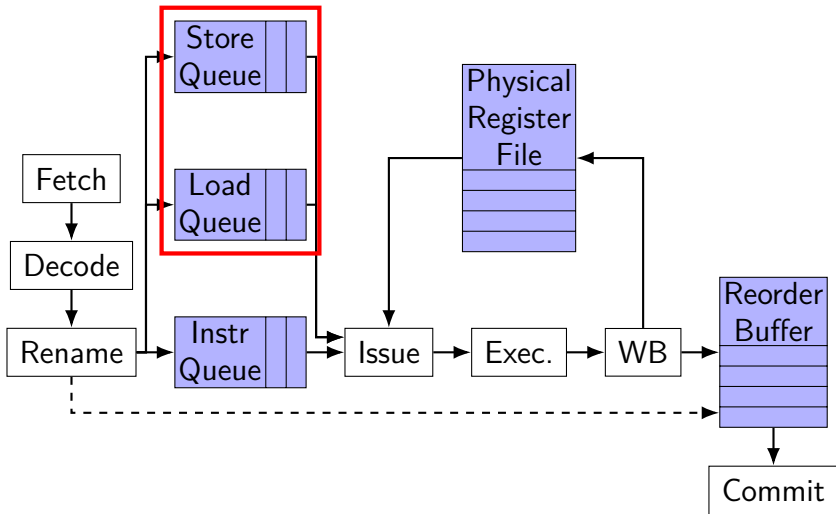


gem5 out-of-order CPU stages

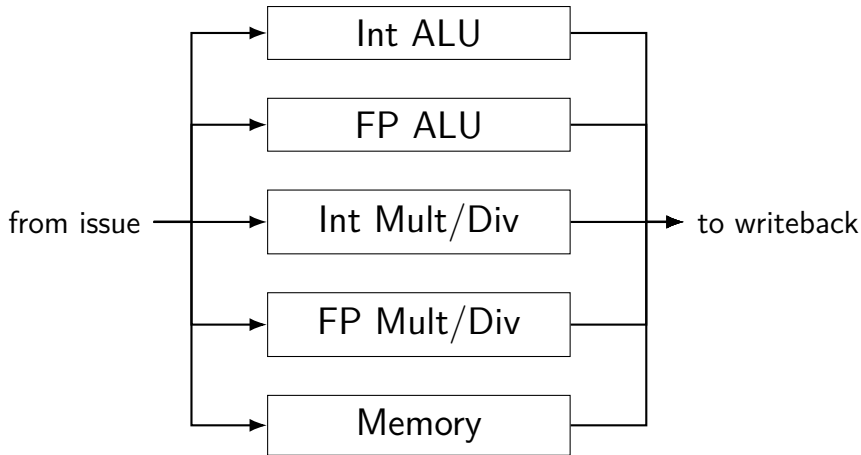


gem5 out-of-order CPU stages

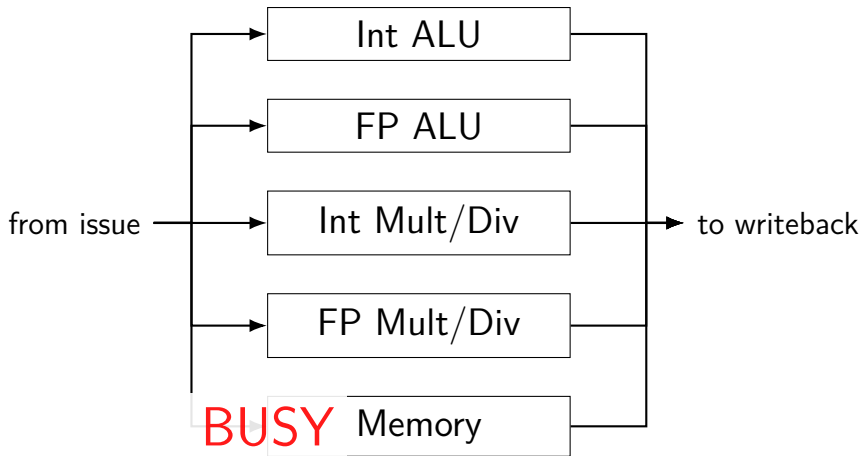
memory ordering
happens here



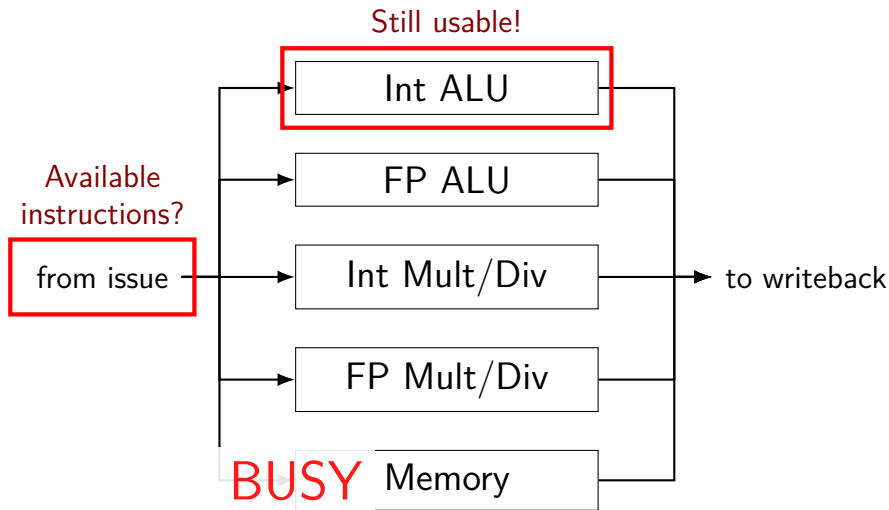
overlap possible: execute



overlap possible: execute



overlap possible: execute



overlap possibilities

Load Queue

| |
|--------------------------------------|
| $R4 \leftarrow \text{memory}[10000]$ |
| ... |

Instr Queue

| |
|-------------------------|
| $R5 \leftarrow R4 + R3$ |
| $R6 \leftarrow R2 - R4$ |
| $R7 \leftarrow R4 * R0$ |
| $R8 \leftarrow R4 * R0$ |
| $R9 \leftarrow R4 / R1$ |
| ... |

Load Queue

| |
|--------------------------------------|
| $R4 \leftarrow \text{memory}[10000]$ |
| ... |

Instr Queue

| |
|---------------------------|
| $R5 \leftarrow R2 + R3$ |
| $R6 \leftarrow R2 - R5$ |
| $R7 \leftarrow R3 * R0$ |
| $R8 \leftarrow R3 * R0$ |
| $R9 \leftarrow R3 / R1$ |
| ... |
| $R34 \leftarrow R4 / R33$ |

overlap possibilities

Load Queue

| |
|--------------------------------------|
| $R4 \leftarrow \text{memory}[10000]$ |
| ... |

Instr Queue

| |
|-------------------------|
| $R5 \leftarrow R4 + R3$ |
| $R6 \leftarrow R2 - R4$ |
| $R7 \leftarrow R4 * R0$ |
| $R8 \leftarrow R4 * R0$ |
| $R9 \leftarrow R4 / R1$ |
| ... |

**everything
needs R4**

Load Queue

| |
|--------------------------------------|
| $R4 \leftarrow \text{memory}[10000]$ |
| ... |

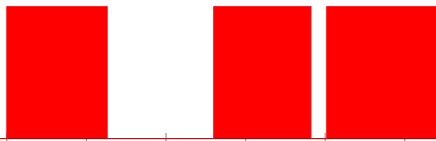
Instr Queue

| |
|---------------------------|
| $R5 \leftarrow R2 + R3$ |
| $R6 \leftarrow R2 - R5$ |
| $R7 \leftarrow R3 * R0$ |
| $R8 \leftarrow R3 * R0$ |
| $R9 \leftarrow R3 / R1$ |
| ... |
| $R34 \leftarrow R4 / R33$ |

**nothing
needs R4**

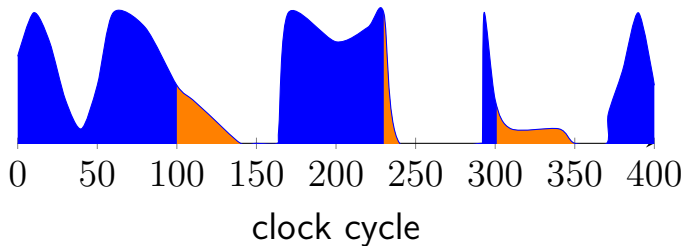
identifying overlap

active cache misses



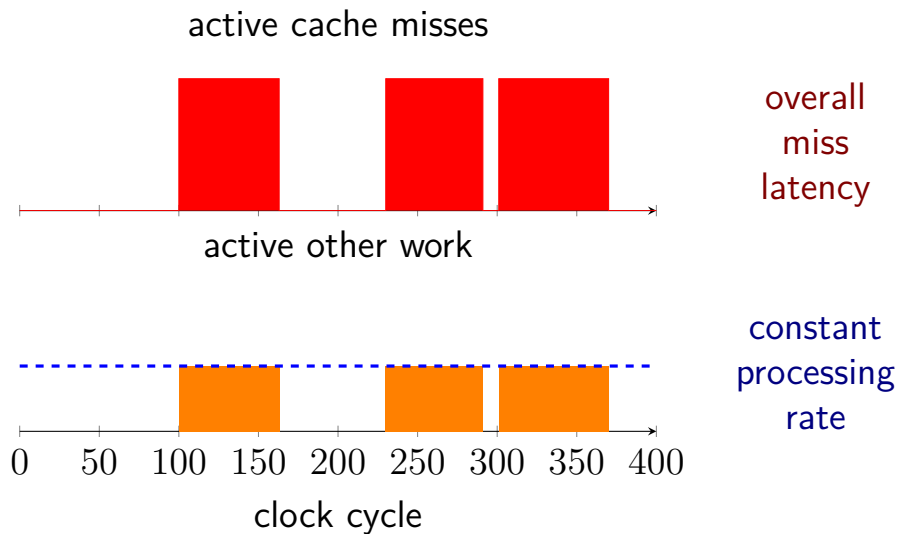
overall
miss
latency

active other work



ops/cycle

model one: “perfect” overlap



model one: “perfect” overlap

active cache misses

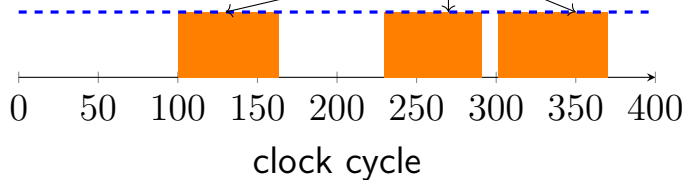


overall
miss
latency

active other work

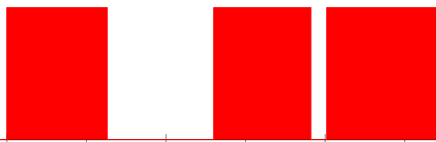
$\text{overlap} \approx \text{miss latency} \times \text{ops/cycle}$

constant
processing
rate

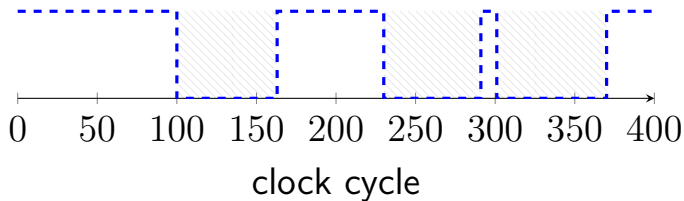


model two: no overlap

active cache misses



active other work



overall
miss
latency

overlap ≈ 0

huge cache and no overlap

active cache misses

**eliminate all cache misses
with larger cache**

active other work

ops/cycle if no overlap

finish if no overlap

0 50 100 150 200 250 300 350 400

clock cycle

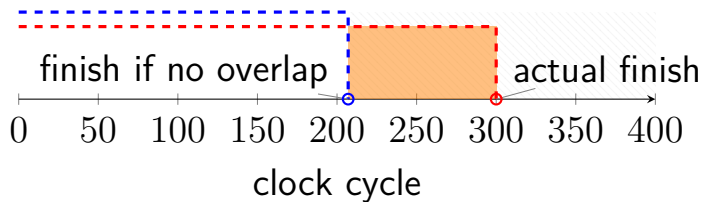
no overlap:
 $\Delta\text{time} =$
miss latency

actual results

active cache misses

**eliminate all cache misses
with larger cache**

active other work

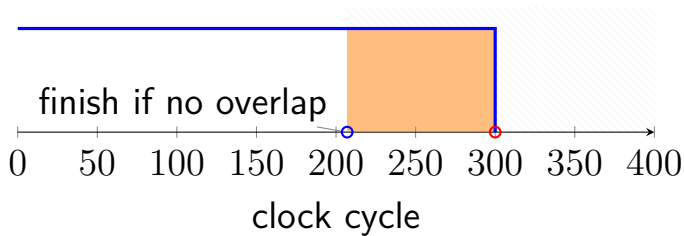


guessed timeline

with cache misses



without cache misses

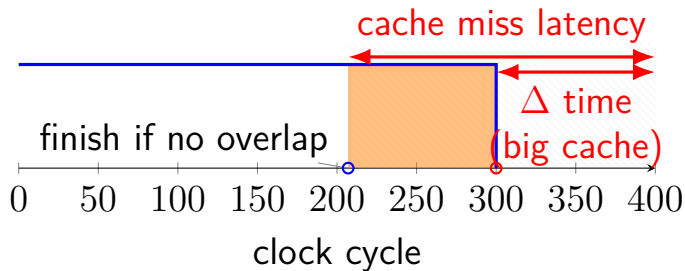


guessed timeline

with cache misses



without cache misses



caveats

uneven rate of work — what is 10%?

- of operations?

- of execute latencies?

- of time without memory delays?

branch mispredictions, etc. changes

not all cache misses eliminated

- still have compulsory misses

- significant for this very short program

multiple overlapping

Load Queue

| |
|---------------------------------|
| R4 \leftarrow memory[0x10000] |
| R5 \leftarrow memory[0xFA233] |
| R6 \leftarrow memory[0x10004] |
| R8 \leftarrow memory[0x10008] |
| ... |

cache (2-ways, 16B blocks, 256 sets)

| set | valid | tag | data | valid | tag | data |
|-----|-------|------|--------------------------------|-------|------|--------------------------------|
| 00 | 0 | | | 1 | 0x23 | M[0x23000] to M[0x2300F] |
| 01 | 1 | 0x43 | M[0x43010] to M[0x4301F] | 1 | 0x23 | M[0x23010] to M[0x2301F] |

multiple overlapping

Load Queue

| |
|---------------------------------|
| R4 \leftarrow memory[0x10000] |
| R5 \leftarrow memory[0xFA233] |
| R6 \leftarrow memory[0x10004] |
| R8 \leftarrow memory[0x10008] |
| ... |

miss for 0x10000 brings in block!

cache (2-ways, 16B blocks, 256 sets)

| set | valid | tag | data | valid | tag | data |
|-----|-------|------|--------------------------------|-------|------|--------------------------------|
| 00 | 1 | 0x00 | M[0x10000] to M[0x1000F] | 1 | 0x23 | M[0x23000] to M[0x2300F] |
| 01 | 1 | 0x43 | M[0x43010] to M[0x4301F] | 1 | 0x23 | M[0x23010] to M[0x2301F] |

multiple overlapping

Load Queue

| |
|---------------------------------|
| R4 \leftarrow memory[0x10000] |
| R5 \leftarrow memory[0xFA233] |
| R6 \leftarrow memory[0x10004] |
| R8 \leftarrow memory[0x10008] |
| ... |

later accesses to block now hit!
if **started after 0x10000 done**

cache (2-ways, 16B blocks, 256 sets)

| set | valid | tag | data | valid | tag | data |
|-----|-------|------|--------------------------------|-------|------|--------------------------------|
| 00 | 1 | 0x00 | M[0x10000] to M[0x1000F] | 1 | 0x23 | M[0x23000] to M[0x2300F] |
| 01 | 1 | 0x43 | M[0x43010] to M[0x4301F] | 1 | 0x23 | M[0x23010] to M[0x2301F] |

latency counting

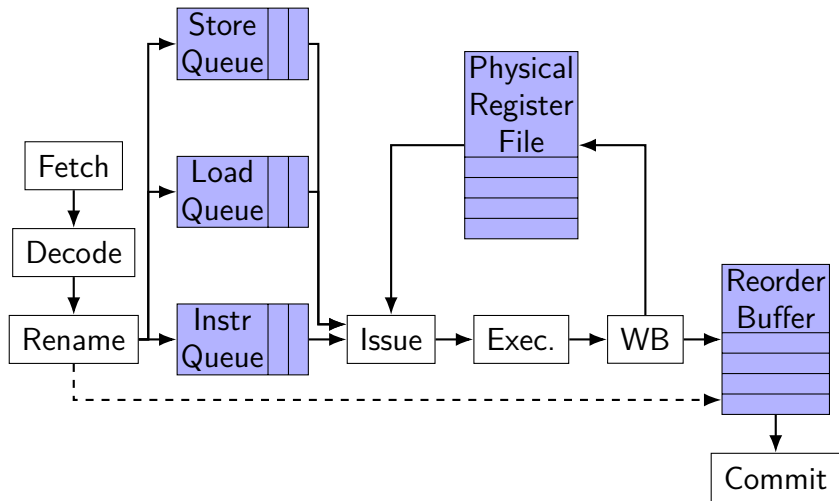
overlapping accesses to same block

two **misses**

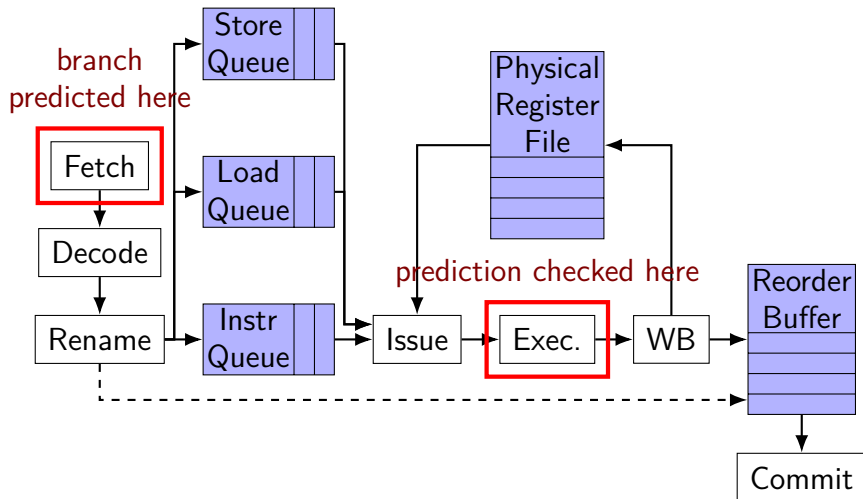
lower average latency — access already started

counted twice — latency for **each access**

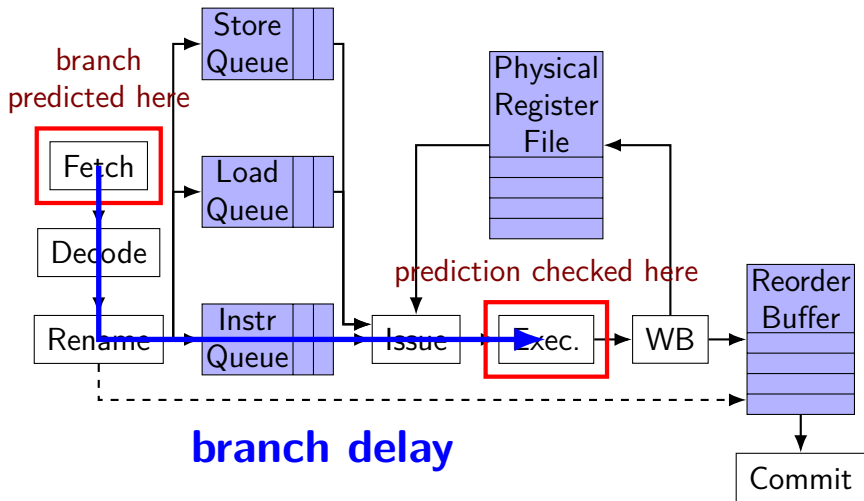
detecting branch mispredict



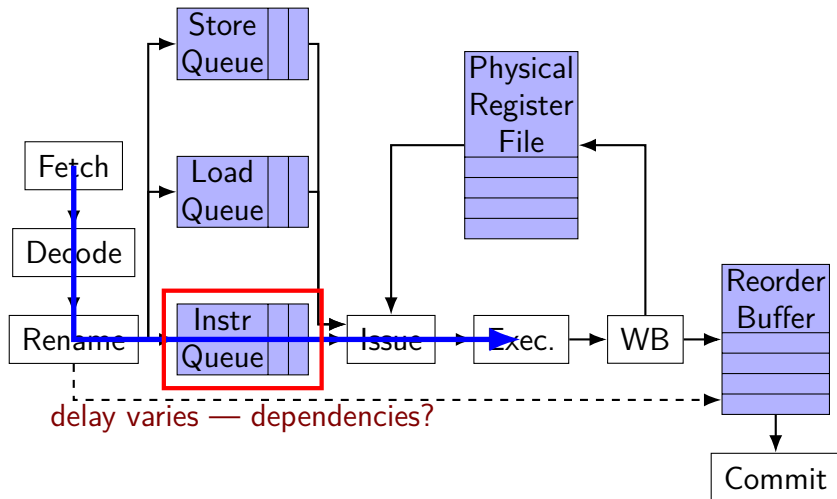
detecting branch mispredict



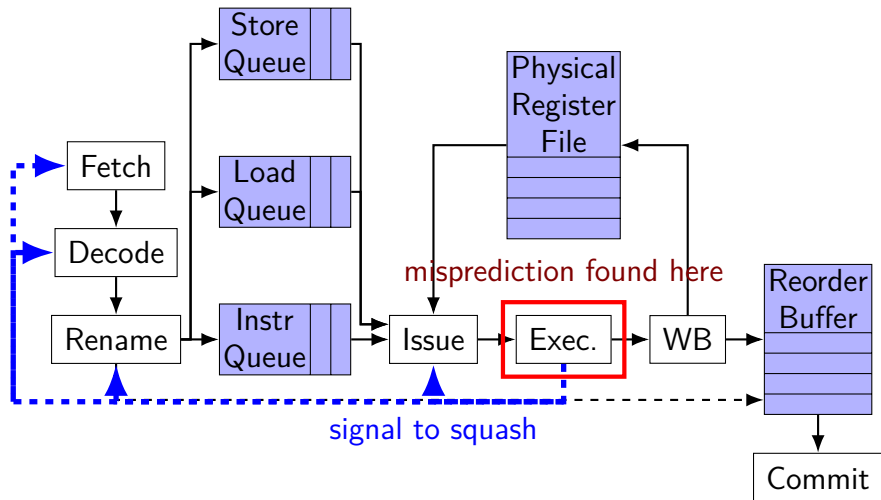
detecting branch mispredict



detecting branch mispredict



acting on branch mispredict



what is squashing

fetch — cancel requests to instruction cache

decode, rename — discard queued instructions

issue — clean up instruction/load/store queues
instruction finished rename, but not writeback

commit — clean up ROB entries
instruction finished rename

misprediction in misprediction

```
Y = Z = 0
X ← Y * Z
IF X > 0
    GOTO L1
W ← X + Y
```

```
L1:
IF Y > 0
    GOTO L2
A ← B + C
```

```
L2:
F ← D + E
```

| <i>fetch/rename</i> | <i>branch FU</i> | <i>mult FU</i> |
|---------------------|------------------|-----------------|
| X ← Y + Z | — | — |
| IF X > 0 ... | — | X ← Y * Z (1/3) |
| IF Y > 0 ... | — | X ← Y * Z (2/3) |
| F ← D + E | Y > 0 | X ← Y * Z (3/3) |
| A ← B + C | X > 0 | — |
| W ← X + Y | — | — |

misprediction in misprediction

```
Y = Z = 0
X ← Y * Z
IF X > 0
    GOTO L1
W ← X + Y
```

```
L1:
IF Y > 0
    GOTO L2
A ← B + C
```

```
L2:
F ← D + E
```

| <i>fetch/rename</i> | <i>branch FU</i> | <i>mult FU</i> |
|---------------------|------------------|-----------------|
| X ← Y + Z | — | — |
| IF X > 0 ... | — | X ← Y * Z (1/3) |
| IF Y > 0 ... | — | X ← Y * Z (2/3) |
| F ← D + E | mispredict Y > 0 | * Z (3/3) |
| A ← B + C | X > 0 | — |
| W ← X + Y | — | — |

misprediction in misprediction

```
Y = Z = 0
X ← Y * Z
IF X > 0
    GOTO L1
W ← X + Y
```

```
L1:
IF Y > 0
    GOTO L2
A ← B + C
```

| <i>fetch/rename</i> | <i>branch FU</i> | <i>mult FU</i> |
|---------------------|------------------|-----------------|
| X ← Y + Z | — | — |
| IF X > 0 ... | — | X ← Y * Z (1/3) |
| IF Y > 0 ... | — | X ← Y * Z (2/3) |
| F ← D + E | Y > 0 | X ← Y * Z (3/3) |
| A ← B + C | X > 0 | — |
| W ← X + Y | — | — |

mispredict X > 0

```
L2:
F ← D + E
```

costs of branch misprediction

time spent running work that can't commit
(instead of work from the correct branch)

time spent squashing instructions

cache pollution from mispredicted loads

estimating branch prediction cost/benefit

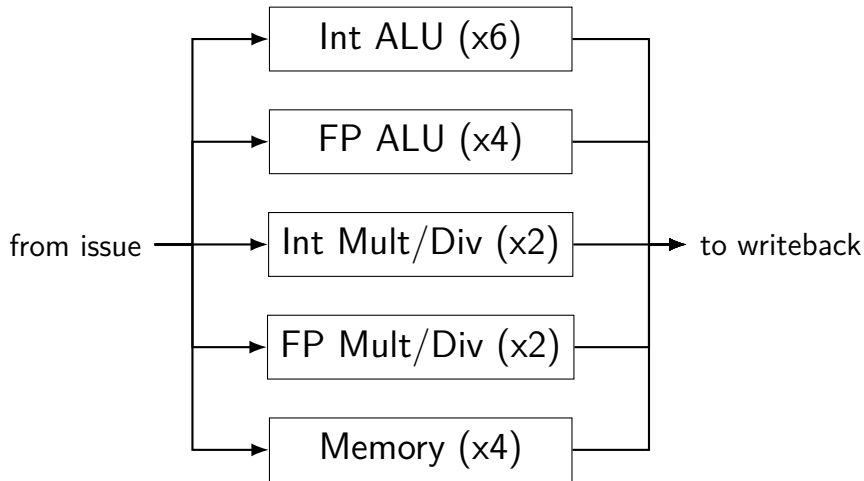
total cost \approx portion of instructions run in incorrect
branch

assumption: same amount as would be in correct
branch

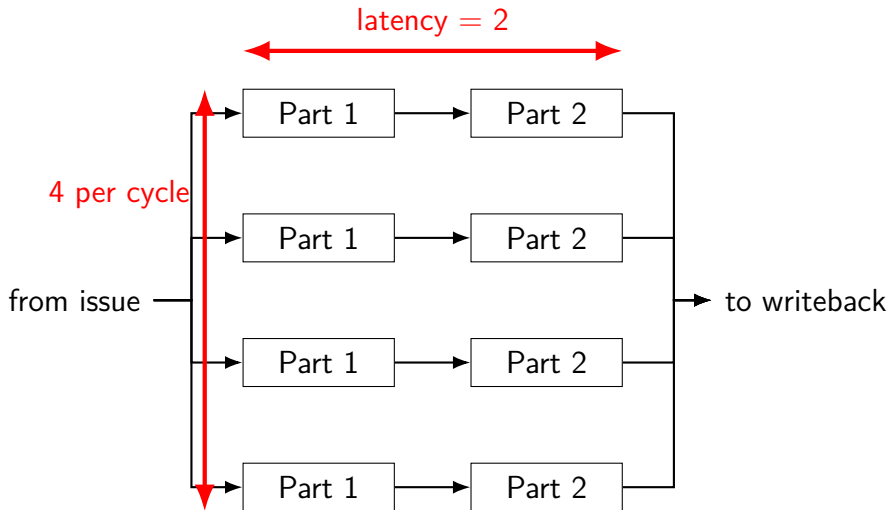
probably not true — e.g. loop versus after loop

benefit: $\#$ correct predictions \times cost per
misprediction

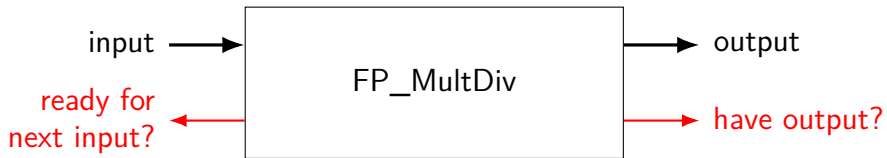
the execute stage



pipelined FP ALU



variable speed functional units



| op type | output in... | ready in ... |
|-----------|---------------------|---------------------------|
| FloatMult | 4 cycles (latency) | 1 cycle (pipelined) |
| FloatDiv | 12 cycles (latency) | 12 cycles (not pipelined) |
| FloatSqrt | 24 cycles (latency) | 24 cycles (not pipelined) |

maximum speed of execute (1)

consider a program with one million FP_ALU ops

... and nothing else

4 FP_ALU functional units

$1\ 000\ 000 \div 4 = 250\ 000$ cycles

4 ops per cycle

maximum speed of execute (2)

consider a program with one million FP_ALU ops

... and **one thousand IntALU ops**

250 000 cycles to issue FP_ALU ops

1000 IntALU ops need $\lceil 1000 \div 6 \rceil = 167$ cycles

total time = 250 000 cycles (not 250 167)

4.004 ops/cycle

determining maximum speed

issue rate for **each functional unit**?

pipelined — *count* per cycle

not pipelined — *count* per *latency* cycles

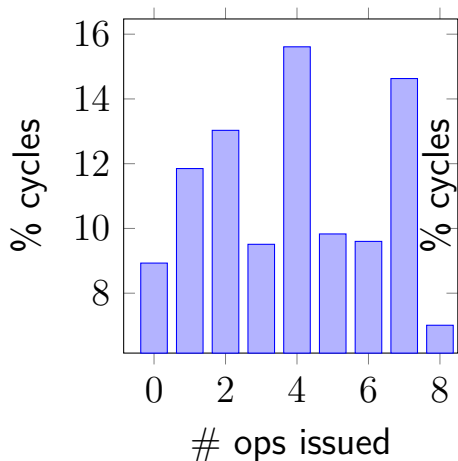
mixed — depends ratio of instruction types

which functional unit is the **bottleneck**

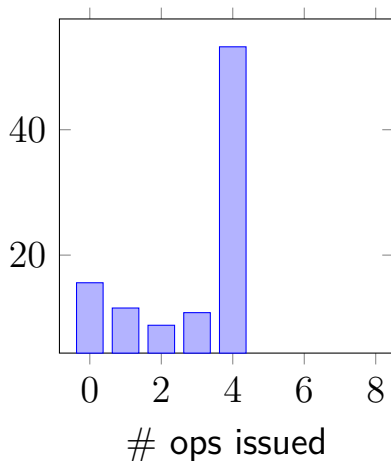
keep instruction ratio constant

actual issue rates — Matmul

all widths 8



all widths 4



widths and branch prediction

wider pipeline — more of mispredicted branches completed

bad for queens

SPMD comments

prediction versus predication

does this result really matter?

what is the actual HW cost of HW divergence management?

SPMD: Predication

vector instructions that operate based on a **mask**

the mask is called a “predicate”

e.g.

```
if (mask[i]) { vresult[i] = va[i] + vb[i]
```

paper's notation:

```
@vresult add vresult, va, vb
```

not prediction

Easy speedup

write some really inefficient code for platform X

spend lots of time optimizing for platform Y

platform Y is 100x faster than platform X!

Easy speedup

write some really inefficient code for ~~platform X~~ CPUs

spend lots of time optimizing for ~~platform Y~~ GPUs

~~platform Y~~ is GPUs are 100x faster than ~~platform X~~ CPUs!

CPU optimization techniques

Multithreading — use multiple cores

(Yes, really, people didn't do this when comparing...)

Cache blocking (Goto paper)

Plan what is in the cache

Split problem into cache-sized units

Reordering data

CPUs have vector support, but must be contiguous

GPU optimization techniques

Avoid synchronization

Corollary: do lots of work with one kernel call

Make use of shared buffer

- Explicitly managed cache

- Replacement for cache blocking

Floating Point BW

paper's CPU: 102 GFlop/sec

$3.2 \text{ GHz} \times 4 \text{ cores} \times 4 \text{ SIMD lanes} \times 2 \text{ FP op/cycle}$

paper's GPU: 934 GFlop/sec.

with fused multiply-add, special functional unit

Intel Core i7-6700: 435 GFlop/sec

with fused-multiply-add

NVidia Tesla P100: 9300 GFlop/sec

Memory BW

paper's CPU: 32 GB/sec? (to normal DRAM)

paper's GPU: 141 GB/sec (to off-chip, on-GPU memory)

paper's GPU: 8 GB/sec to/from CPU memory

On-chip storage

paper's CPU: approx. 6KB registers + 8MB caches
(12KB registers with SMT)

paper's GPU: approx. 2MB registers + 480KB
shared memory + 232KB caches

NVidia Tesla P100: approx. 14 MB registers + 3MB
shared memory/cache + 512KB caches

The stride challenge

```
struct Color { float red; float green; float blue
```

```
Color colors[N];
```

```
...
```

```
for (int i = 0; i < N; ++i) {  
    colors[i].red *= 0.8;  
}
```

needs **strided memory access**

Intel has vector instructions, but not this kind of load/store

AoS versus SoA

// Array of Structures

```
struct Color { float red; float green; float blue  
Color colors[N];
```

...

```
colors[i].red *= 0.8
```

// Structure of Array

```
struct Colors {  
    float reds[N];  
    float greens[N];  
    float blues[N];
```

```
};
```

```
Colors colors;
```

...

```
colors.reds[i] *= 0.8
```


honest performance comparisons

sometimes — fundamental limits

- peak floating point operations

- memory bandwidth + minimal communication

often research doesn't know how to optimize on
“other” platform

lots of subtle tuning

CPU SIMD support

Modern CPUs support vector operations

Generally less flexible than GPUs

Still many, many less ALUs/chip than GPUs

x86 SIMD timeline (1)

Intel MMX (1997 Pentium)

- 64-bit registers, vector 32/16/8-bit integer instructions
- 'saturating' add/subtract (overflow yields MAX_INT)
- 64-bit loads/stores (contiguous only)

AMD 3DNow! (1998 AMD K6-2)

- 64-bit registers, vector 32-bit float instructions

Intel SSE/SSE2 (1999 Pentium III; 2001 Pentium 4)

- 128-bit registers
- vector 32/64-bit float instructions
- vector 32/16/8-bit integer instructions
- 128-bit loads/stores (contiguous only)
- vector 'shuffling' instructions

x86 SIMD timeline (2)

Intel SSE3/SSE4

Intel AVX (2011 Sandy Bridge)

- 256-bit registers
- floating point only

Intel AVX2 (2013 Haswell)

- 256-bit registers
- fused multiply-add
- adds integer instructions

Intel AVX-512 (2015 Knights Landing)

- 512-bit registers (maybe)
- scatter/gather instructions
- vector mask/predication support

horizontal instructions

```
// Horizontal ADD Packed Double  
// %xmm1, %xmm2 are vectors of two  
// 64-bit floating point values  
haddpd %xmm1, %xmm2  
// XMM2[0] ← XMM1[0] + XMM1[1]  
// XMM2[1] ← XMM2[0] + XMM2[1]
```

predicate notation

```
@vf0 vx = vy
```

```
// same as:
```

```
forall i: if (vf0[i]) vx[i] ← vx[j]
```

Predicated instructions

```
// forall i:  
//   vf0[i] ← (va[i] < vb[i])  
vf0 = vslt va, vb
```

```
// forall i:  
//   if (vf0[i])  
//     vc[i] ← vop1[i]  
@vf0 vc = vop1
```

```
// forall i:  
//   if (!vf0[i])  
//     vc[i] ← vop2[i]  
!@vf0 vc = vop2
```

Skipping + predication

```
    vf0 = vslt va, vb
    s0 = vpopcnt vf0
    // if all vf0[i] == 0:
    //    goto else
    branch.eqz s0, else
@vf0 vc = vop1
    s1 = vpopcnt !vf0
    // if all vf0[i] == 1:
    //    goto out
    branch.eqz s1, out
else:
!@vf0 vc = vop2
out:
```


Predicated instructions: hardware skipping

```
vf0 = vslt va, vb
push.stack out
tbranch.eqz vf0, else
vc = vop1
pop.stack
else:
vc = vop2
pop.stack
out:
```

divergence stack

```
push.stack out
tbranch.eqz vf0, else
vc = vop1
pop.stack
else:
vc = vop2
pop.stack
out:
```

state: thread mask, *divergence stack*

divergence stack

```
push.stack out
tbranch.eqz vf0, else
vc = vop1
pop.stack
else:
vc = vop2
pop.stack
out:
```

state: thread mask, *divergence stack*

Case 1: both taken

divergence stack

| PC | thread mask |
|------|-------------------------|
| else | (not vf0) and startMask |
| out | startMask |
| ... | ... |

tbranch: push else+mask, set mask, **goto vop1**
(set mask using vf0
normal next instruction)

divergence stack

```
push.stack out
tbranch.eqz vf0, else
vc = vop1
pop.stack
else:
vc = vop2
pop.stack
out:
```

state: thread mask, *divergence stack*

Case 1: both taken

divergence stack

| PC | thread mask |
|------|-------------------------|
| else | (not vf0) and startMask |
| out | startMask |
| ... | ... |

pop: set mask, **goto else**
(PC, mask taken from stack)

divergence stack

```
push.stack out
tbranch.eqz vf0, else
vc = vop1
pop.stack
else:
vc = vop2
pop.stack
out:
```

state: thread mask, *divergence stack*

divergence stack

```
push.stack out
tbranch.eqz vf0, else
vc = vop1
pop.stack
else:
vc = vop2
pop.stack
out:
```

state: thread mask, *divergence stack*

Case 2: only else

divergence stack

| PC | thread mask |
|-----|--------------|
| out | originalMask |
| ... | ... |

tbranch: just goto else
(set mask using vf0)

divergence stack

```
push.stack out
tbranch.eqz vf0, else
vc = vop1
pop.stack
else:
vc = vop2
pop.stack
out:
```

state: thread mask, *divergence stack*

Case 2: only else

divergence stack

| PC | thread mask |
|-----|--------------|
| out | originalMask |
| ... | ... |

pop: reset mask, got out
(PC, mask taken from stack)

software divergence management

do everything using predication

compiler must track multiple mask registers

more instructions unless compiler predicts branch

(less if it does)

trickiness in software divergence

loops: mask of un-exited CUDA thread

loop actually executed maximum iterations times

results

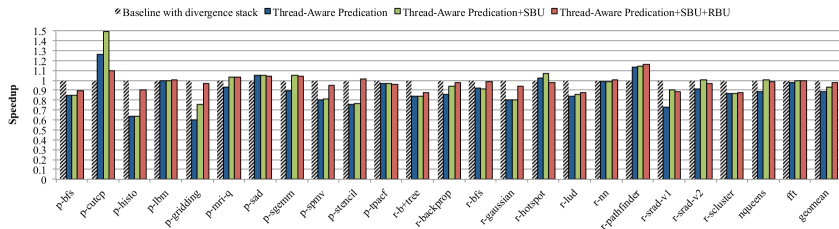


Figure 7. Speedup of thread-aware prediction against divergence stack on NVIDIA Tesla K20c. SBU=Static Branch-Uniformity optimization. RBU=Runtime Branch-Uniformity optimization.

paper's future work

more compiler improvements

branch if any instruction

profile-guided optimization

next time: FPGAs

this topic: vector accelerators

next two lectures — more accelerators

FPGAs — reconfigurable hardware

“configuration” not “instructions”

later: fully custom chips