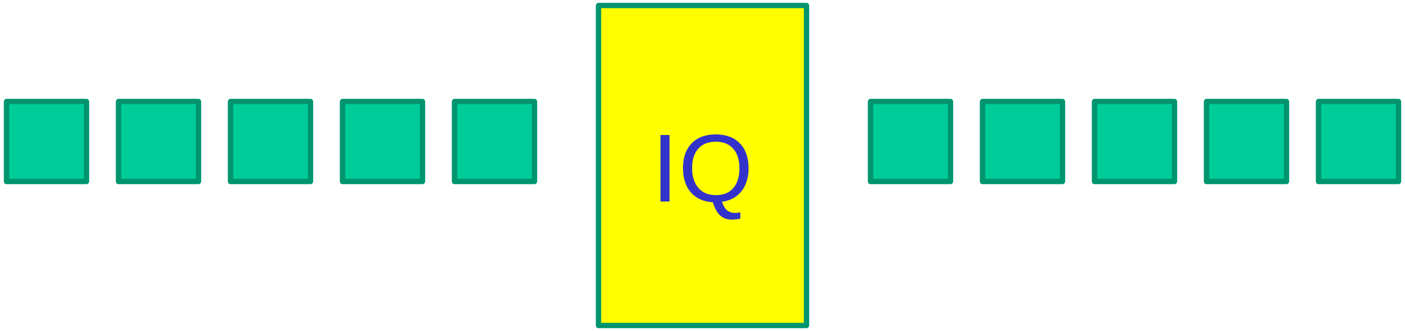
Lecture 11: ILP Innovations and SMT

- Today: out-of-order example, ILP innovations, SMT (Sections 3.9-3.10 and supplementary notes)

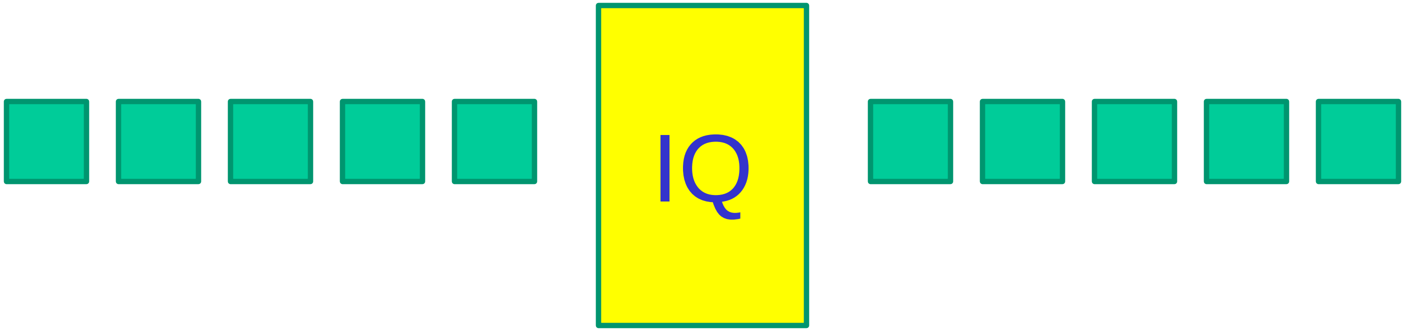
- HW4 due on Tuesday

OOO Example



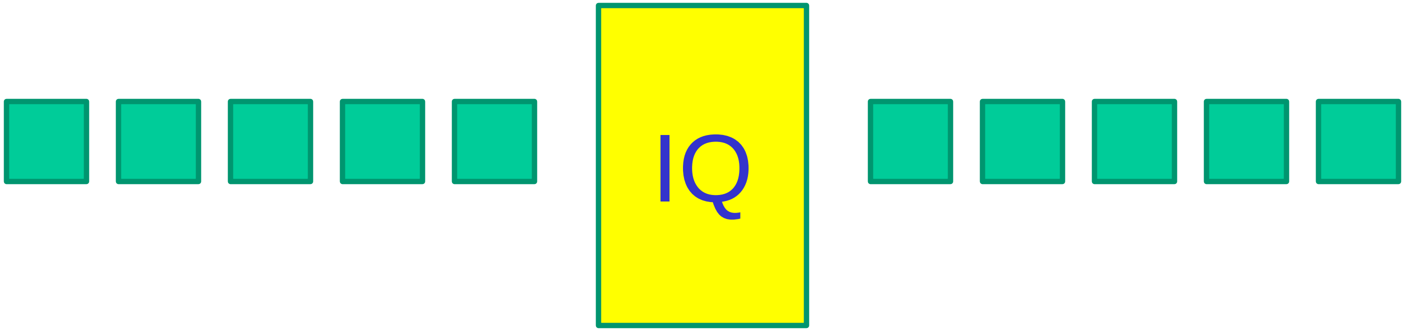
- Assumptions same as HW 4, except there are 36 physical registers and 32 logical registers, and width is 4
- Estimate the issue time, completion time, and commit time for the sample code

Assumptions



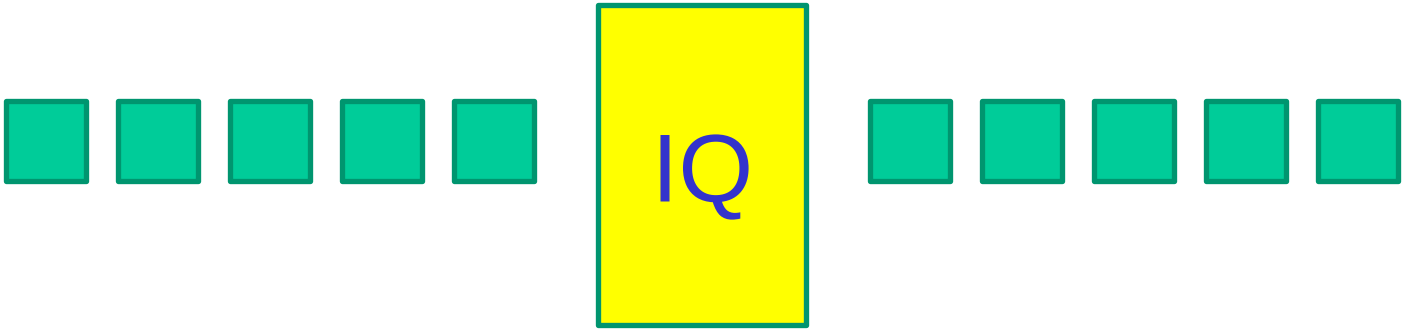
- Perfect branch prediction, instruction fetch, caches
- ADD → dep has no stall; LD → dep has one stall
- An instr is placed in the IQ at the end of its 5th stage, an instr takes 5 more stages after leaving the IQ (ld/st instrs take 6 more stages after leaving the IQ)

OOO Example



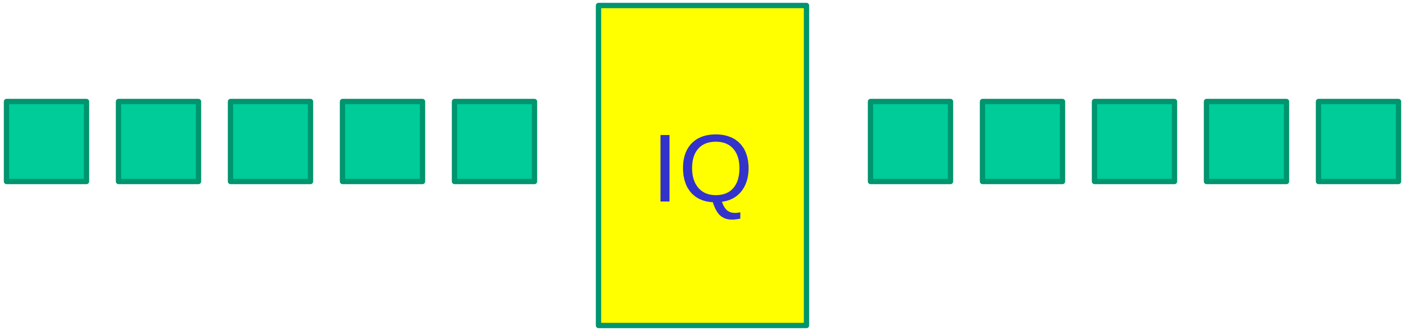
Original code		Renamed code
ADD R1, R2, R3		ADD P33, P2, P3
LD R2, 8(R1)		LD P34, 8(P33)
ADD R2, R2, 8		ADD P35, P34, 8
ST R1, (R3)		ST P33, (P3)
SUB R1, R1, R5		SUB P36, P33, P5
LD R1, 8(R2)	Must wait	
ADD R1, R1, R2		

OOO Example



Original code	Renamed code	InQ	Iss	Comp	Comm
ADD R1, R2, R3	ADD P33, P2, P3	i	i+1	i+6	i+6
LD R2, 8(R1)	LD P34, 8(P33)	i	i+2	i+8	i+8
ADD R2, R2, 8	ADD P35, P34, 8	i	i+4	i+9	i+9
ST R1, (R3)	ST P33, (P3)	i	i+2	i+8	i+9
SUB R1, R1, R5	SUB P36, P33, P5	i+1	i+2	i+7	i+9
LD R1, 8(R2)					
ADD R1, R1, R2					

OOO Example



Original code	Renamed code	InQ	Iss	Comp	Comm
ADD R1, R2, R3	ADD P33, P2, P3	i	i+1	i+6	i+6
LD R2, 8(R1)	LD P34, 8(P33)	i	i+2	i+8	i+8
ADD R2, R2, 8	ADD P35, P34, 8	i	i+4	i+9	i+9
ST R1, (R3)	ST P33, (P3)	i	i+2	i+8	i+9
SUB R1, R1, R5	SUB P36, P33, P5	i+1	i+2	i+7	i+9
LD R1, 8(R2)	LD P1, 8(P35)	i+7	i+8	i+14	i+14
ADD R1, R1, R2	ADD P2, P1, P35	i+9	i+10	i+15	i+15

Reducing Stalls in Rename/Regfile

- Larger ROB/register file/issue queue

- Runahead: while a long instruction waits, let a thread run ahead to prefetch (this thread can deallocate resources more aggressively than a processor supporting precise execution)

- Two-level register files: values being kept around in the register file for precise exceptions can be moved to 2nd level

Stalls in Issue Queue

- Two-level issue queues: 2nd level contains instructions that are less likely to be woken up in the near future

- Value prediction: tries to circumvent RAW hazards

- Memory dependence prediction: allows a load to execute even if there are prior stores with unresolved addresses

- Load hit prediction: instructions are scheduled early, assuming that the load will hit in cache

Functional Units

- Clustering: allows quick bypass among a small group of functional units; FUs can also be associated with a subset of the register file and issue queue

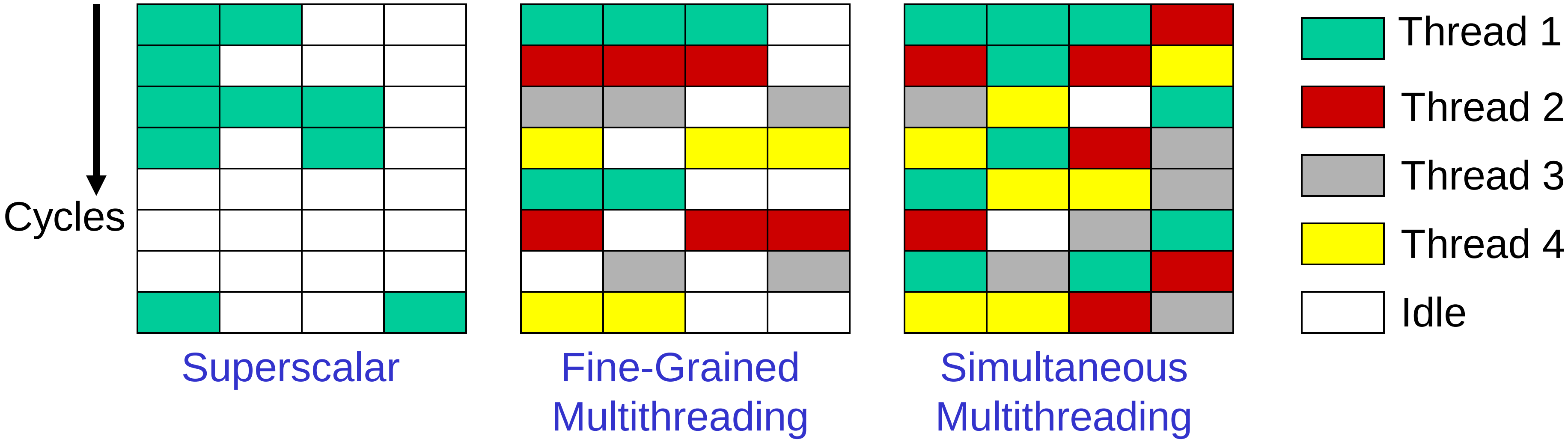
Thread-Level Parallelism

- Motivation:
  - a single thread leaves a processor under-utilized for most of the time
  - by doubling processor area, single thread performance barely improves

- Strategies for thread-level parallelism:
  - multiple threads share the same large processor → reduces under-utilization, efficient resource allocation Simultaneous Multi-Threading (SMT)
  - each thread executes on its own mini processor → simple design, low interference between threads Chip Multi-Processing (CMP) or multi-core

How are Resources Shared?

Each box represents an issue slot for a functional unit. Peak thruput is 4 IPC.



Superscalar

🟩	🟩	⬜	⬜
🟩	⬜	⬜	⬜
🟩	🟩	🟩	⬜
🟩	⬜	🟩	⬜
⬜	⬜	⬜	⬜
⬜	⬜	⬜	⬜
⬜	⬜	⬜	⬜
🟩	⬜	⬜	🟩

Fine-Grained Multithreading

🟩	🟩	🟩	⬜
🟥	🟥	🟥	⬜
▨	▨	⬜	▨
🟨	⬜	🟨	🟨
🟩	🟩	⬜	⬜
🟥	⬜	🟥	🟥
⬜	▨	⬜	▨
🟨	🟨	⬜	⬜

Simultaneous Multithreading

🟩	🟩	🟩	🟥
🟥	🟩	🟥	🟨
▨	🟨	⬜	🟩
🟨	🟩	🟥	▨
🟩	🟨	🟨	▨
🟥	⬜	▨	🟩
🟩	▨	🟩	🟥
🟨	🟨	🟥	▨

- Superscalar processor has high under-utilization – not enough work every cycle, especially when there is a cache miss
- Fine-grained multithreading can only issue instructions from a single thread in a cycle – can not find max work every cycle, but cache misses can be tolerated
- Simultaneous multithreading can issue instructions from any thread every cycle – has the highest probability of finding work for every issue slot

What Resources are Shared?

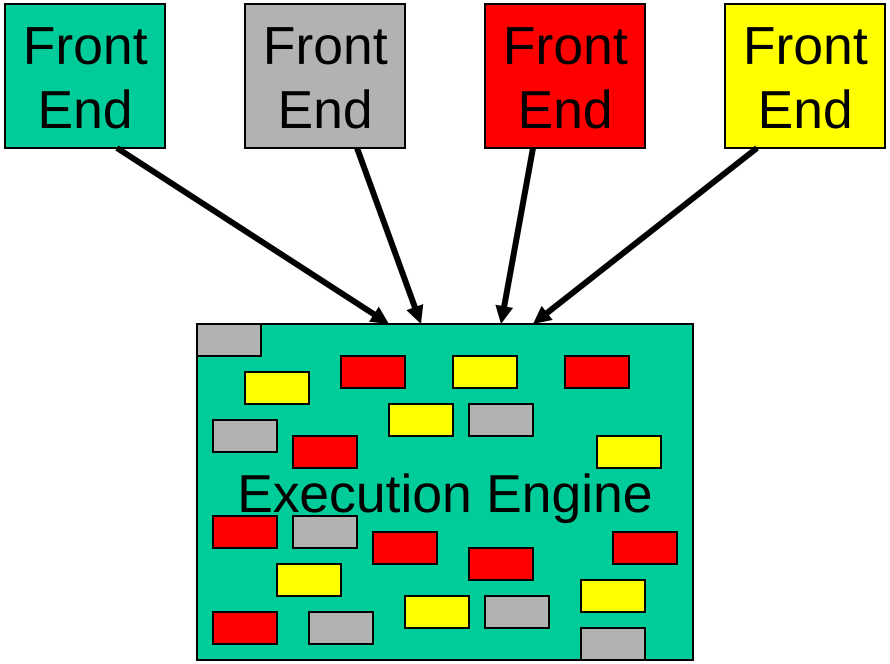
- Multiple threads are simultaneously active (in other words, a new thread can start without a context switch)

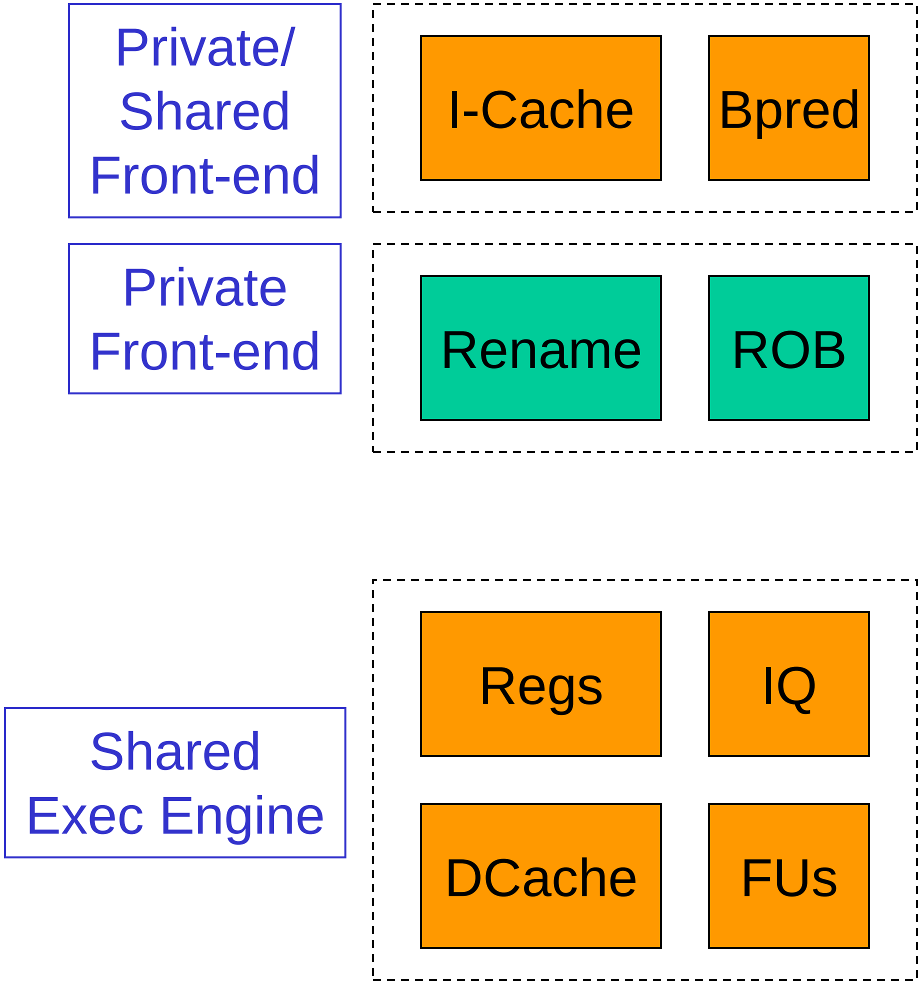
- For correctness, each thread needs its own PC, IFQ, logical regs (and its own mappings from logical to phys regs)

- For performance, each thread could have its own ROB/LSQ (so that a stall in one thread does not stall commit in other threads), I-cache, branch predictor, D-cache, etc. (for low interference), although note that more sharing → better utilization of resources

- Each additional thread costs a PC, IFQ, rename tables, and ROB – cheap!

Pipeline Structure





Private/ Shared Front-end

I-Cache

Bpred

Private Front-end

Rename

ROB

Shared Exec Engine

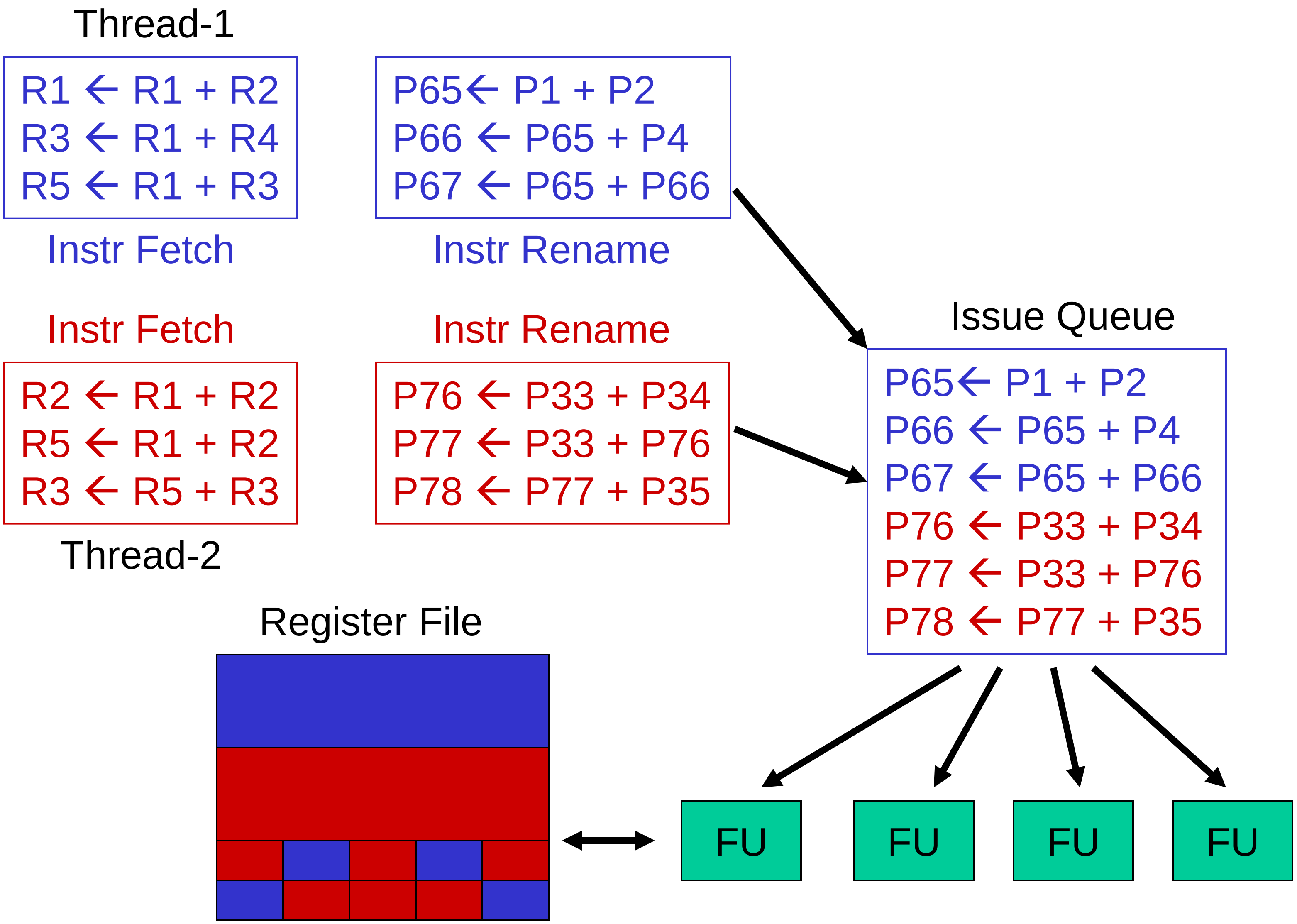
Regs

IQ

DCache

FUs

Resource Sharing



Thread-1

R1 ← R1 + R2
R3 ← R1 + R4
R5 ← R1 + R3

Instr Fetch

P65← P1 + P2
P66 ← P65 + P4
P67 ← P65 + P66

Instr Rename

Instr Fetch

R2 ← R1 + R2
R5 ← R1 + R2
R3 ← R5 + R3

Thread-2

Instr Rename

P76 ← P33 + P34
P77 ← P33 + P76
P78 ← P77 + P35

Issue Queue

P65← P1 + P2
P66 ← P65 + P4
P67 ← P65 + P66
P76 ← P33 + P34
P77 ← P33 + P76
P78 ← P77 + P35

Register File

FU
FU
FU
FU

Performance Implications of SMT

- Single thread performance is likely to go down (caches, branch predictors, registers, etc. are shared) – this effect can be mitigated by trying to prioritize one thread

- While fetching instructions, thread priority can dramatically influence total throughput – a widely accepted heuristic (ICOUNT): fetch such that each thread has an equal share of processor resources

- With eight threads in a processor with many resources, SMT yields throughput improvements of roughly 2-4

Pentium4 Hyper-Threading

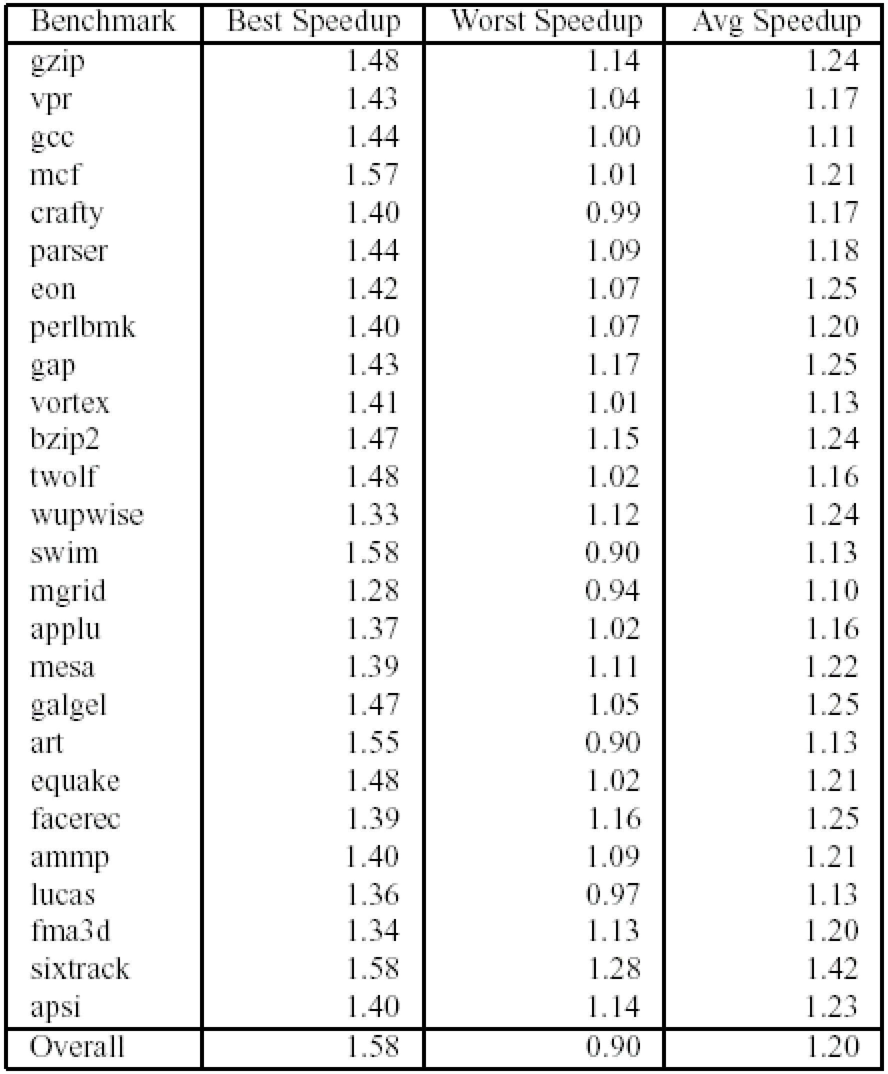
- Two threads – the Linux operating system operates as if it is executing on a two-processor system

- When there is only one available thread, it behaves like a regular single-threaded superscalar processor

- Statically divided resources: ROB, LSQ, issueq -- a slow thread will not cripple thruput (might not scale)

- Dynamically shared: trace cache and decode (fine-grained multi-threaded, round-robin), FUs, data cache, bpred

Multi-Programmed Speedup



Benchmark	Best Speedup	Worst Speedup	Avg Speedup
gzip	1.48	1.14	1.24
vpr	1.43	1.04	1.17
gcc	1.44	1.00	1.11
mcf	1.57	1.01	1.21
crafty	1.40	0.99	1.17
parser	1.44	1.09	1.18
eon	1.42	1.07	1.25
perlbmk	1.40	1.07	1.20
gap	1.43	1.17	1.25
vortex	1.41	1.01	1.13
bzip2	1.47	1.15	1.24
twolf	1.48	1.02	1.16
wupwise	1.33	1.12	1.24
swim	1.58	0.90	1.13
mgrid	1.28	0.94	1.10
applu	1.37	1.02	1.16
mesa	1.39	1.11	1.22
galgel	1.47	1.05	1.25
art	1.55	0.90	1.13
equake	1.48	1.02	1.21
facerec	1.39	1.16	1.25
ammp	1.40	1.09	1.21
lucas	1.36	0.97	1.13
fma3d	1.34	1.13	1.20
sixtrack	1.58	1.28	1.42
apsi	1.40	1.14	1.23
Overall	1.58	0.90	1.20

- sixtrack and eon do not degrade their partners (small working sets?)
- swim and art degrade their partners (cache contention?)
- Best combination: swim & sixtrack worst combination: swim & art
- Static partitioning ensures low interference – worst slowdown is 0.9

Title

- Bullet