CS 6354: Static Scheduling / Branch Prediction

12 September 2016

On out-of-order RDTSC

Because of pipelines, etc.:

- RDTSC can actually take its time measurement after it starts
- Earlier instructions can still be completing by the time RDTSC starts

Solutions:

- Time longer things so the overlap isn't significant
- Use synchronizing instructions like CPUID

General notes about accurate timing

take multiple measurements — detect/avoid outliers

make sure instruction caches, etc. are filled — discard first few measurements

prefer longer benchmarks — it's easier to accurately measure microseconds than nanoseconds

Some pitfalls

initialize arrays

- OS may map multiple virtual pages to one physical page of zeroes

if you want something in cache, access it to do that

- allocating it doesn't count

Note about optimizations

```
int test(void) {
    int array[1024 * 1024];
    memset(array, sizeof array, 42);
    return 0;
}
```

generated assembly from `gcc -O`:

```
test:
        movl    $0, %eax
        ret
```

Note about throughput/latency

try to exclude benchmark overhead

throughput should take prefetching, concurrent memory requests, etc. into account

Other homework 1 questions?

Recall: The MIPS pipeline

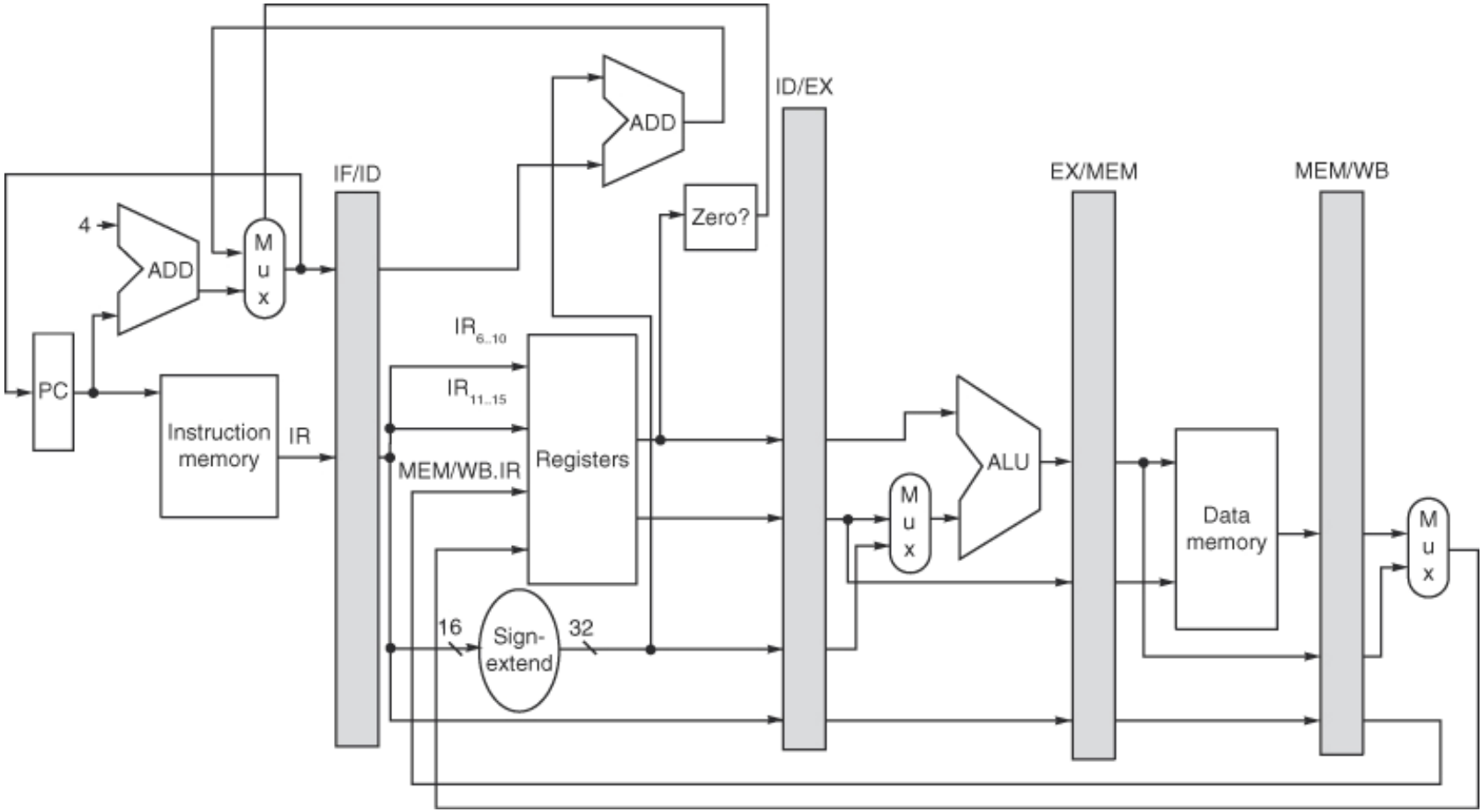


Figure C.28 The stall from branch hazards can be reduced by moving the zero test and branch-target calculation into the ID phase of the pipeline. Notice that we have made two important changes, each of which removes 1 cycle from the 3-cycle stall for branches. The first change is to move both the branch-target address calculation and the branch condition decision to the ID cycle. The second change is to write the PC of the instruction in the IF phase, using either the branch-target address computed during ID or

Figure: H&P Appendix C

Forwarding/Stalls Example

```
memcpy: ; $a0 = destiation, $a1 = source
        ; $a2 = length
      lb $t0, ($a1) ; $t0 <- MEM[$a1]
      sb $t0, ($a0) ; MEM[$a0] <- $t0
      addiu $a0, $a0, 1
      addiu $a1, $a1, 1
      addiu $a2, $a2, -1
      beqz $a2, memcpy
      nop ; DELAY SLOT
```

Quick reference:

IF	fetch instruction
ID	decode (read regs, compute branches)
EX	execute (ALU)
MEM	data memory
WB	write back

Which of these require forwarding or stalls?
Between what pipeline stages?

1. for $t0 between lb and sb
2. for $a0 between sb and addiu
3. for $a1 between lb and addiu
4. for $a2 between addiu and beqz
5. for $a1 between addiu and the following iteration's lb

Forwarding/Stalls Example

```
memcpy: ; $a0 = destiation, $a1 = source
        ; $a2 = length
      lb $t0, ($a1) ; $t0 <- MEM[$a1]
      sb $t0, ($a0) ; MEM[$a0] <- $t0
      addiu $a0, $a0, 1
      addiu $a1, $a1, 1
      addiu $a2, $a2, -1
      beqz $a2, memcpy
      nop ; DELAY SLOT
```

Quick reference:

IF	fetch instruction
ID	decode (read regs, compute branches)
EX	execute (ALU)
MEM	data memory
WB	write back

Which of these require forwarding or stalls?
Between what pipeline stages?

1. for $t0 between lb and sb
2. for $a0 between sb and addiu
3. for $a1 between lb and addiu
4. for $a2 between addiu and beqz
5. for $a1 between addiu and the following iteration's lb

Forwarding/Stalls Example

```
memcpy: ; $a0 = destiation, $a1 = source
        ; $a2 = length
      lb $t0, ($a1) ; $t0 <- MEM[$a1]
      sb $t0, ($a0) ; MEM[$a0] <- $t0
      addiu $a0, $a0, 1
      addiu $a1, $a1, 1
      addiu $a2, $a2, -1
      beqz $a2, memcpy
      nop ; DELAY SLOT
```

Quick reference:

IF	fetch instruction
ID	decode (read regs, compute branches)
EX	execute (ALU)
MEM	data memory
WB	write back

	T=1	T=2	T=3	T=4	T=5	T=6	T=7
IF	sb	addiu	addiu	addiu	beqz	nop	lb
ID	lb	sb	addiu	addiu	addiu	beqz	lb
EX		lb	sb	addiu	addiu	addiu	beqz
MEM			lb	sb	addiu	addiu	addiu
WB				lb	sb	addiu	addiu

Forwarding/Stalls Example

```
memcpy: ; $a0 = destiation, $a1 = source
        ; $a2 = length
      lb $t0, ($a1) ; $t0 <- MEM[$a1]
      sb $t0, ($a0) ; MEM[$a0] <- $t0
      addiu $a0, $a0, 1
      addiu $a1, $a1, 1
      addiu $a2, $a2, -1
      beqz $a2, memcpy
      nop ; DELAY SLOT
```

Quick reference:

IF	fetch instruction
ID	decode (read regs, compute branches)
EX	execute (ALU)
MEM	data memory
WB	write back

	T=1	T=2	T=3	T=4	T=5	T=6	T=7
IF	sb	addiu	addiu	addiu	beqz	nop	lb
ID	lb	sb	addiu	addiu	addiu	beqz	lb
EX		lb	sb	addiu	addiu	addiu	beqz
MEM			lb →	sb	addiu	addiu	addiu
WB				lb	sb	addiu	addiu

Forwarding/Stalls Example

```
memcpy: ; $a0 = destiation, $a1 = source
        ; $a2 = length
      lb $t0, ($a1) ; $t0 <- MEM[$a1]
      sb $t0, ($a0) ; MEM[$a0] <- $t0
      addiu $a0, $a0, 1
      addiu $a1, $a1, 1
      addiu $a2, $a2, -1
      beqz $a2, memcpy
      nop ; DELAY SLOT
```

Quick reference:

IF	fetch instruction
ID	decode (read regs, compute branches)
EX	execute (ALU)
MEM	data memory
WB	write back

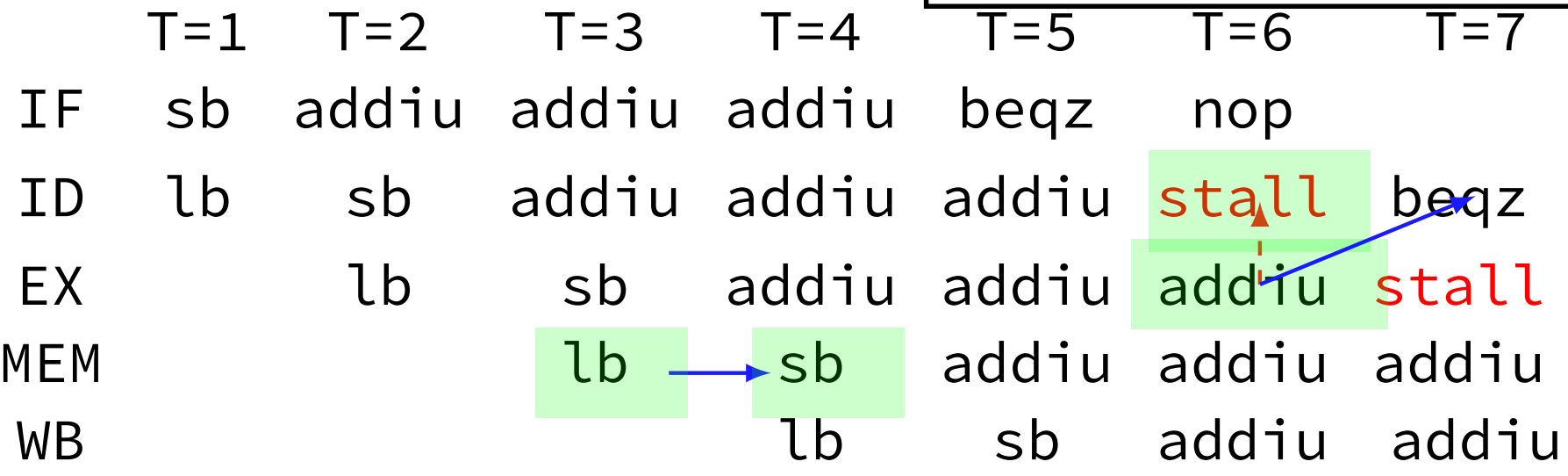
	T=1	T=2	T=3	T=4	T=5	T=6	T=7
IF	sb	addiu	addiu	addiu	beqz	nop	lb
ID	lb	sb	addiu	addiu	addiu	beqz	lb
EX		lb	sb	addiu	addiu	addiu	beqz
MEM			lb	sb	addiu	addiu	addiu
WB				lb	sb	addiu	addiu

Forwarding/Stalls Example

```
memcpy: ; $a0 = destiation, $a1 = source
        ; $a2 = length
      lb $t0, ($a1) ; $t0 <- MEM[$a1]
      sb $t0, ($a0) ; MEM[$a0] <- $t0
      addiu $a0, $a0, 1
      addiu $a1, $a1, 1
      addiu $a2, $a2, -1
      beqz $a2, memcpy
      nop ; DELAY SLOT
```

Quick reference:

IF	fetch instruction
ID	decode (read regs, compute branches)
EX	execute (ALU)
MEM	data memory
WB	write back



MIPS R2000 pipeline

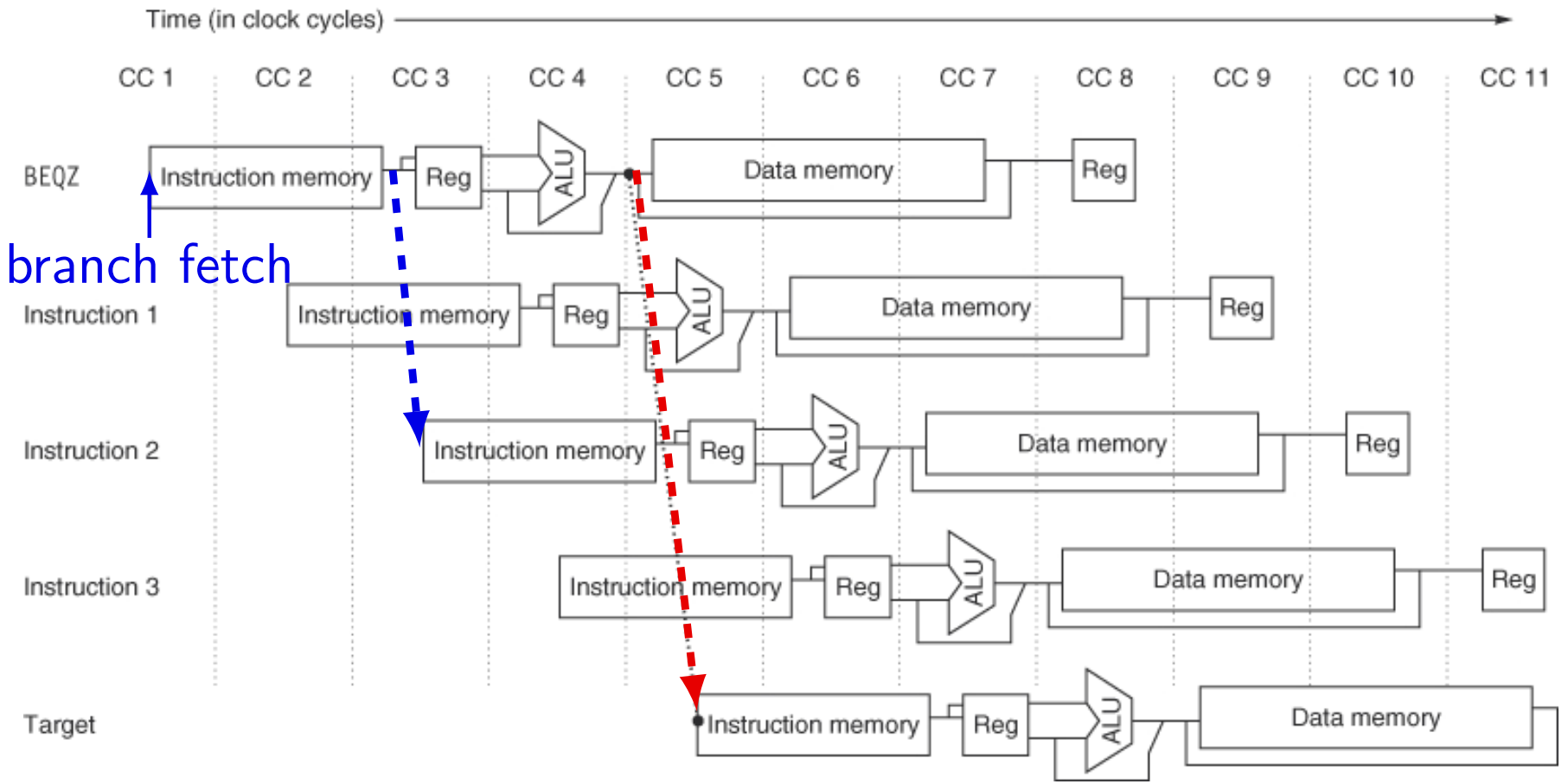


Figure C.44 The basic branch delay is 3 cycles, since the condition evaluation is performed during EX.

Figure: H&P Appendix C

MIPS R2000 pipeline

4 cycles without prediction

2 cycles with taken/not-taken prediction

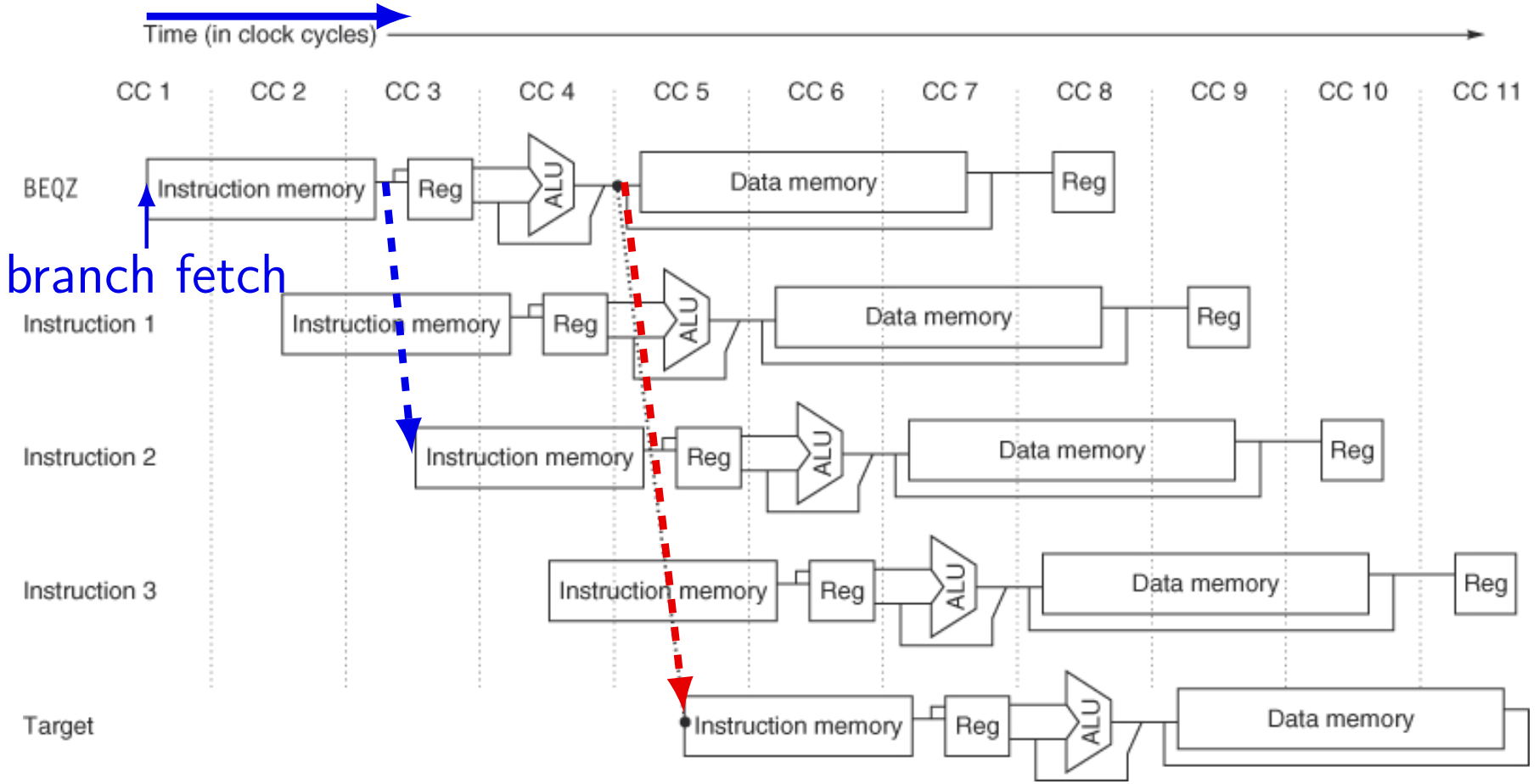
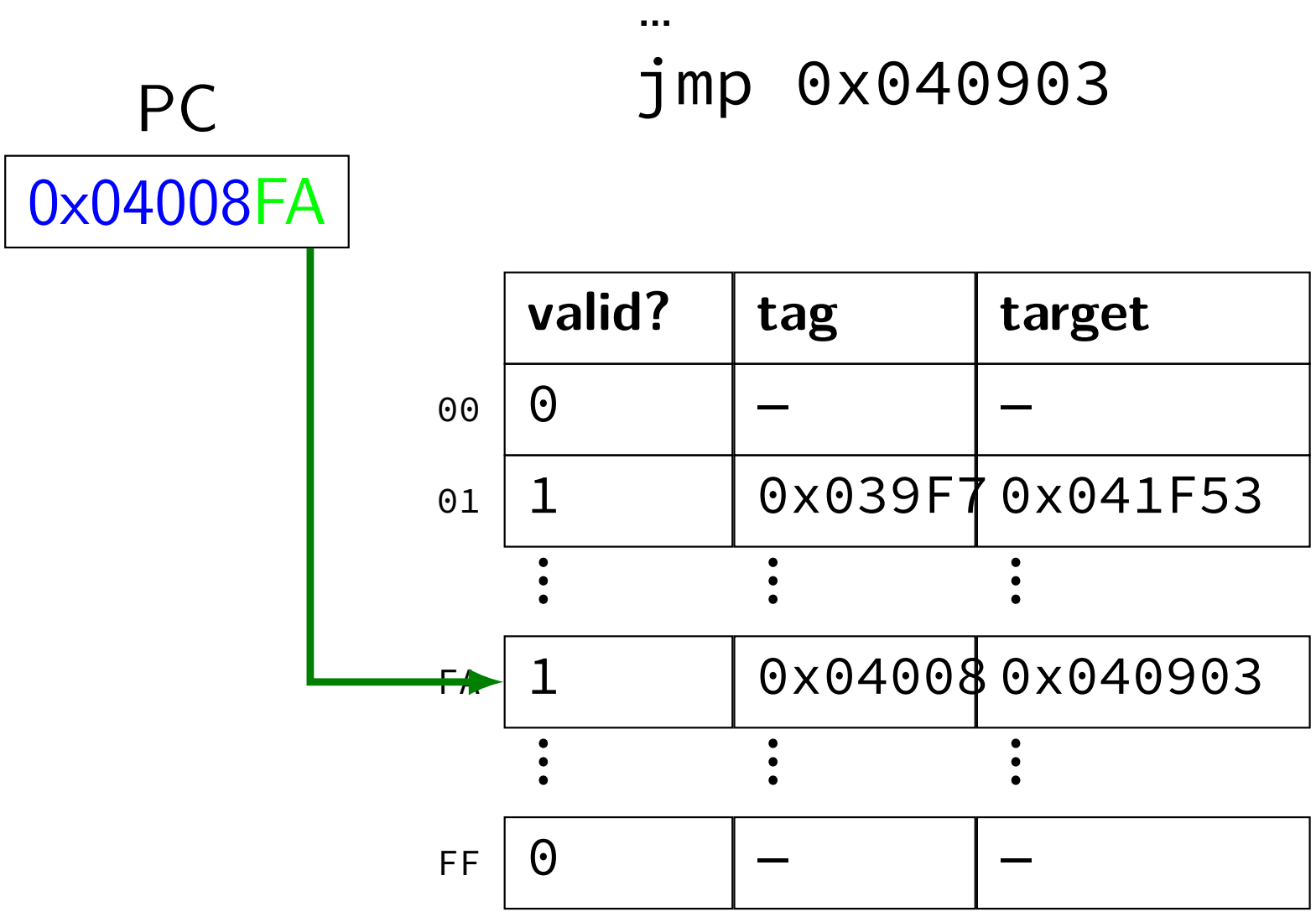


Figure C.44 The basic branch delay is 3 cycles, since the condition evaluation is performed during EX.

Figure: H&P Appendix C

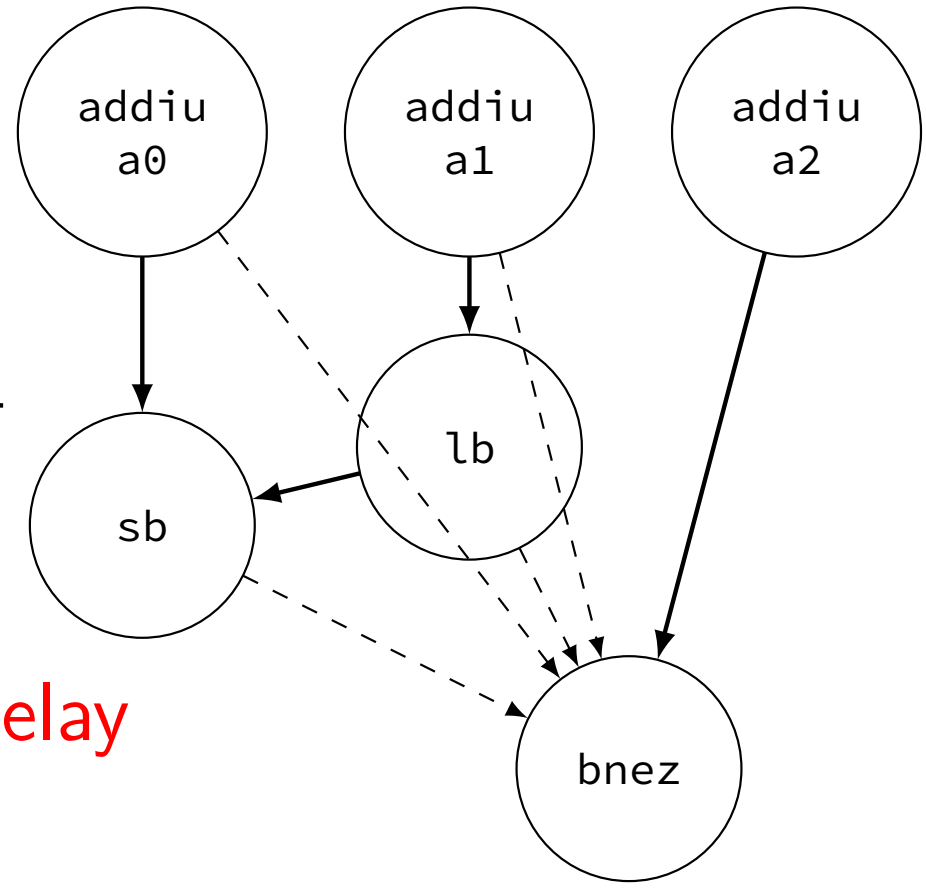
Branch Target Buffers



Scheduling dependencies

(badly ordered loop:)

```
addiu $a0, $a0, 1
addiu $a1, $a1, 1
lb $t0, ($a1)
sb $t0, ($a0)
addiu $a2, $a2, -1
bnez $a2, loop
```

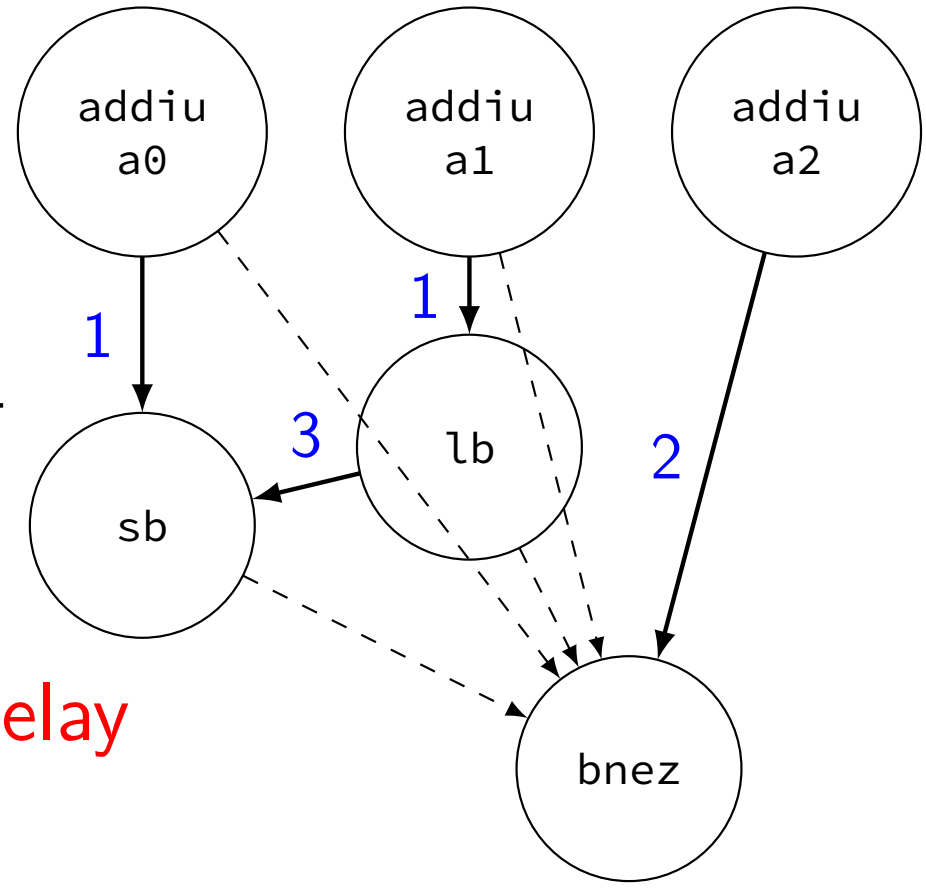


label edges with necessary delay

Scheduling dependencies

(badly ordered loop:)

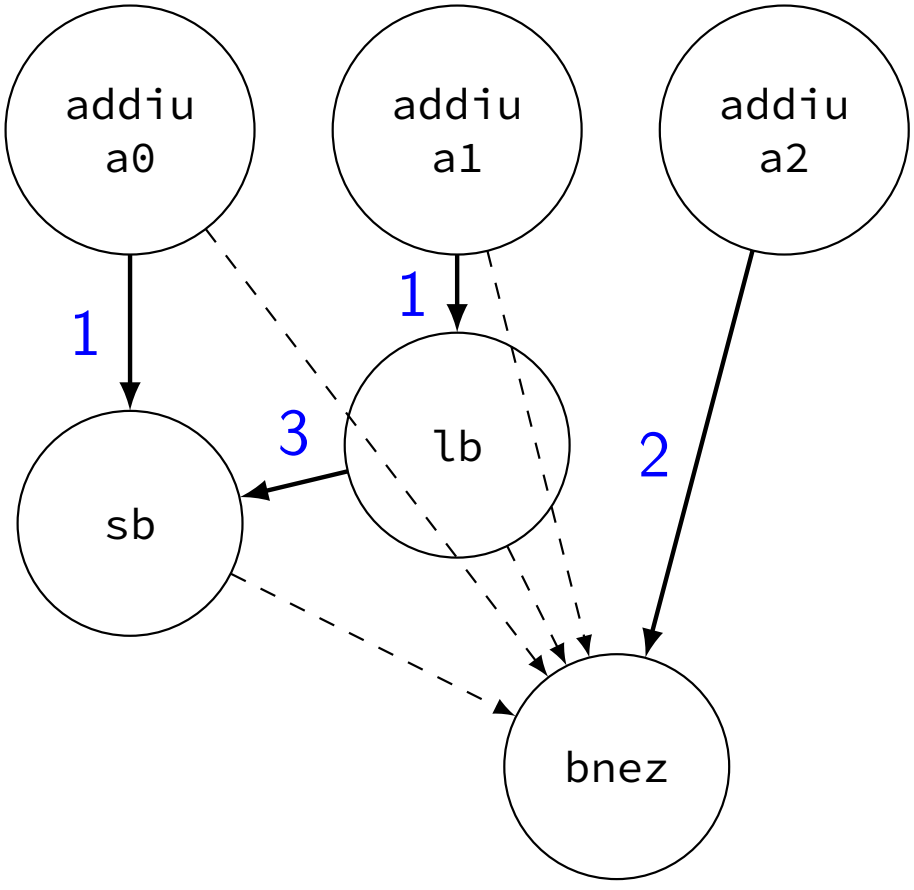
```
addiu $a0, $a0, 1
addiu $a1, $a1, 1
lb $t0, ($a1)
sb $t0, ($a0)
addiu $a2, $a2, -1
bnez $a2, loop
```



label edges with necessary delay

made up delays above

Scheduling dependencies



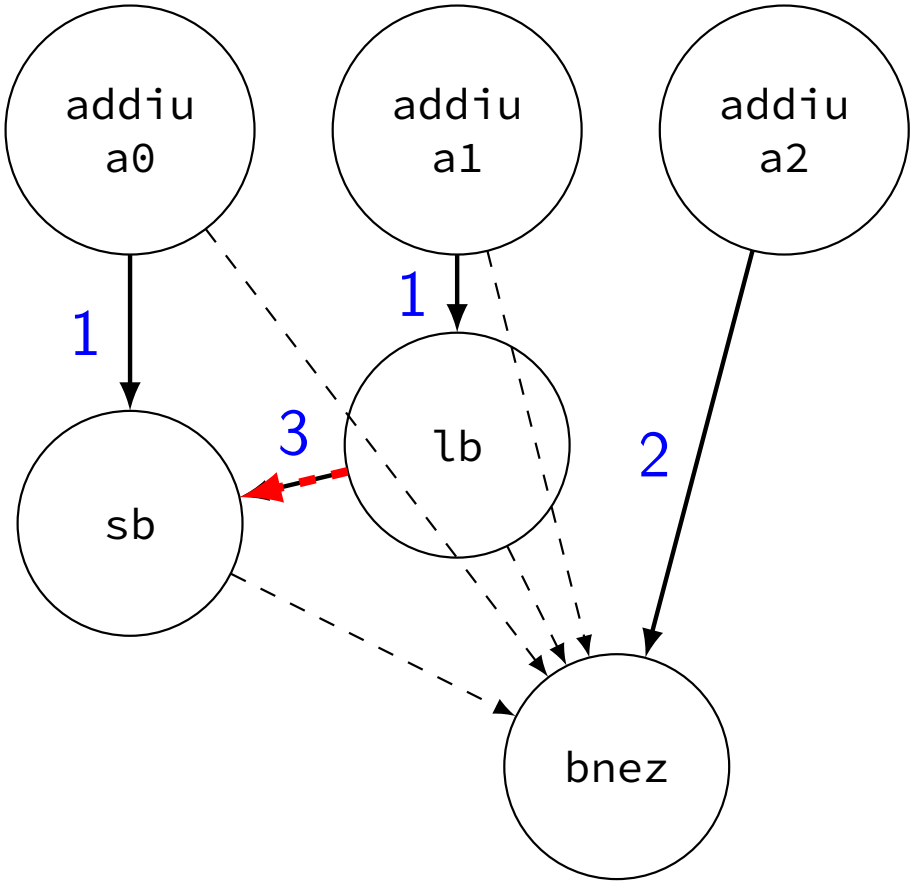
Good schedule:

```
addiu $a1, $a1, 1
lb $t0, ($a1)
addiu $a2, $a2, -1
addiu $a0, $a0, 1
sb $t0, ($a0)
bnez $a2, loop
```

correctness: parents before children

performance: distance $\geq$ edge weight

Scheduling dependencies



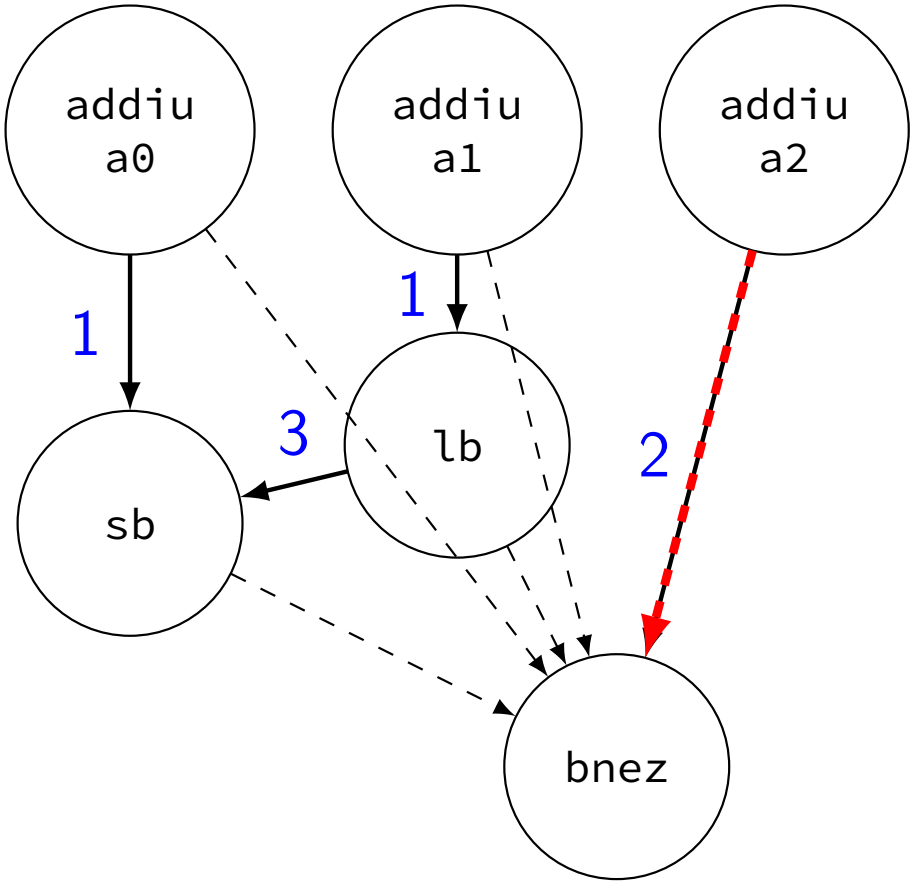
Good schedule:

```
addiu $a1, $a1, 1
lb $t0, ($a1)
addiu $a2, $a2, -1
addiu $a0, $a0, 1
sb $t0, ($a0)
bnez $a2, loop
```

correctness: parents before children

performance: distance $\geq$ edge weight

Scheduling dependencies



Good schedule:

```
addiu $a1, $a1, 1
lb $t0, ($a1)
addiu $a2, $a2, -1
addiu $a0, $a0, 1
sb $t0, ($a0)
bnez $a2, loop
```

correctness: parents before children

performance: distance $\geq$ edge weight

Loop Unrolling

```
for (int i = 0; i < size; i += 1) {
    dest[i] = source[i];
}
```

converts into

```
assert(i % 4 == 0);
for (int i = 0; i < size; i += 4) {
    dest[i] = source[i];
    dest[i+1] = source[i+1];
    dest[i+2] = source[i+2];
    dest[i+3] = source[i+3];
}
```

Loop Unrolling — Instructions

One iteration:

normal (4 instrs.)	unroll x 4 (10 instrs.)
load	load
store	store
add	load
branch	store
	load
	store
	load
	store
	add
	branch

$4N$ versus $2.5N = \frac{10}{4}N$

Loop Unrolling — Assembly

```
; $a0 = dest, $a1 = source, $a2 = i
lb $t0, 0($a1) ; load
sb $t0, 0($a0) ; store
lb $t0, 1($a1) ; load
sb $t0, 1($a0) ; store
lb $t0, 2($a1) ; load
sb $t0, 2($a0) ; store
lb $t0, 3($a1) ; load
sb $t0, 3($a0) ; store
addiu $a2, $a2, -1
bnez $a2, $a2, loop
```

Loop Unrolling — Fringe

```
for (int i = 0; i + 3 < size; i += 4) {
    dest[i] = source[i];
    dest[i+1] = source[i+1];
    dest[i+2] = source[i+2];
    dest[i+3] = source[i+3];
}
for (i = 0; i < size; i += 1) {
    dest[i] = source[i];
}
```

size $= N$ items: $10 \lfloor N/4 \rfloor + 1 + 4(N \mod 4)$

still less than normal if $N > 4$

Loop Unrolling — Cache Pressure

4x unrolled — $10 + 4 + 1 = 25$ instrs. in cache

normal — 4 instrs. in cache

6.25x space in I-cache for 4x unrolled

Loop Unrolling — Assembly

```
; $a0 = dest, $a1 = source, $a2 = i
lb $t0, 0($a1) ; load
sb $t0, 0($a0) ; store
lb $t0, 1($a1) ; load
sb $t0, 1($a0) ; store
lb $t0, 2($a1) ; load
sb $t0, 2($a0) ; store
lb $t0, 3($a1) ; load
sb $t0, 3($a0) ; store
addiu $a2, $a2, -1
bnez $a2, $a2, loop
```

Loop Unrolling — Scheduling?

```
; $a0 = dest, $a1 = source, $a2 = i
lb $t0, 0($a1) ; load 0
lb $t1, 1($a1) ; load 1
lb $t2, 2($a1) ; load 2
lb $t3, 3($a1) ; load 3
sb $t0, 0($a0) ; store 0
sb $t1, 1($a0) ; store 1
sb $t2, 2($a0) ; store 2
sb $t3, 3($a0) ; store 3
addiu $a2, $a2, -1
bnez $a2, $a2, loop
```

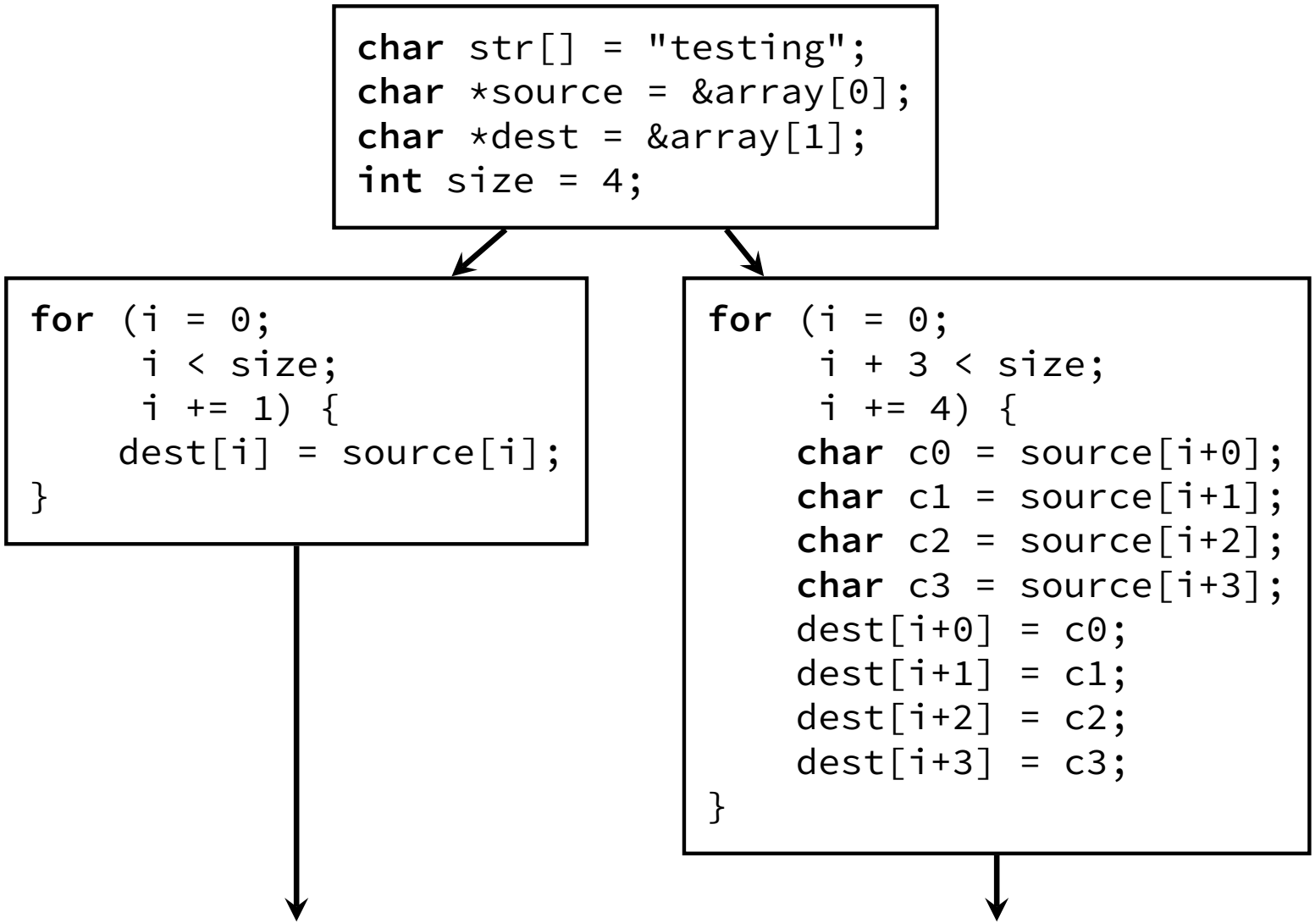
Maybe faster — but not always the same!

Loop Unrolling — Scheduling?

```
; $a0 = dest, $a1 = source, $a2 = i
lb $t0, 0($a1) ; load 0
lb $t1, 1($a1) ; load 1
lb $t2, 2($a1) ; load 2
lb $t3, 3($a1) ; load 3
sb $t0, 0($a0) ; store 0
sb $t1, 1($a0) ; store 1
sb $t2, 2($a0) ; store 2
sb $t3, 3($a0) ; store 3
addiu $a2, $a2, -1
bnez $a2, $a2, loop
```

Maybe faster — but not always the same!

Aliasing


char str[] = "testing";
char *source = &array[0];
char *dest = &array[1];
int size = 4;
for (i = 0;
i < size;
i += 1) {
dest[i] = source[i];
}
for (i = 0;
i + 3 < size;
i += 4) {
char c0 = source[i+0];
char c1 = source[i+1];
char c2 = source[i+2];
char c3 = source[i+3];
dest[i+0] = c0;
dest[i+1] = c1;
dest[i+2] = c2;
dest[i+3] = c3;
}
str is "ttttttng"
str is "ttestng"

Aliasing testing

```
if (source - dest > 0 && source - dest < 4) {
    for (i = 0; i + 3 < size; i += 4) {
        dest[i+0] = source[i+0]; dest[i+1] = source[i+1];
        dest[i+2] = source[i+2]; dest[i+3] = source[i+3];
    }
} else {
    for (i = 0; i + 3 < size; i += 4) {
        char c0 = source[i+0], c1 = source[i+1], c2 = source[i+2], c3 = s
        dest[i+0] = c0; dest[i+1] = c1; dest[i+2] = c2; dest[i+3] = c3;
    }
}
for (; i < size; i += 1) {
    dest[i] = source[i];
}
```

even bigger code size

tricky for compilers

Aliasing testing

```
if (source - dest > 0 && source - dest < 4) {
    for (i = 0; i + 3 < size; i += 4) {
        dest[i+0] = source[i+0]; dest[i+1] = source[i+1];
        dest[i+2] = source[i+2]; dest[i+3] = source[i+3];
    }
} else {
    for (i = 0; i + 3 < size; i += 4) {
        char c0 = source[i+0], c1 = source[i+1], c2 = source[i+2], c3 = s
        dest[i+0] = c0; dest[i+1] = c1; dest[i+2] = c2; dest[i+3] = c3;
    }
}
for (; i < size; i += 1) {
    dest[i] = source[i];
}
```

even bigger code size

tricky for compilers

Loop Unrolling: Register Pressure

better scheduling — extra registers

our example: 7 vs. 4 in original

generally — extra places to put results to hide

Software Pipelining

Hide memory latency in loops

Load earlier

More time for arithmetic to catch up

Store later

Be sure arithmetic alreday finished

Instruction scheduling — across iterations

Splitting Loops

```
for (i = 0; i < size; i += 1) {
    a[i] += b[i] * c[i];
}
/* transforms into: */
for (i = 0; i < size; i += 1) {
    /* LOADS: */
    curA = a[i]
    curB = b[i];
    curC = c[i];
    /* EXECUTE: */
    curA += curB * curC;
    /* STORES: */
    a[i] = curA;
}
```

Rearranging

```
LOOP:  L_i
       E_i
       S_i
       JC LOOP
```

(a)

```
       L_1
LOOP:  E_i
       S_i
       L_{i+1}
       JC LOOP
       E_n
       S_n
```

(b)

```
       J ENTRY
LOOP:  S_{i-1}
ENTRY: L_i
       E_i
       JC LOOP
       S_n
```

(c)

```
       L_1
       J ENTRY
LOOP:  S_{i-1}
ENTRY: E_i
       L_{i+1}
       JC LOOP
       S_{n-1}
       E_n
       S_n
```

(d)

```
       L_1
LOOP:  E_i
       L_{i+1}
       S_i
       JC LOOP
       E_n
       S_n
```

(e)

```
       L_1
       J ENTRY
LOOP:  L_i
       S_{i-1}
ENTRY: E_i
       JC LOOP
       S_n
```

(f)

Order (e)

```
/* Load 1 */
curA = a[0]; curB = b[0]; curC = c[0];
for (i = 0; i < size - 1; i += 1) {
    /* Execute i */
    prevA = curA + curB * curC;
    /* Load i + 1 */
    curA = a[i + 1];
    /* Store i */
    a[i] = prevA;
}
prevA = curA + curB * curC;
a[i] = prevA;
```

Software Pipelining: Register Pressure

load i + 1 before storing i — twice the registers!

Decoupled Architecture

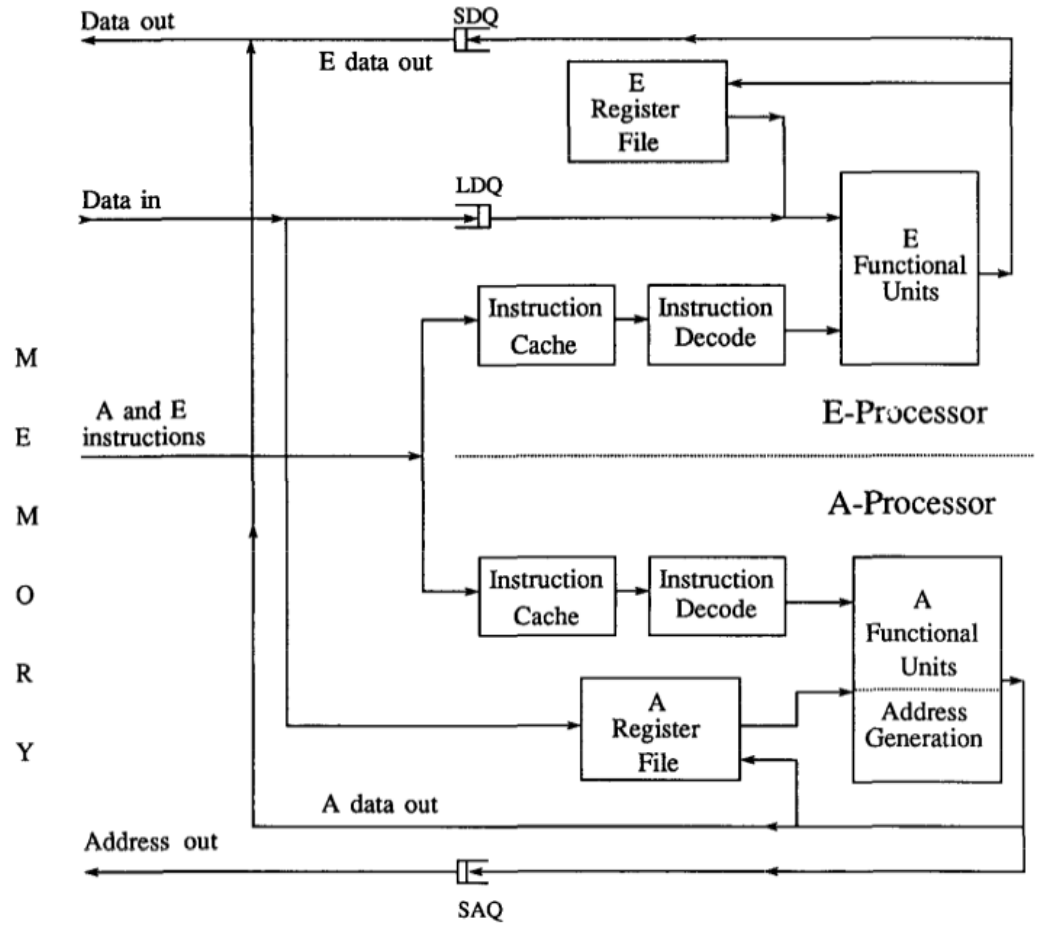


Fig. 5. A streamlined decoupled architecture.

idea: execute memory instructions independently

synchronize with queues

Decoupled Architecture Decomposition

```
/* Access Program */
for (i = 0; i < size - 1; i += 1) {
    EnqueueLoad(&a[i]);
    EnqueueLoad(&b[i]);
    EnqueueLoad(&c[i]);
    EnqueueStore(&a[i]);
}
/* Execute Program */
for (i = 0; i < size - 1; i += 1) {
    curA = DequeueLoad();
    curB = DequeueLoad();
    curC = DequeueLoad();
    EnqueueStore(curA + curB * curC);
}
```

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

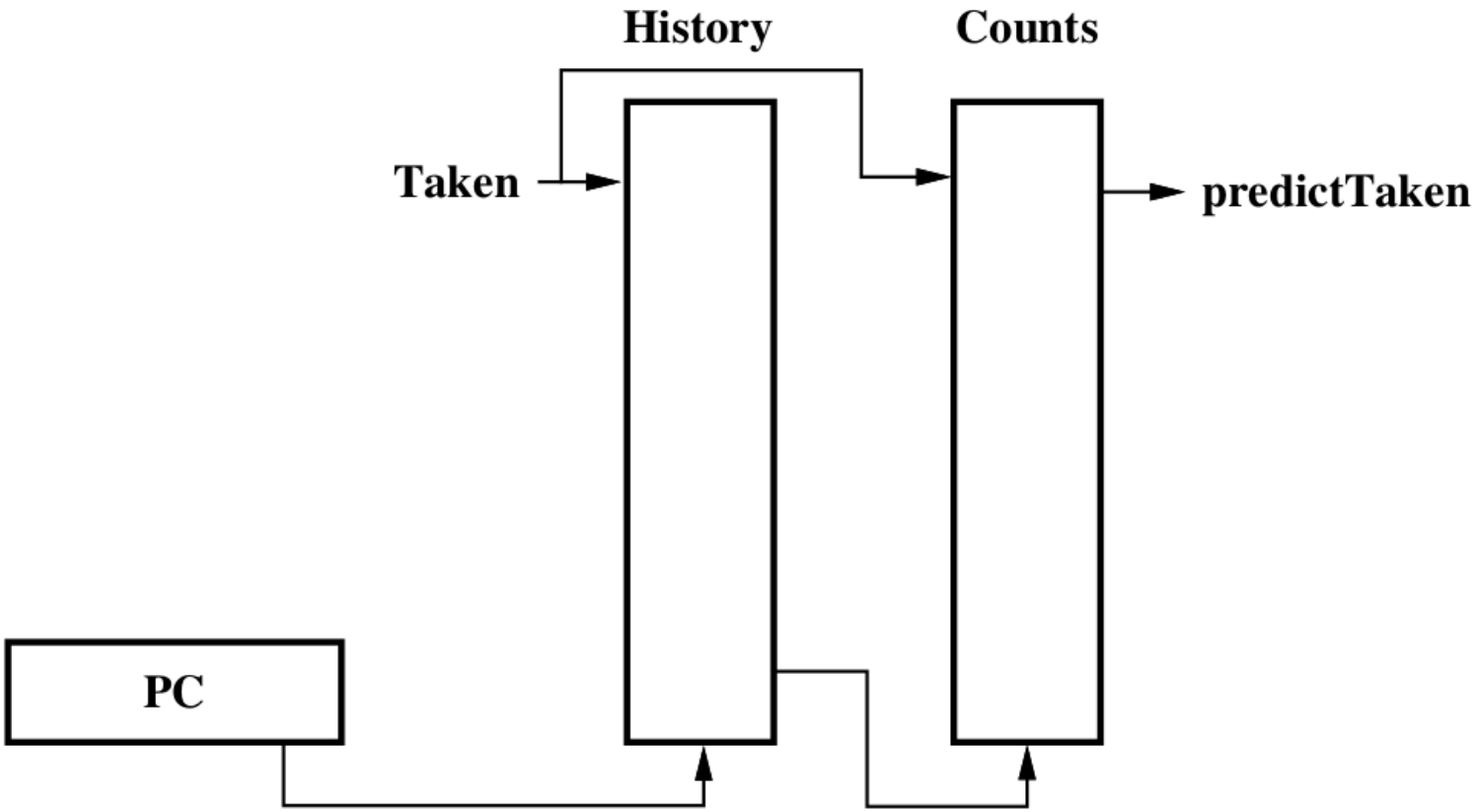


Figure 4: Local History Predictor Structure

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

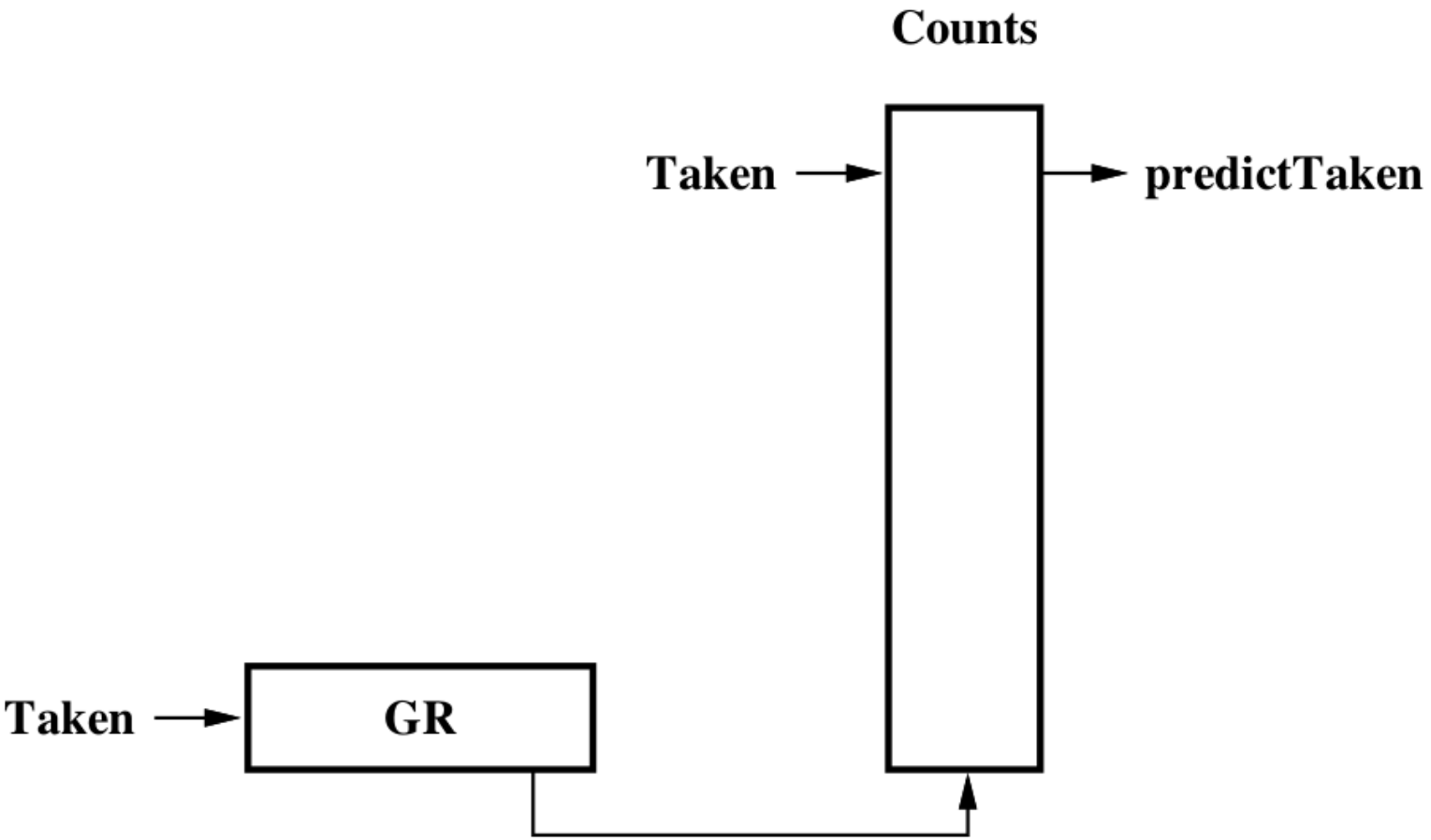
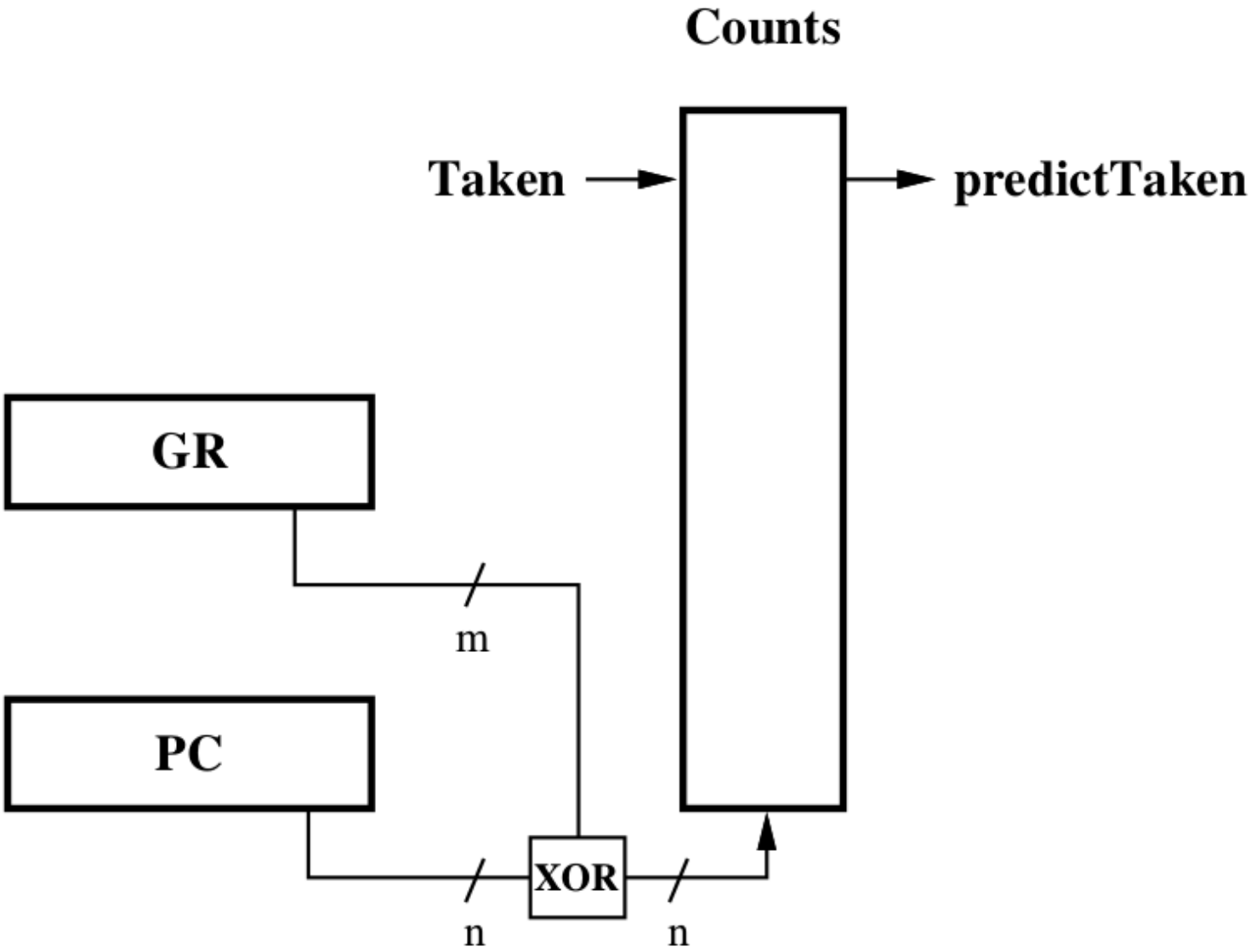


Figure 6: Global History Predictor Structure

Combining local and global



Combining generally

branch predictor predictor

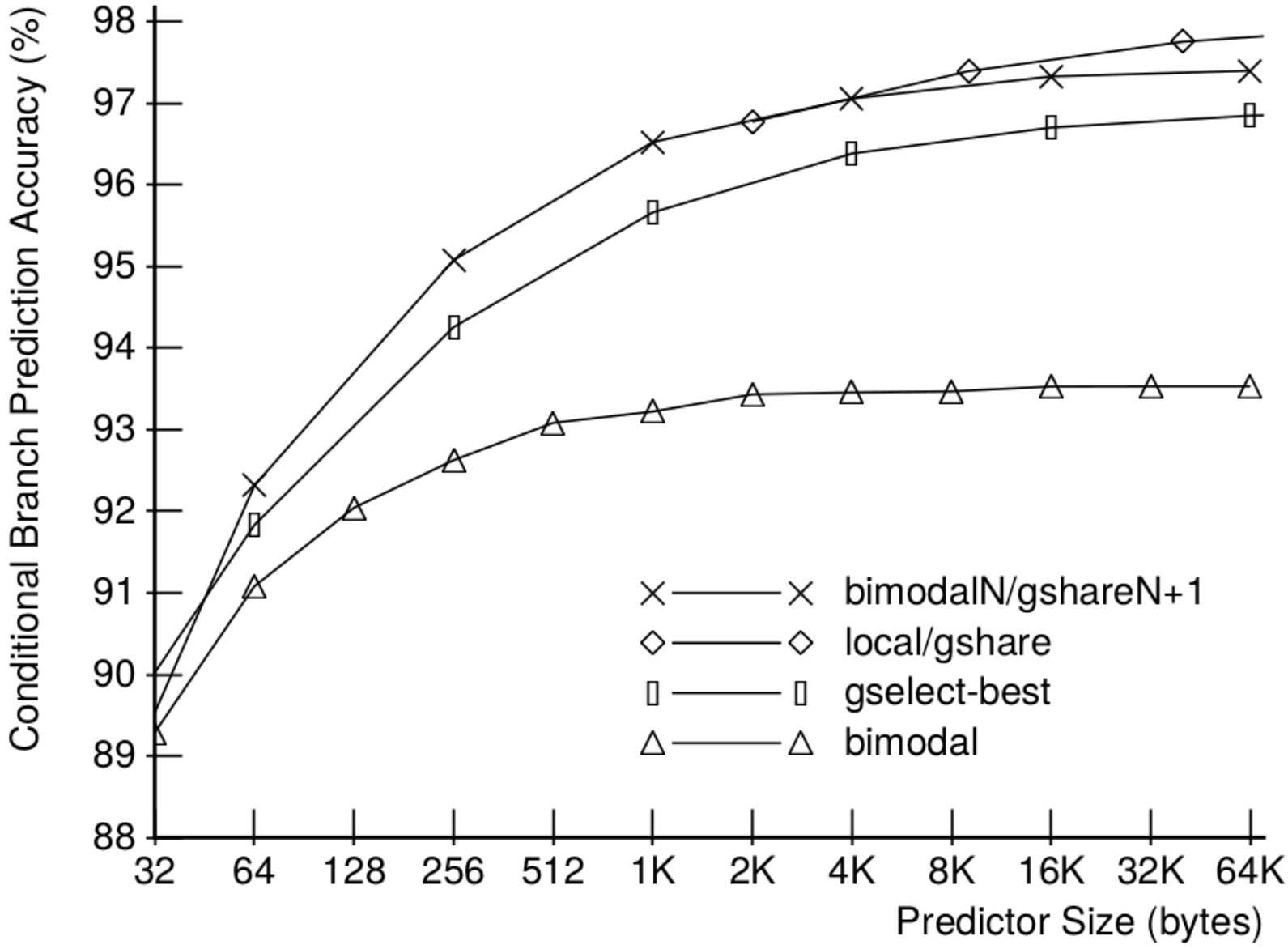
counter per branch:

- 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

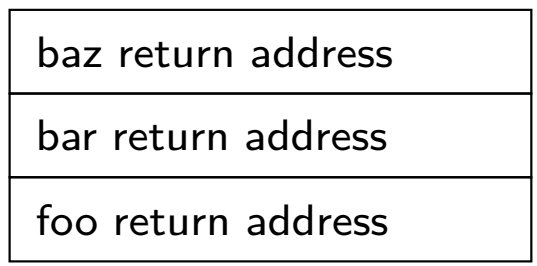


Return address stack

Predicting function call return addresses

Solution: stack in processor registers

stack in memory



shadow stack
in CPU 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?

...

Preview: Vectorization

single-instruction multiple-data

convert byte-oriented iteration:

```
lb $t0, 0($a1) ; load 0
lb $t1, 1($a1) ; load 1
lb $t2, 2($a1) ; load 2
lb $t3, 3($a1) ; load 3
sb $t0, 0($a0) ; store 0
sb $t1, 1($a0) ; store 1
sb $t2, 2($a0) ; store 2
sb $t3, 3($a0) ; store 3
```

into word (4-byte)-oriented

```
lw $t0, 0($a1) ; load 0–3
sw $t0, 0($a1) ; store 0–3
```

Preview: Vectorization

single instruction multiple-data

```
x[i] *= y[i]; x[i+1] *= y[i+1];
```

can become (x86):

```
movupd (%rsi,%rcx,8), %xmm0 # xmm0 = {y[i],y[i+1]}
movupd (%rdi,%rcx,8), %xmm1 # xmm1 = {x[i],x[i+1]}
mulpd %xmm0, %xmm1          # xmm1 = {xmm0[0]*xmm1[0], xmm0[1]*xmm1[1]}
movupd %xmm1, (%rdi,%rcx,8) # {x[i],x[i+1]} = xmm1
```

instead of:

```
movsd (%rsi,%rcx,8), %xmm0  # xmm0 = y[i]
movsd (%rdi,%rcx,8), %xmm1  # xmm1 = x[i]
mulsd %xmm0, %xmm1          # xmm1 = xmm0*xmm1
movsd %xmm1, (%rdi,%rcx,8)  # x[i] = xmm1
movsd 8(%rsi,%rcx,8), %xmm0 # xmm0 = y[i+1]
movsd 8(%rdi,%rcx,8), %xmm1 # xmm1 = x[i+1]
mulsd %xmm0, %xmm1          # xmm1 = xmm0*xmm1
movsd %xmm1, 8(%rdi,%rcx,8) # x[i+1] = xmm1
```

Modern vectorization

many more functional units (e.g. multipliers)

only accessible using vectorized interfaces

Next time

starting multiple instructions at the same time

intuition: multiple copies of pipeline

to start: scheduled by compiler