CS 6354: Branch Prediction (con't) / Multiple Issue

14 September 2016

Last time: forwarding/stalls

```
add $a0, $a2, $a3
; zero or more instructions
sub $t0, $a0, $a1
```

sub depends on calcuation from add

No forwarding: get `$a0` via register file
- 'write back' of add completes before 'decode' of sub

Forwarding: transfer values via pipeline registers instead
- 'execute' of add completes before 'execute' of sub

Stall: Delay instruction
- don't start 'execute' of sub before 'execute' of add

Last time: scheduling to avoid stalls

find all dependencies

track required delays between instructions

software pipelining: even across loop iterations
- can be more than loads/stores

unroll loops to have more to fit in delays

Why bimodal: loops

```
for (int i = 0; i < 10000; i += 1) {
    ...
}
```

```
        li $t0, 0 ; i <- 0
loop:   ...
        addiu $t0, $t0, 1
        blt $t0, 10000, loop
```

Why bimodal: non-loops

```
char *data = malloc(...);
if (!data) handleOutOfMemory();
```

Why more than 1-bit?

```
for (int j = 0; j < 10000; ++j) {
    for (int i = 0; i < 4; ++i) {
        ...
    }
}
```

iteration	last taken	prediction	correct?
0	—	—	—
1	yes	taken	yes
2	yes	taken	yes
3	yes	taken	yes
4	yes	taken	no
5	no	not taken	no
6	yes	taken	yes
...	...	...	...

Saturating counters (1)

```
for (int j = 0; j < 10000; ++j) {
    for (int i = 0; i < 4; ++i) {
        ...
    }
}
```

iteration	counter before	prediction	correct?
0	0?	—	—
1	1	taken	yes
2	2 (MAX)	taken	yes
3	2 (MAX)	taken	yes
4	2 (MAX)	taken	no
5	1	taken	yes
6	2 (MAX)	taken	yes
...	...	...	...

Saturating counters (2)

```
for (int j = 0; j < 10000; ++j) {
    for (int i = 0;; ++i) {
        ...
        if (i == 3) break;
    }
}
```

iteration	counter before	prediction	correct?
0	0?	—	—
1	-1 (MIN)	not taken	yes
2	-1 (MIN)	not taken	yes
3	-1 (MIN)	not taken	yes
4	0	not taken	no
5	-1 (MIN)	not taken	yes
6	-1 (MIN)	not taken	yes
...	...	...	...

Local history: loops

```
for (int j = 0; j < 10000; ++j) {
    for (int i = 0; i < 4; ++i) {
    }
}
```

observation: taken/not-taken pattern is NNNTNNNTNNNT…

construct table:

prior five results	prediction
TNNNT	N
NTNNN	T
NNTNN	N
NNNTN	N

Local history predictor

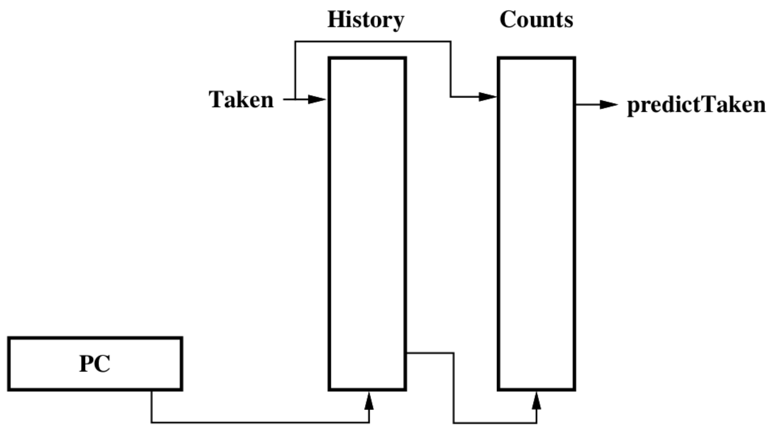


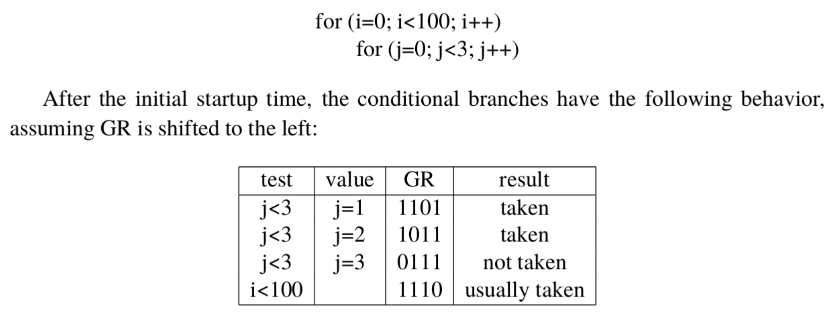
Figure 4: Local History Predictor Structure

Global history

```
    if (x >= 0) ...
    if (x <= 0) ...
```

result for prior branches	prediction
…N	T
…T	N

Global history identifes branches



```
for (i=0; i<100; i++)
    for (j=0; j<3; j++)
```

After the initial startup time, the conditional branches have the following behavior, assuming GR is shifted to the left:

test	value	GR	result
j<3	j=1	1101	taken
j<3	j=2	1011	taken
j<3	j=3	0111	not taken
i<100		1110	usually taken

Global history predictor

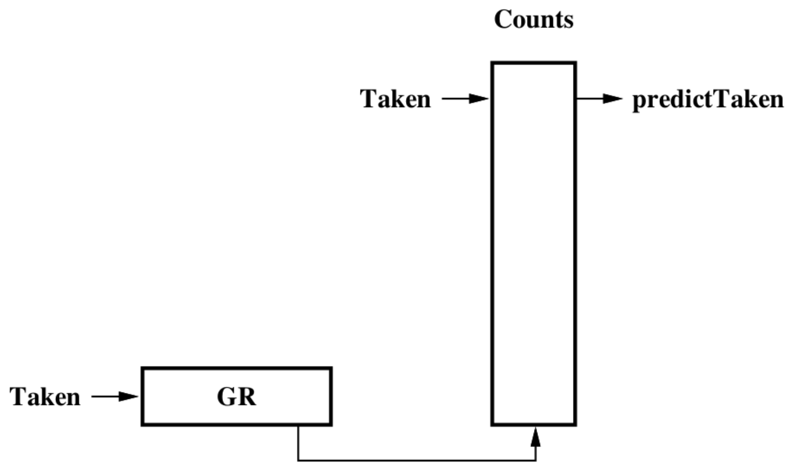


Figure 6: Global History Predictor Structure

Combining local and global

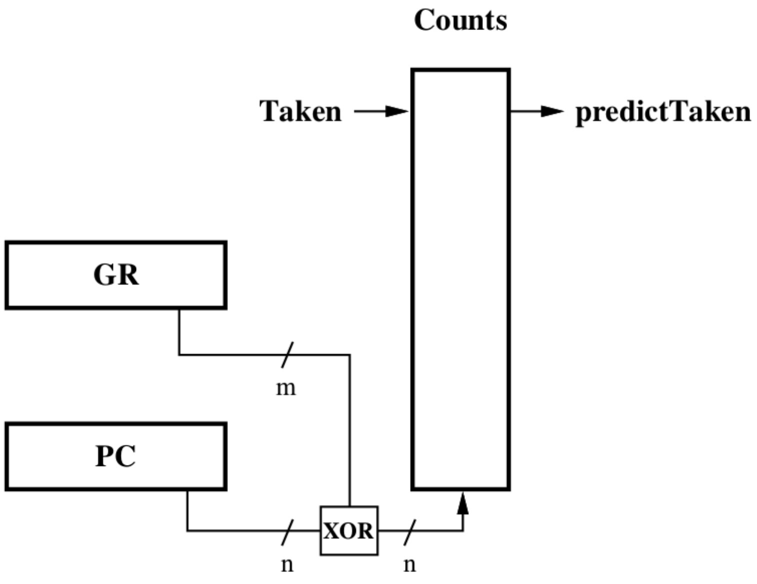


Figure 10: Global History Predictor with Index Sharing

Combining generally

branch predictor predictor

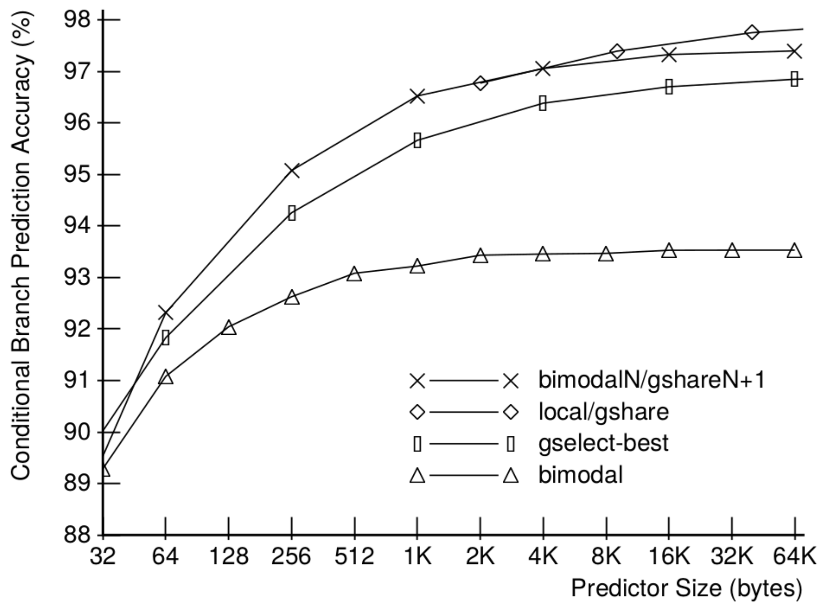
counter per branch:

- increment when first predictor is better
- decrement when first predictor is worse

use first predictor if non-negative

2-bit saturating — predictor gets ‘second chance’

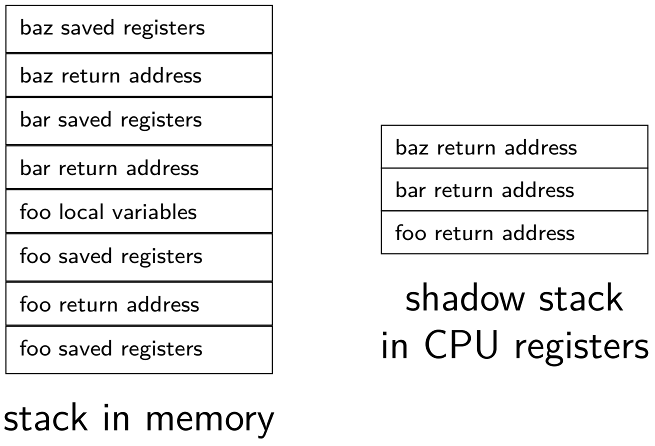
The experiments



Return address stack

Predicting function call return addresses

Solution: stack in processor registers



Speculation: Loop termination prediction

Predicting loops with fixed iteration counts

Solution: record last iteration count

(times since change of branch result)

address	last # iters	current #
0x040102	128	98
...	...	...

loop count table

```
for (int i = 0;
     i < 128;
     ++i) {
    ...
}
```

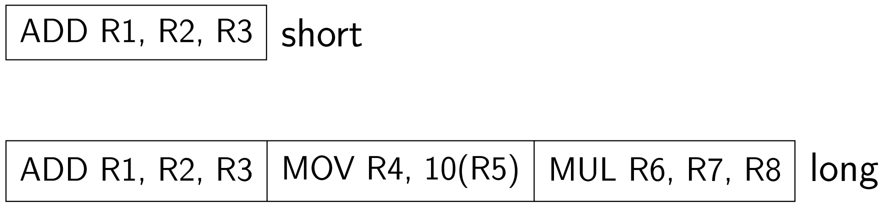
Speculation: More

register value prediction

will two loads/stores alias each other?

...

Very Long Instruction Word



bundle of instructions
issued and execute together

VLIW Pipeline

Normal RISC-like pipeline

Fetch	Read Regs	Execute (ALU)	Memory	Write Back

$\xrightarrow{\qquad\qquad}$ time

Longer instruction word pipeline

Fetch	Read Regs	Execute (ALU)	Memory	Write Back
	Read Regs	Execute (ALU)	Memory	Write Back
	Read Regs	Execute (ALU)	Memory	Write Back

VLIW Pipeline

Normal RISC-like pipeline

Fetch	Read Regs	Execute (ALU)	Memory	Write Back

$\xrightarrow{\qquad\qquad}$ time

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

Itanium

VLIW-derived processor

Called EPIC — tries to address some shortcomings of VLIW

Intel designed ISA, introduced c. 2001

"Bundles" of 3 instructions:

1. Slot 1 — Usually Memory or Integer
2. Slot 2 — Usually Memory or Integer or Floating Point
3. Slot 3 — Usually Integer or Floating Point or Branch

Example assembly:

```
{ .mmf ; Bundle of Memory/Memory/Float
LDFD f83 = [r35], r21   ; f83 <— MEM[r35+r21]
LDFD f89 = [r16], r21   ; f89 <— MEM[r16+r21]
FMA f11 = f43, f91, f11 ; f11 <— f43 * f91 + f11
}
```

ELI-512

Bundles of 24+ 'instructions':

- 8 32-bit integer operations
- 8 64-bit integer/floating point operations
- 8 memory accesses
- 32 register accesses
- 1 very fancy conditional jump
- ?? register-register movements

Don't want, e.g., a memory access?

- put a no-op in that slot

ELI-512

Bundles of 24+ ‘instructions’:

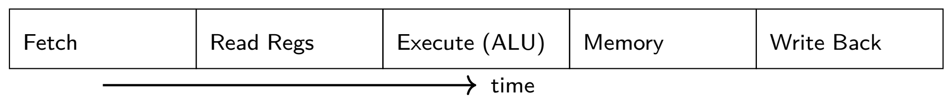
- 8 32-bit integer operations
- 8 64-bit integer/floating point operations
- 8 memory accesses
- 32 register accesses
- 1 very fancy conditional jump
- ?? register-register movements

Don’t want, e.g., a memory access?

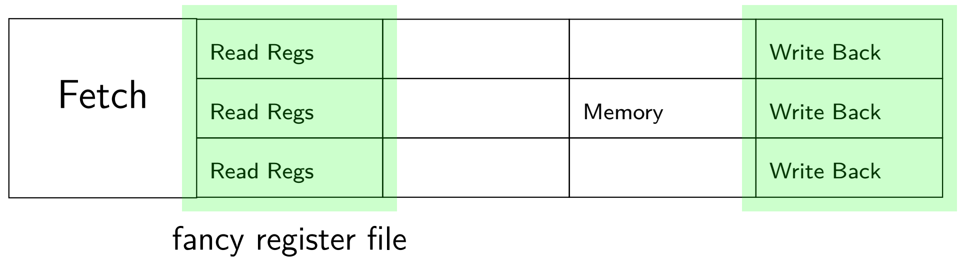
- put a no-op in that slot

VLIW Pipeline

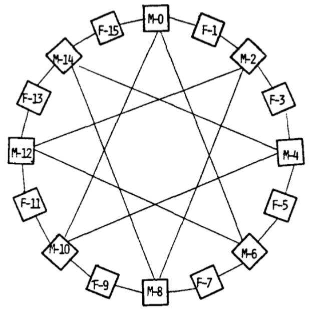
Normal RISC-like pipeline



Longer instruction word pipeline



ELI-512: Multiple Register Banks



Global Interconnection Scheme of the ELI-512

16 modules

each has own registers

explicitly move values between modules

ELI-512

Bundles of 24+ ‘instructions’:

- 8 32-bit integer operations
- 8 64-bit integer/floating point operations
- 8 memory accesses
- 32 register accesses
- 1 very fancy conditional jump
- ?? register-register movements

Don’t want, e.g., a memory access?

- put a no-op in that slot

VLIW Pipeline

Normal RISC-like pipeline

Fetch	Read Regs	Execute (ALU)	Memory	Write Back

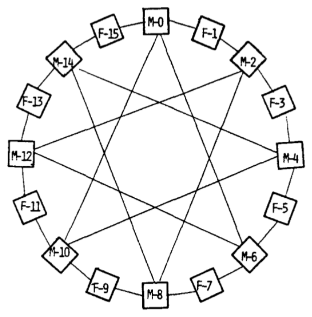
$\longrightarrow$ time

Longer instruction word pipeline

Fetch	Read Regs			Write Back
	Read Regs		Memory	Write Back
	Read Regs			Write Back

fancy cache??

ELI-512: Multiple Memory Banks



Global Interconnection Scheme of the ELI-512

16 modules

each M module has own memory

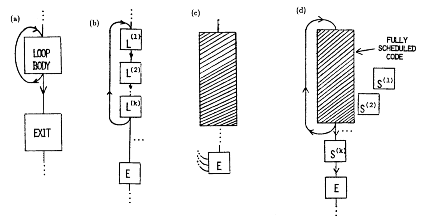
explicitly choose which module to use

Compiler challenges

need 24+ indepedent instructions to fill bundle

not found in natural code

Solution for loops

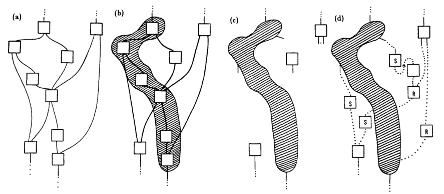


Unroll it!

How do we know this is safe (e.g. no array overlap)?

Compiler does fancy equation solving
Doesn't work? Can't generate good code.

Solution for non-loops



Guess most common branches

Generate that code

Then generate compensation code for wrong guesses

Loop unroling for VLIW

original code

```
for (i = 0; i < 15; i += 1) {
    a[i] *= 2;
}
```

unroll x 3

```
for (i = 0; i < 15; i += 3) {
    a[i+0] *= 2;
    a[i+1] *= 2;
    a[i+2] *= 2;
}
```

unroll x 3 + schedule

```
loop:
/* bundle 1: */
    a2 = a[i+2];
    a0 *= 2;  // loaded last iter
    a1 *= 2;
/* bundle 2: */
    a[i+0] = a0;
    a[i+1] = a1;
    nextI = i + 3;
/* bundle 3: */
    a0 = a[nextI+0]; // load for next i
    a1 = a[nextI+1];
    a2 *= 2;
/* bundle 4: */
    a[i+2] = a2;
    i = nextI;
    if (nextI < 15) goto loop;
```

Trace scheduling: Interlocks?

ELI-512 and TRACE had no “interlocks”

no forwarding — longer delays

wrong answer if compiler doesn’t schedule properly

Forwarding logic would be too complex/slow

Trace scheduling

original code

```
if (x > 0) {
    a += 1;
    if (y > 0) {
        b *= c;
        d -= e;
    }
} else {
    a -= 1;
}
```

guess x > 0 and y > 0

```
/* common case: */
a += 1;
newB = b * c;
newD = d - e;
/* compensation code: */
if (x <= 0) {
    a -= 1;
    newB = b;
    newD = d
} else if (y <= 0) {
    newB = b;
    newD = d;
}
```

1st bundle:
add
multiply
subtract
conditional jump

reason for fancy conditional jump

common case is one cycle

Assisting compilers

many registers

- Itanium: 128 integer, 128 float, 128 condition
- makes unrolling + rescheduling loops easier

conditional instructions

- Itanium: every instruction can be conditional
- e.g. add if condition register true
- avoid expensive branches for short fixup code

Compiler speculation

Trace schedule ≈ compiler branch prediction

Itanium: explicit speculative loads

- `ld.s`: load value only if valid address

Itanium: aliasing detection

- `ld.a`: load value, watch for stores to address
- `chk.a`: branch if that address was written since load

VLIW problems

bet on good compilers

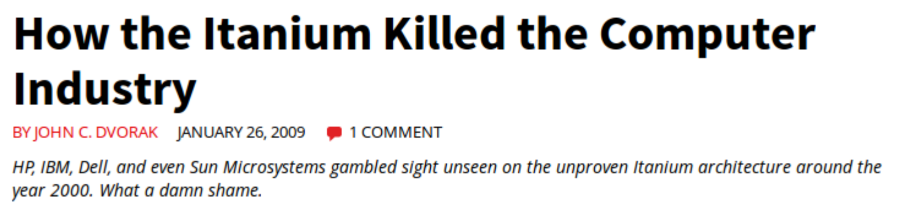
recompile to increase bundle size

- Itanium solution: bundles have 'stop' bit
- assembler sets stop bit if dependency
- otherwise, CPU can start at same time

compilers don't know enough to schedule

- will load use cache or take a long time?
- Itanium solution: prefetch, speculative loads

The Itanium story



How the Itanium Killed the Computer Industry

BY JOHN C. DVORAK JANUARY 26, 2009 1 COMMENT

HP, IBM, Dell, and even Sun Microsystems gambled sight unseen on the unproven Itanium architecture around the year 2000. What a damn shame.

VLIW: Moving forward

Not the winner

Itanium is being phased out

- some minor commerical VLIW architectures
- (e.g. SHARC for digital signal processors)

things like trace scheduling done in hardware

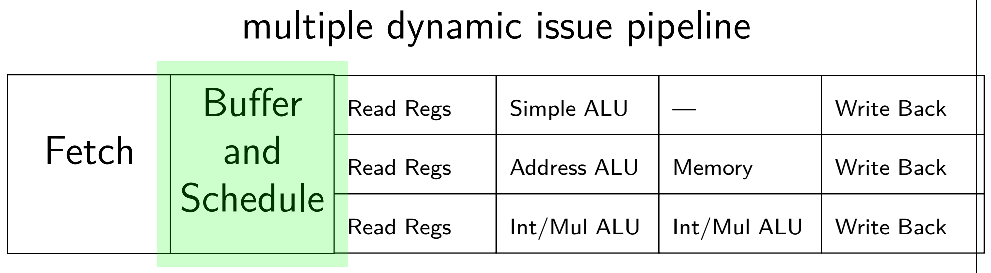
branch prediction ≈ trace scheduling

```
if (x > 0) {
    a += 1; /* common case */
    if (y > 0) {
        b *= c; /* common case */
        d -= e; /* common case */
    }
} else {
    a -= 1;
}
```

good branch predictor runs common case

hardware will undo if wrong

Preview: Dynamic Issue



Next time: Precise interrupts

goal: reschedule (reorder) instructions

But...

- page fault — enter OS to change page table, restart
- I/O interrupt — run OS handler, restart
- timer interrupt — OS save state, restores later

illusion of one-by-one in-order execution

- OS doesn't know about internals of pipelines
- OS saves/restores registers + single current instruction