
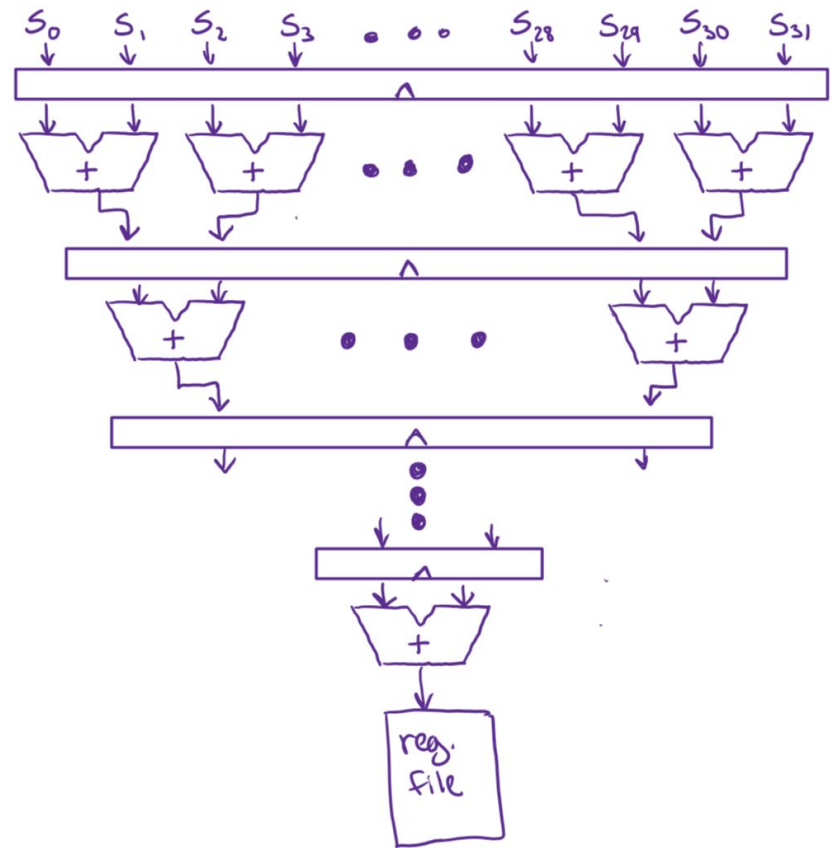
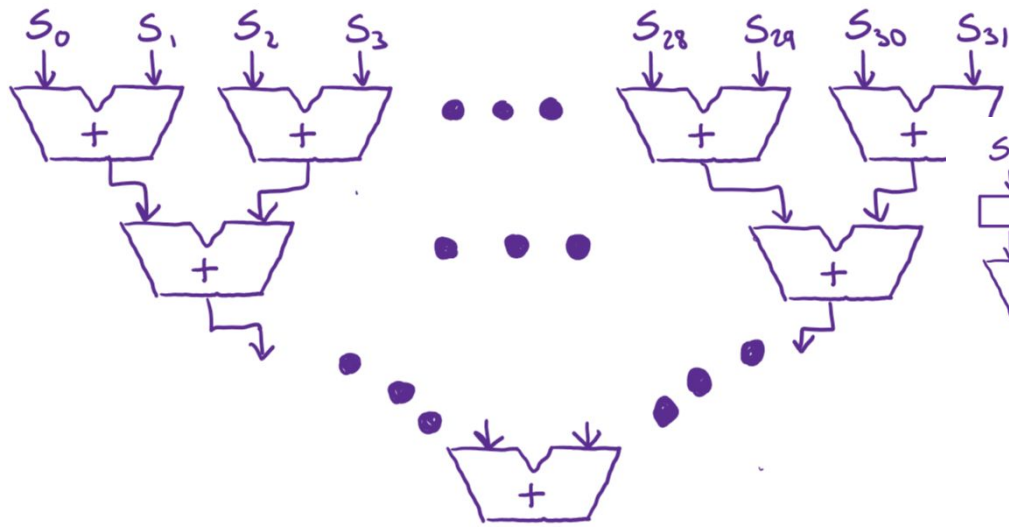


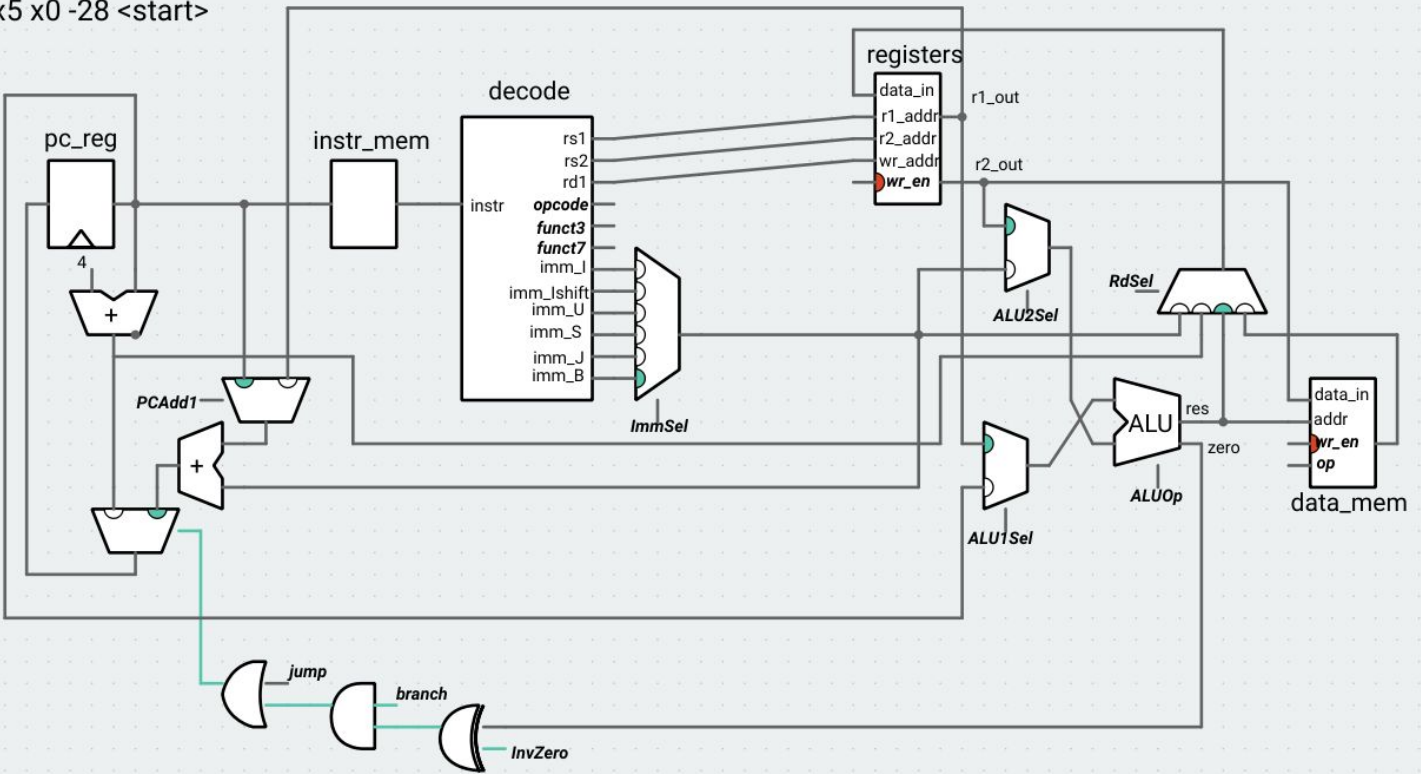
**Instruction-level  
parallelism; data  
and control  
dependences**

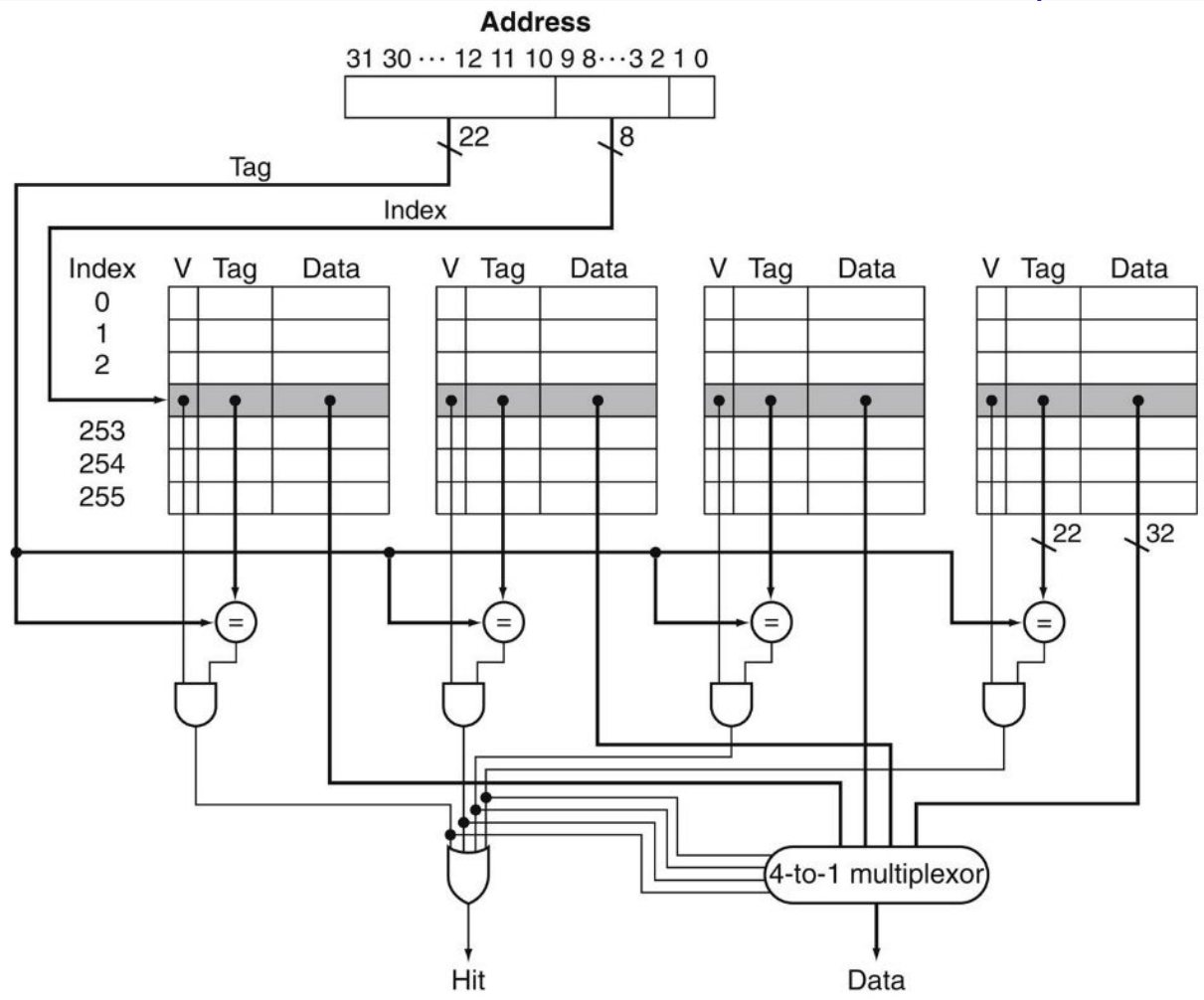




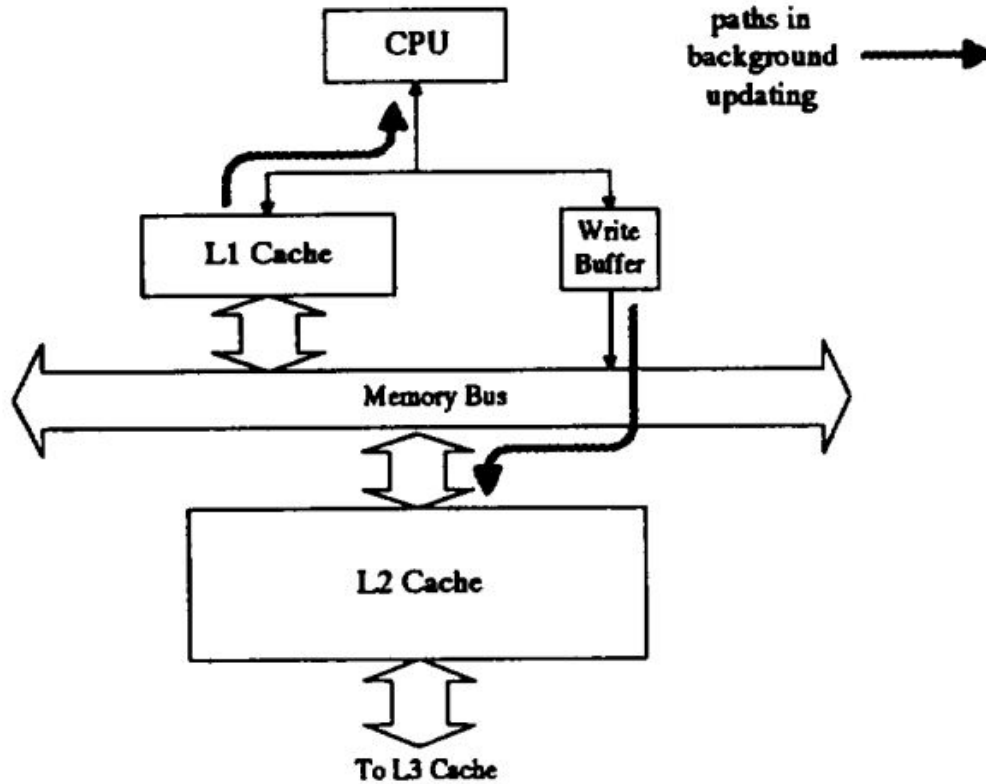
# Single-stage CPU (built from scratch in CS1952y!)

bne x5 x0 -28 <start>





P&H fig. 5.18



**Figure 1: 2-Level Cache with Write Buffer**

*P. P. Chu and R. Gottipati, "Write buffer design for on-chip cache," Proceedings 1994 IEEE International Conference on Computer Design: VLSI in Computers and Processors, Cambridge, MA, USA, 1994, pp. 311-316, doi: 10.1109/ICCD.1994.331913.*

two ways to interpret an address:



need phys. addr for this

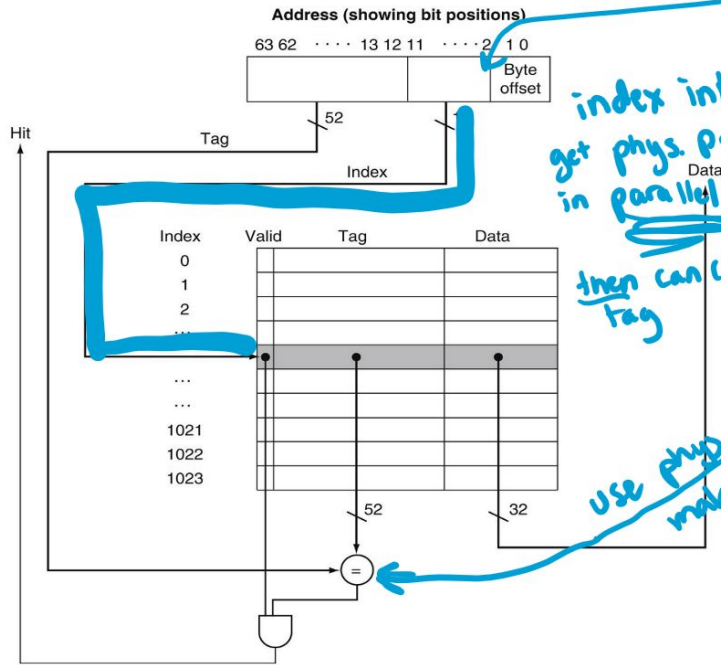
same for v & p addr  
if \$I\$ size is smaller than page size, these bits will live in the page offset

TLB

| Virtual page number | Valid | Dirty | Ref | Tag | Physical page address |
|---------------------|-------|-------|-----|-----|-----------------------|
|                     | 1     | 0     | 1   |     |                       |
|                     | 1     | 1     | 1   |     |                       |
|                     | 1     | 1     | 1   |     |                       |
|                     | 1     | 0     | 1   |     |                       |
|                     | 0     | 0     | 0   |     |                       |
|                     | 1     | 0     | 1   |     |                       |

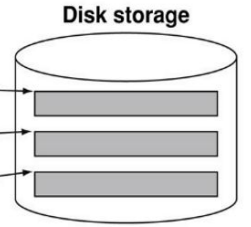
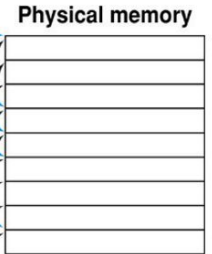
index into \$I\$ & get phys. page no. in parallel!  
then can compare tag

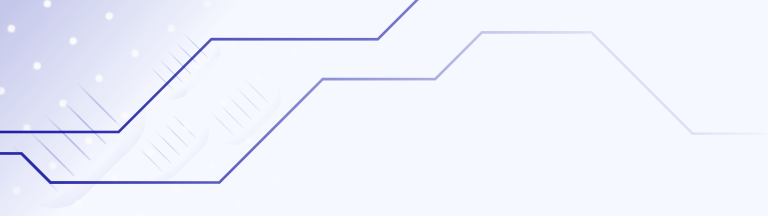
use phys. addr to make tag bits




Page table

| Valid | Dirty | Ref | Physical page or disk address |
|-------|-------|-----|-------------------------------|
| 1     | 0     | 1   |                               |
| 1     | 0     | 0   |                               |
| 1     | 0     | 0   |                               |
| 1     | 0     | 1   |                               |
| 0     | 0     | 0   |                               |
| 1     | 0     | 1   |                               |
| 0     | 0     | 0   |                               |
| 1     | 1     | 1   |                               |
| 1     | 1     | 1   |                               |
| 0     | 0     | 0   |                               |
| 1     | 1     | 1   |                               |

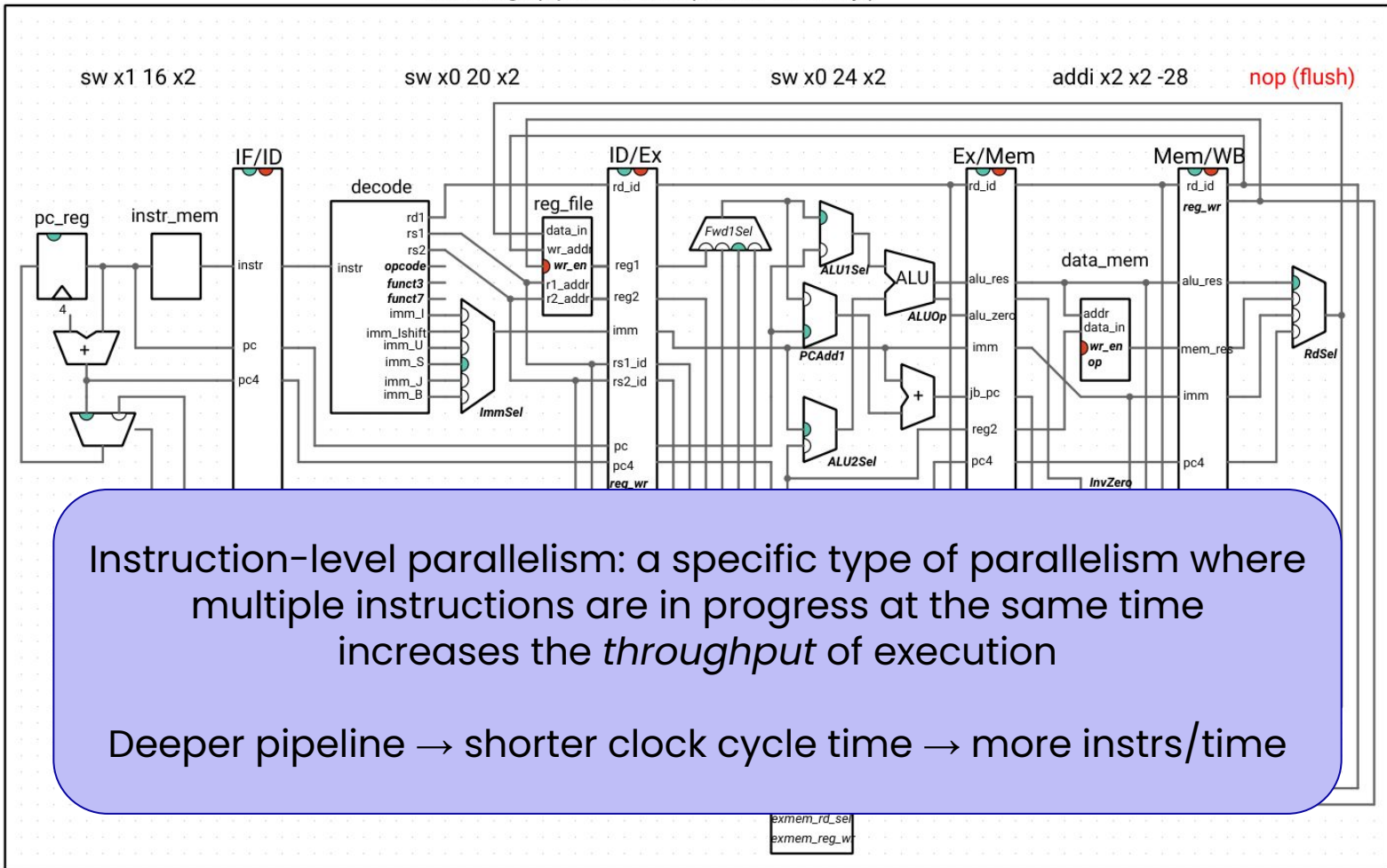




**Hardware allows us to speed up execution time by performing operations in parallel**



## Five-stage pipelined CPU (built in CS1952y!)



Instruction-level parallelism: a specific type of parallelism where multiple instructions are in progress at the same time increases the *throughput* of execution

Deeper pipeline → shorter clock cycle time → more instrs/time

exmem\_rd\_sel  
exmem\_reg\_wr





What makes ILP challenging?

# Where's the hazard?

```
for (int i = 0; i < 100; i++) {  
    A[i] = A[i] + B[i];  
}
```

```
addi t0, x0, 0    // t0/i = 0  
addi t1, x0, 100 // t1 = 100  
loop: bge t0, t1, end  
slli t2, t0, 2    // t2 = t0/i * 4  
add t3, a0, t2    // t3 = A + t2  
add t4, a1, t2    // t4 = B + t2  
lw t2, 0(t4)     // t2 = B[i]  
lw t4, 0(t3)     // t4 = A[i]  
add t4, t4, t2    // t4 = A[i] + B[i]  
sw t4, 0(t3)     // A[i] = A[i] + B[i]  
addi t0, t0, 1   // t0/i++  
j loop  
end: nop
```

# Same result, lower CPI (1.44/1.35)

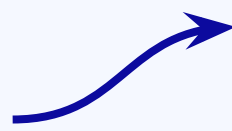
```
addi t0, x0, 0
addi t1, x0, 100
loop: bge t0, t1, end
slli t2, t0, 2
add t3, a0, t2
add t4, a1, t3
lw t2, 0(t4)
lw t4, 0(t3)
add t4, t4, t2
sw t4, 0(t3)
addi t0, t0, 1
j loop
end: nop
```

We could have the compiler do this

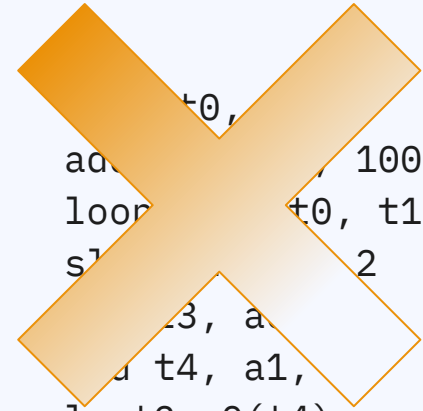
or

We could have the CPU do this

Either way: how do we maintain correctness?



```
addi t0, x0, 0
addi t1, x0, 100
loop: bge t0, t1, end
slli t2, t0, 2
add t3, a0, t2
add t4, a1, t3
lw t2, 0(t4)
lw t4, 0(t3)
add t4, t4, t2
sw t4, 0(t3)
addi t0, t0, 1
j loop
end: nop
```



```
addi t0, x0, 0
addi t1, x0, 100
loop: bge t0, t1, end
slli t2, t0, 2
add t3, a0, t2
add t4, a1, t3
lw t2, 0(t4)
lw t4, 0(t3)
sw t4, 0(t3)
add t4, t4, t2
addi t0, t0, 1
j loop
end: nop
```

# Data dependences (H&P 3.1.2.1)

instruction  $j$  is data dependent on instruction  $i$  when:

- 1) instruction  $i$  produces a result that may be used by instruction  $j$

or

- 2) Instruction  $j$  is data dependent on instruction  $k$ , and instruction  $k$  is data dependent on instruction  $i$

→  $i$  should happen before  $j$

```
addi t0, x0, 0
addi t1, x0, 100
loop: bge t0, t1, end
slli t2, t0, 2
add t3, a0, t2
add t4, a1, t2
lw t2, 0(t4)
lw t4, 0(t3)
add t4, t4, t2
sw t4, 0(t3)
addi t0, t0, 1
j loop
end: nop
```

# Name dependences (H&P 3.1.2.2)

Dependences where no flow of data exists between instructions  $i$  and  $j$

**Antidependence:** instruction  $j$  writes a register or memory location that instruction  $i$  reads.

**Output dependence:** instructions  $i$  and  $j$  write to the same register or memory location

→  $i$  should happen before  $j$

```
addi t0, x0, 0
addi t1, x0, 100
loop: bge t0, t1, end
slli t2, t0, 2
add t3, a0, t2
add t4, a1, t2
lw t2, 0(t4)
lw t4, 0(t3)
add t4, t4, t2
sw t4, 0(t3)
addi t0, t0, 1
j loop
end: nop
```

# Data hazard classification

The pipelines we saw needed to stall on a RAW (read after write) hazard

```
lw t4, 0(t3)
add t4, t4, t2
```

Depending on the processor configuration, there may also be:

WAW (write after write) hazards: possible in pipelines that write in multiple stages

WAR (write after read) hazards: not an issue in modern *in-order* pipelines (reads happen before writes), but arise in *out-of-order* processors due to antidependences

# Basic blocks

```
addi t0, x0, 0    // t0/i = 0
addi t1, x0, 100 // t1 = 100
loop: bge t0, t1, end
    slli t2, t0, 2    // t2 = t0/i * 4
    add t3, a0, t2    // t3 = A + t2
    add t4, a1, t2    // t4 = B + t2
    lw t2, 0(t4)      // t2 = B[i]
    lw t4, 0(t3)      // t2 = A[i]
    add t4, t4, t2    // t4 = A[i] + B[i]
    sw t4, 0(t3)      // A[i] = A[i] + B[i]
    addi t0, t0, 1    // t0/i++
    j loop
end: nop
```

Sequence of instructions between branches/jumps



What else could we parallelize here?

```
for (int i = 0; i < 100; i++) {  
    A[i] = A[i] + B[i];  
}
```





# Where else is the hazard?

We reduced CPI by about 6% by reordering... but the real culprit keeping CPI > 1 is the branch and jump!

We can use hardware to compute jump addresses earlier (P&H 4.8) ... but there will still be at least one cycle wasted

```
addi t0, x0, 0    // t0/i = 0
addi t1, x0, 100 // t1 = 100
loop: bge t0, t1, end
slli t2, t0, 2    // t2 = t0/i * 4
add t3, a0, t2    // t3 = A + t2
add t4, a1, t2    // t4 = B + t2
lw t2, 0(t4)     // t2 = B[i]
lw t4, 0(t3)     // t4 = A[i]
add t4, t4, t2   // t4 = A[i] + B[i]
sw t4, 0(t3)     // A[i] = A[i] + B[i]
addi t0, t0, 1   // t0/i++
j loop
end: nop
```

# Control dependences

```
if x:  
    P1 // depends on x, not y  
if y:  
    P2 // depends on y, not x
```

When dealing with control dependences:

- 1) instruction dependent on branch should not be moved before branch
- 2) instruction not dependent on branch should not be moved after branch

But this is pretty restrictive... instead, we may allow for instructions to be executed (or partially executed) as long as we can preserve the correctness of the program somehow

# Branch delay slot

Some architectures execute one instruction immediately after a branch/jump instruction (regardless if the branch is taken)

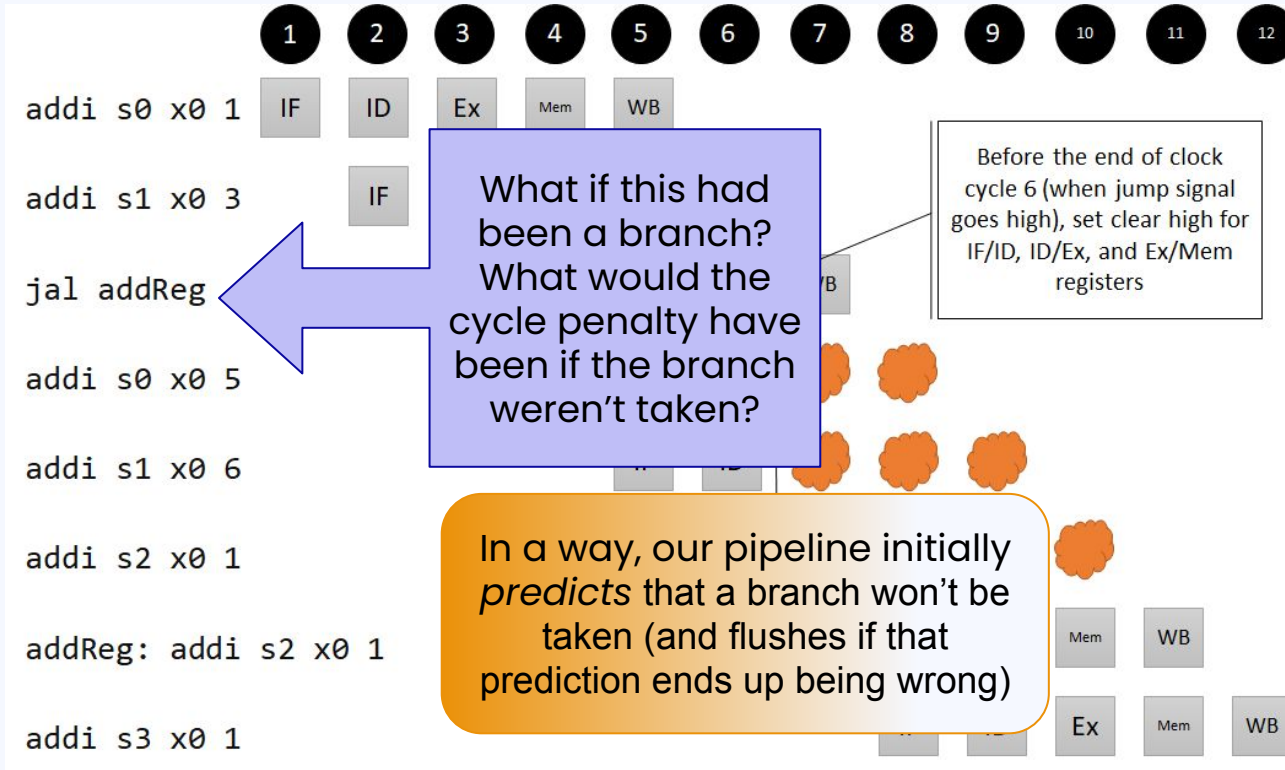
Up to compiler and/or CPU to move an independent instruction into that slot

Along w/ hardware, this helps us basically hide the cost of unconditional jumps

What about branches?

```
addi t0, x0, 0
addi t1, x0, 100
loop: bge t0, t1, end
(some independent instr from earlier)
slli t2, t0, 2
add t3, a0, t2
add t4, a1, t2
lw t2, 0(t4)
lw t4, 0(t3)
add t4, t4, t2
addi t0, t0, 1
j loop
sw t4, 0(t3)
end: nop
```

# We've already done this, sort of!





Can we do better than “predict branch not taken?”

# Double-loop code

```
addi t0, x0, 0      // t0 = 0
addi t1, x0, 400    // t1 = 400
l1: addi t2, x0, 3   // t2 = 3
l2: addi t0, t0, 1   // t0++
addi t2, t2, -1     // t2--
bne t2, x0, l2      // loop while t2 > 0
slli t0, t0, 1      // t0 <<= 1
addi t1, t1, -1     // t1--
bne t1, x0, l1      // loop while t1 > 0
```

Dynamic branch prediction:  
CPU can choose to predict  
(keep executing as if)  
branch is taken OR branch  
is not taken

Flushes instructions if it's  
wrong

How? Keep a Branch  
Prediction Buffer (Branch  
History Table) that maps  
branch instr addresses to  
predictions

# 1-bit BPB entry

Works great for outer loop (one misprediction and then 400 correct predictions)

Works less great for inner loop:

## Branch prediction

Wednesday, March 6, 2024 9:42 AM

```
addi t0, x0, 0    // t0 = 0
addi t1, x0, 400  // t1 = 400
l1: addi t2, x0, 3 // t2 = 3
l2: addi t0, t0, 1 // t0++
addi t2, t2, -1   // t2--
→ bne t2, x0, l2   // loop while t2 > 0
slli t0, t0, 1    // t0 <= 1
addi t1, t1, -1   // t1--
bne t1, x0, l1    // loop while t1 > 0
```

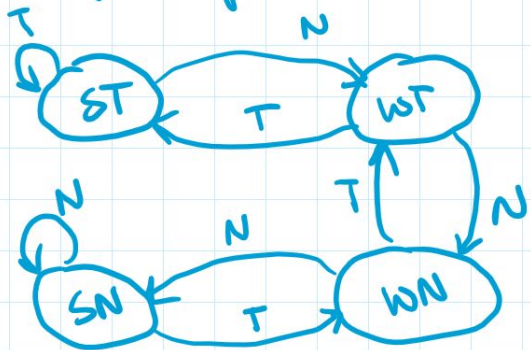
| time<br>encountering<br>branch | t2  | BPB<br>entry | branch<br>taken? | correct<br>prediction? |
|--------------------------------|-----|--------------|------------------|------------------------|
| 0                              | 1   | 0            | 1                | 1                      |
| 1                              | 2   | 0            | 1                | X                      |
| 2                              | 1   | 0            | 1                | ✓                      |
| 3                              | 0   | 0            | 1                | X                      |
| 4                              | 2   | 0            | 1                | X                      |
| 5                              | 1   | 0            | 1                | ✓                      |
| 6                              | 0   | 0            | 1                | X                      |
|                                | ... |              |                  |                        |

# 2-bit BPB entry

2 bits can keep track of 4 states: strong taken, weak taken, weak not taken, strong not taken

Keeps some of history (means branch prediction needs to be wrong twice instead of once before changing) – works better for the inner loop!

input is actual behavior  
output is prediction



| time<br>of counting<br>branch | t?  | BPB<br>entry | branch<br>taken? | correct<br>prediction? |
|-------------------------------|-----|--------------|------------------|------------------------|
| 0                             | -   | WN           | -                | -                      |
| 1                             | 2   | NN           | Y                | X                      |
| 2                             | 1   | WT           | Y                | ✓                      |
| 3                             | 0   | ST           | N                | X                      |
| 4                             | 2   | WT           | Y                | ✓                      |
| 5                             | 1   | ST           | Y                | ✓                      |
| 6                             | 0   | ST           | N                | X                      |
| ...                           | ... | ...          | ...              | ...                    |