Exam Review 2

ROB: head/tail

PC	log. reg	prev. phys.	store?	except?	ready?	
A	R3	X3	no	none	yes	← old tail
B	R1	X1	no	none	yes	← tail
C	R1	X6	no	none	yes	
D	R4	X4	no	none	yes	
E	---	---	yes	none	yes	
F	---	---	no	none	yes	
A	R3	X5	no	none	yes	
B	R1	X7	no	fault	yes	
C	R1	X10	no	none	no	
D	R4	X8	no	none	yes	← head
---	---	---	---	---	---	← next entry

rename map
(for next rename)

log.	phys.
R0	X0
R1	X1
R2	X11
R3	X9
R4	X12

free list:
X11, X3

exercise: result of processing rest?

Questions?

vector instructions

register types: scalar, vector, predicate/mask, length

made-up syntax follows:

```
@MaskRegister V0 ← V1 + V2,
@MaskRegister VADD V0, V1, V2
for (int i = 0;
     i < MIN(VectorLengthRegister,
             MaxVectorLength);
     i += 1) {
    if (MaskRegister[i]) {
       V0[i] = V1[i] + V2[i];
    }
}
```

vector exercise

```
void vector_add_one(int *x, int length) {
    for (int i = 0; i < length; ++i) {
        x[i] += 1;
    }
}
```

exercise: write as a vector machine program with 64-element vectors

vector length register or predicate (mask) registers

vector exercise answer

```
void vector_add_one(int *x, int length) {
    for (int i = 0; i < length; ++i) {
        x[i] += 1;
    }
}

// R1 contains X, R2 contains length
      VL ← R2 MOD 64
Loop: IF R2 <= 0, goto End
      V1 ← MEMORY[R1]
      V1 ← V1 + 1
      MEMORY[R1] ← V1
      R2 ← R2 − VL
      VL ← 64
      goto Loop
End:
```


relaxed memory models ex 1

reasons for reorderings?

relaxed reasons

optimizations to think about:

- executing loads/stores out-of-order (if addresses don't conflict)
- combining two loads for same address ("load forwarding")
- combining load + store for same address ("store forwarding")
- not waiting for invalidations to be acknowledged (esp. non-bus network)

relaxed memory models ex 2

What can happen?

```
X = Y = 0
CPU1:        CPU2:
R1 ← Y       R4 ← X
X ← 1        X ← 2
R2 ← Y       Y ← 2
R3 ← X
```

examples of possible sequential orders? (there are 8)

examples of non-sequential orders?

what could happen to cause other orders?

possible sequential orders

```
X = Y = 0
CPU1:        CPU2:
R1 ← Y       R4 ← X
X ← 1        X ← 2
R2 ← Y       Y ← 2
R3 ← X
```

R1	R2	R3	R4
0	0	1	0
0	0	1	1
0	0	2	0
0	0	2	1
0	2	1	0
0	2	2	0
0	2	2	1
2	2	1	0

non-seq orders

```
X = Y = 0
CPU1:        CPU2:
R1 ← Y       R4 ← X
X ← 1        X ← 2
R2 ← Y       Y ← 2
R3 ← X
```

R2 = 2 and R3 = 1 and R4 = 1

- example cause: store forwarding (use stored value in X)
- example cause: load forwarding (reuse first load)

R1 = 2 and R3 = 2

- example cause: reordered stores in CPU2
- example cause: CPU2 doesn't wait for CPU1 invalidate

(HW) transactional memory

what is a transaction?

limitations?

when is performance good/bad?

(HW) transactional memory

what is a transaction?

- atomic — as if uninterrupted by other things

limitations?

when is performance good/bad?

(HW) transactional memory

what is a transaction?

- atomic — as if uninterrupted by other things

limitations?

- I/O
- amount of space to store "transaction log"

when is performance good/bad?

(HW) transactional memory

what is a transaction?

- atomic — as if uninterrupted by other things

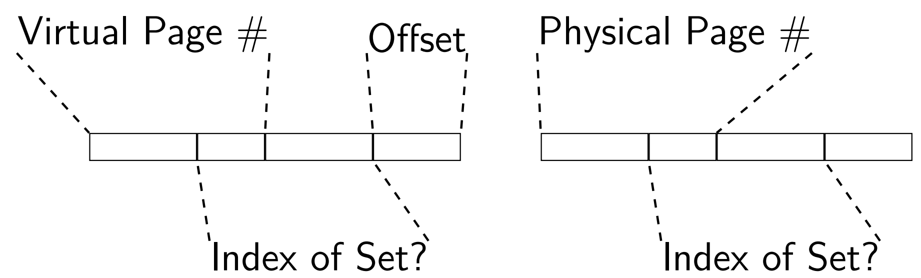
limitations?

- I/O
- amount of space to store "transaction log"

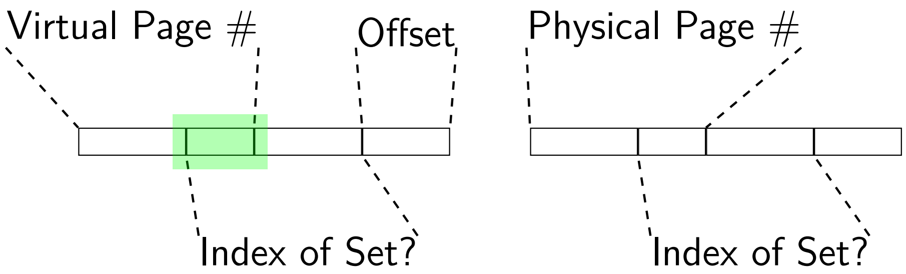
when is performance good/bad?

- livelock — transcations abort each other over and over?
- possibly more "wasted work" if contention (e.g. short transaction aborts long one)
- fairness?
- overhead to manipulate transaction log if lots of items?

Virtual and Physical

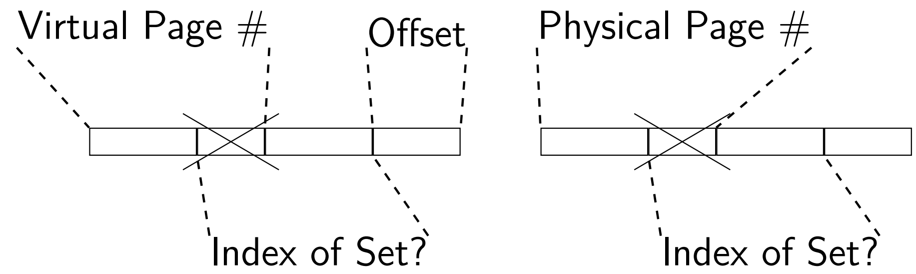


Virtual and Physical



Cache has virtual indexes?

Virtual and Physical



Cache has virtual indexes?

Solution #1: Disallow overlap

Physically Tagged, Virtually Indexed

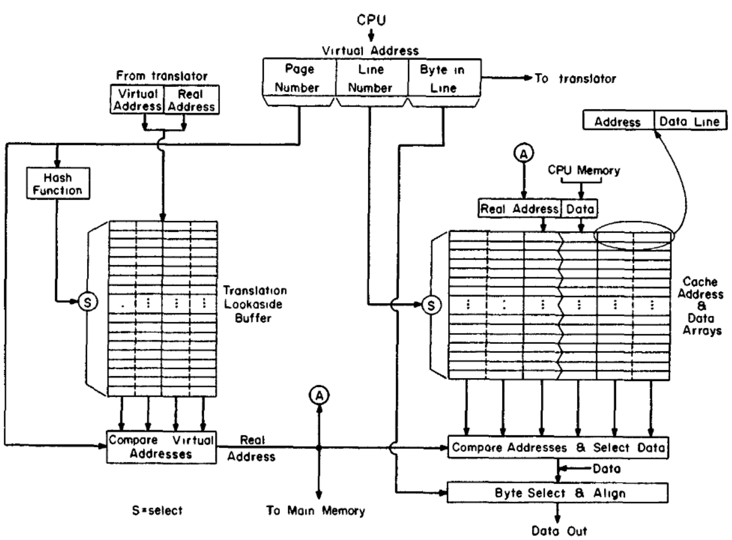
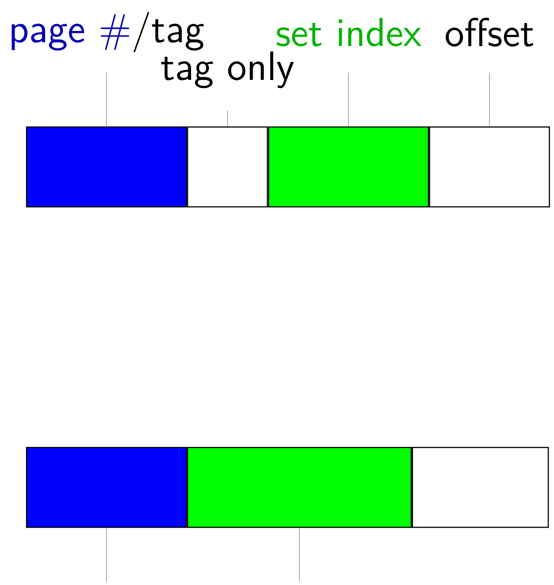
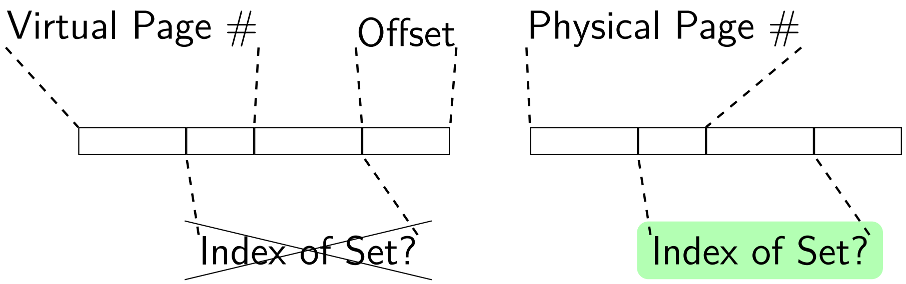


Figure 2. A typical cache and TLB design.

Plausible splits



Virtual and Physical

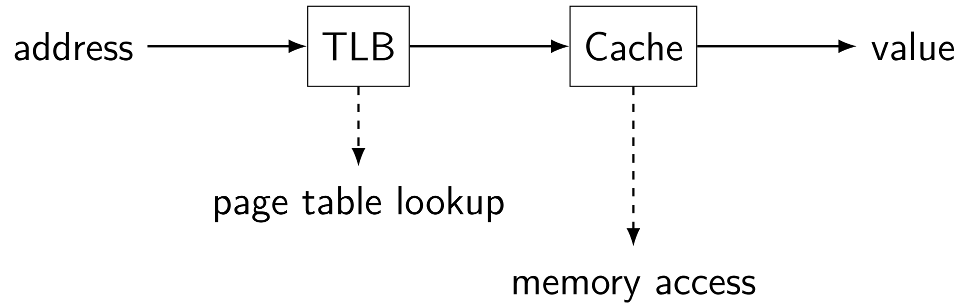


Cache has virtual indexes?

Solution #1: Disallow overlap

Solution #2: Translate first

Translate First



Virtual Caches

no translation for entire cache lookup
- including tag checking

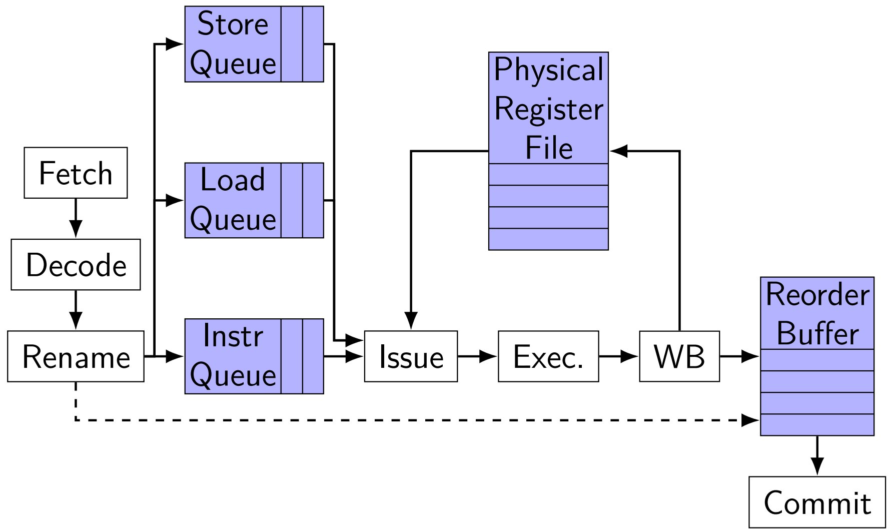
exist, but more complicated

need to handle aliasing
- multiple virtual addresses for one physical

example ways:
- OS must prevent/manage aliasing
- physical L2 tracks virtual to physical mappping in L1

OOO tradeoffs

gem5 pipeline



OOO tradeoffs (1)

dependencies plus latency limits performance
- diminishing returns from additional computational resources

latencies that can be especially long:
- cache/memory accesses
- branch resolution

speculation helps "cheat" on dependencies
- branch prediction
- memory reordering (+ check if addresses conflict later)

OOO tradeoffs (2)

limits on number of instructions "in flight"

number of physical registers

size of queues (instruction, load/store)

size of reorder buffer

active cache misses

OOO tradeoffs (3)

miscellaneous issues:

right types of functional units for programs?

wasted work from frequent "exceptions"?
- might include, e.g., memory ordering error

OOO tradeoff exercise

what programs will be most affected by a smaller/larger:

- reorder buffer
- instruction queue
- number of floating point adders
- number of physical registers
- number of instructions fetched/decoded/renamed/issued/committed per cycle

VLIW

fetch instruction **bundles**

parallel pipelines, shared registers

specialized pipelines

Longer instruction word pipeline

Fetch	Read Regs	Simple ALU	—	Write Back
	Read Regs	Address ALU	Memory	Write Back
	Read Regs	Int/Mul ALU 1	Int/Mul ALU 2	Write Back

VLIW vs OOO

VLIW is like OOO but...

instructions are scheduled at compile-time, not run-time

eliminates OOO scheduling logic/queues

compiler does dependency detection

- including dealing with functional unit latency

possibly eliminates reorder buffer

VLIW problems

requires smart compiler

can't reschedule based on memory latency, etc.

assembly/machine code tied to particular HW design

VLIW exercise

```
int *foo; int *bar;
...
for (int i = 0; i < 1000; ++i) {
    *foo = *foo * *bar;
    foo += 1;
    bar += 1;
}
```

ouline what assembly for a VLIW processor with:

- bundles of two instructions:
- 1: load/store (address is reg+offset) or add/subtract
- 2: compare-and-branch or multiply or add/subtract
- all instructions take two cycles to produce usable result
- all instructions take registers or constants
- adds can load a constant

VLIW exercise: slow answer

```
# R0: FOO; R1: BAR; R2: I
# R3: FOO temp1, R4: BAR temp1
R2 ← 0          .  NOP
Loop:
NOP             .  IF R1 < 1000 GOTO End
R3 ← M[R0+0]    .  R2 ← R2 + 1  // foo . ++i
R4 ← M[R1+0]    .  R1 ← R1 + 4  // bar . ++bar
NOP             .  R0 ← R0 + 4  //     . ++foo
NOP             .  R3 ← R3 × R4 //     . ×
NOP             .  NOP          // wait for ×
M[R1−4] ← R3    .  NOP          // foo .
NOP             .  GOTO Loop
End:
```


VLIW exercise: faster answer?

needed nops due to instruction delays/lack of work

alternative:

unroll loop several times

move loads/stores between iterations of the loop

eliminate branch at beginning

final notes

a bunch of multiple choice (because I could write it)

have room until 7:15PM — will give 2 hours

office hours Friday 10am–12pm / Piazza

super last minute questions? office hours Monday 1pm–3pm