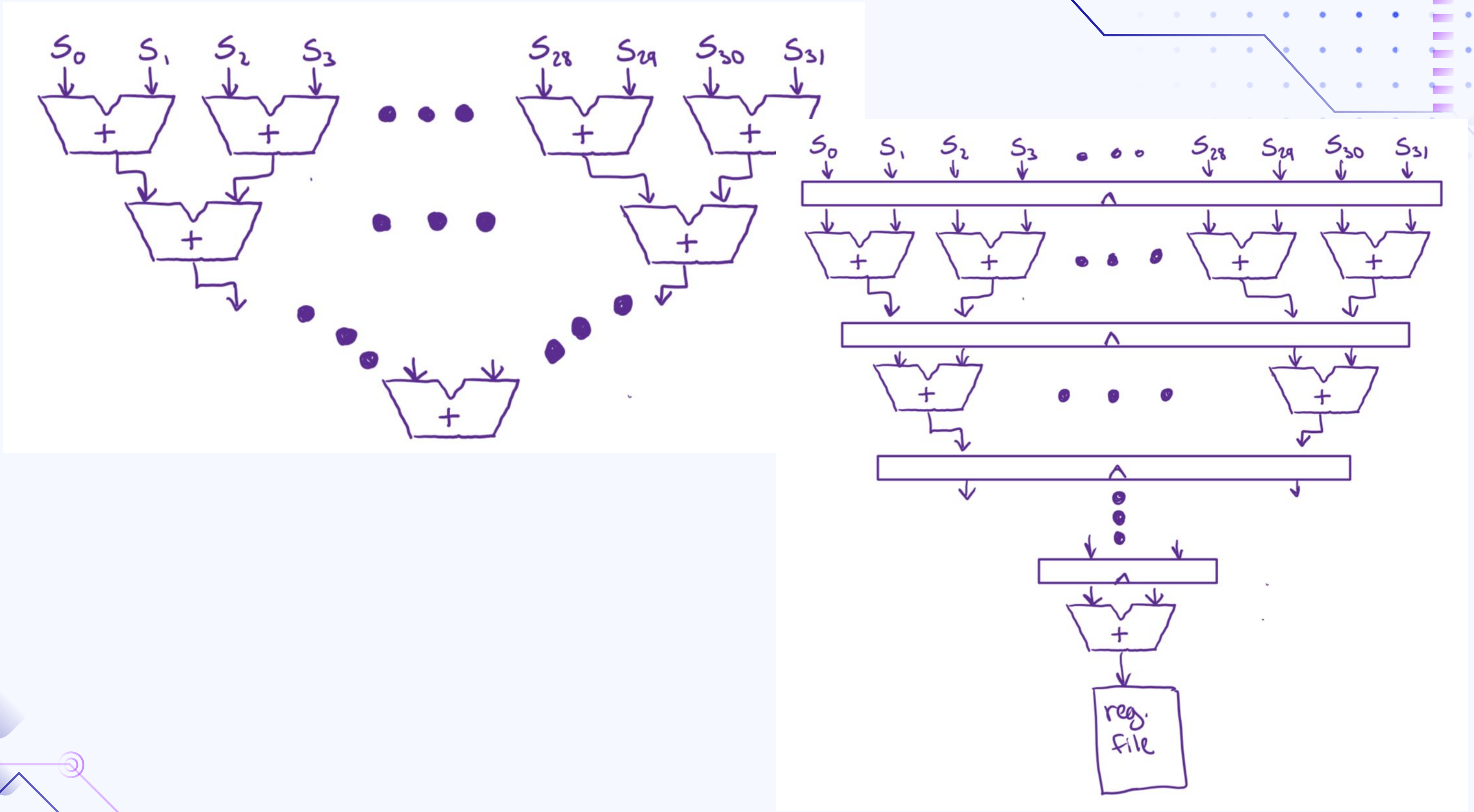
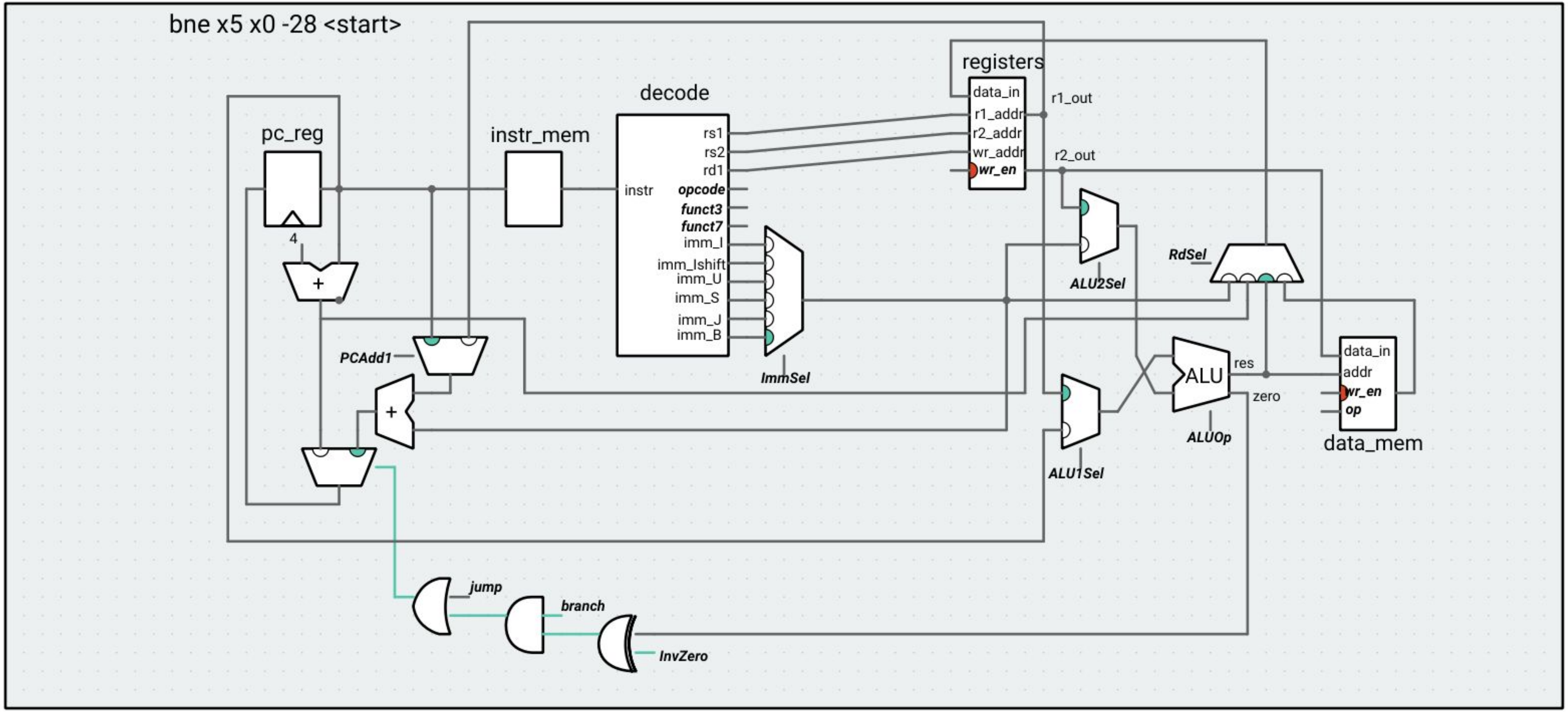
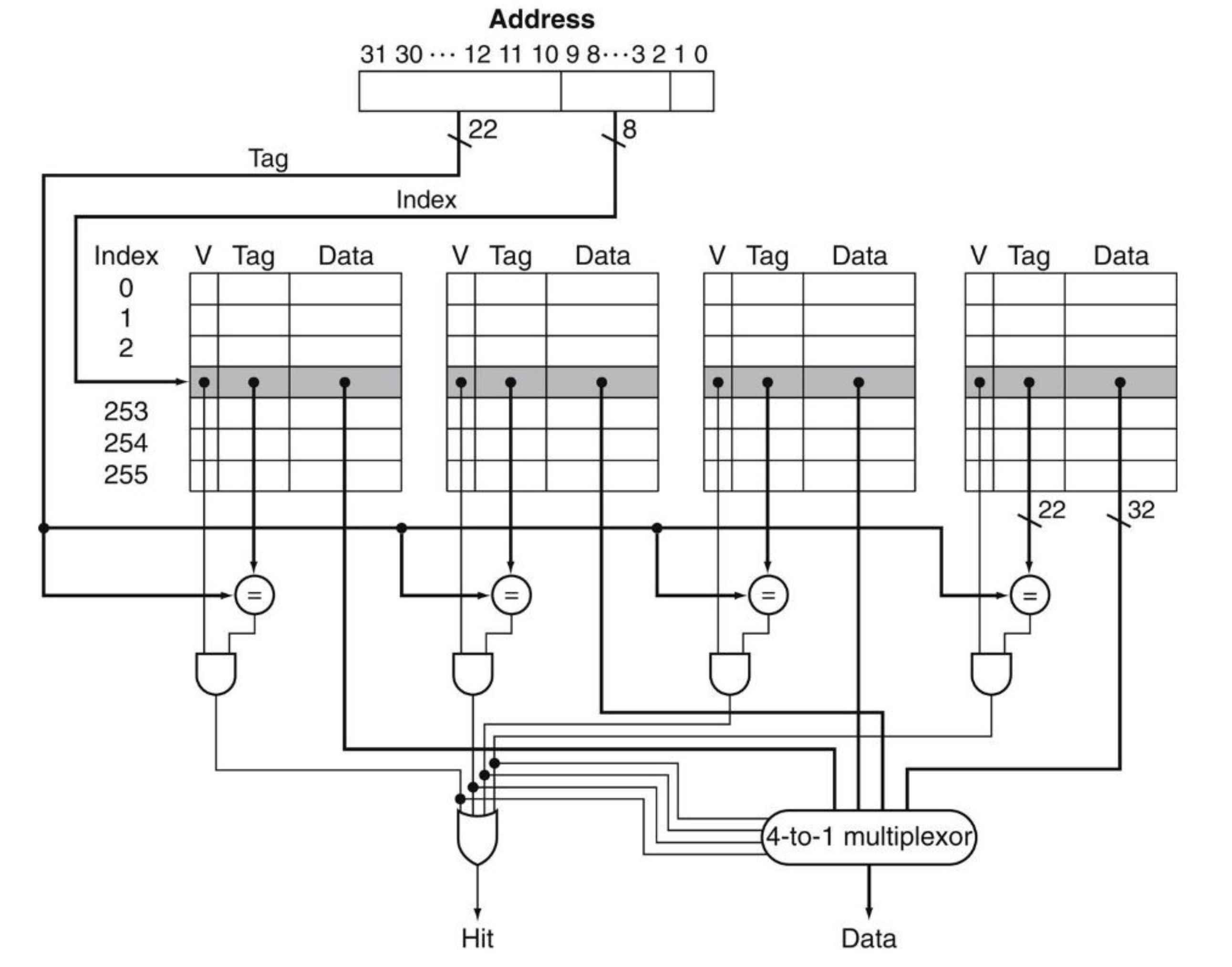
Instruction-level parallelism


S0
S1
S2
S3
S28
S29
S30
S31
+
S0
S1
S2
S3
S28
S29
S30
S31
+
reg. file

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




Address
31 30 ··· 12 11 10 9 8···3 2 1 0
22
8
Tag
Index
Index
0
1
2
253
254
255
V
Tag
Data
V
Tag
Data
V
Tag
Data
V
Tag
Data
=
=
=
=
22
32
4-to-1 multiplexor
Hit
Data

P&H fig. 5.18

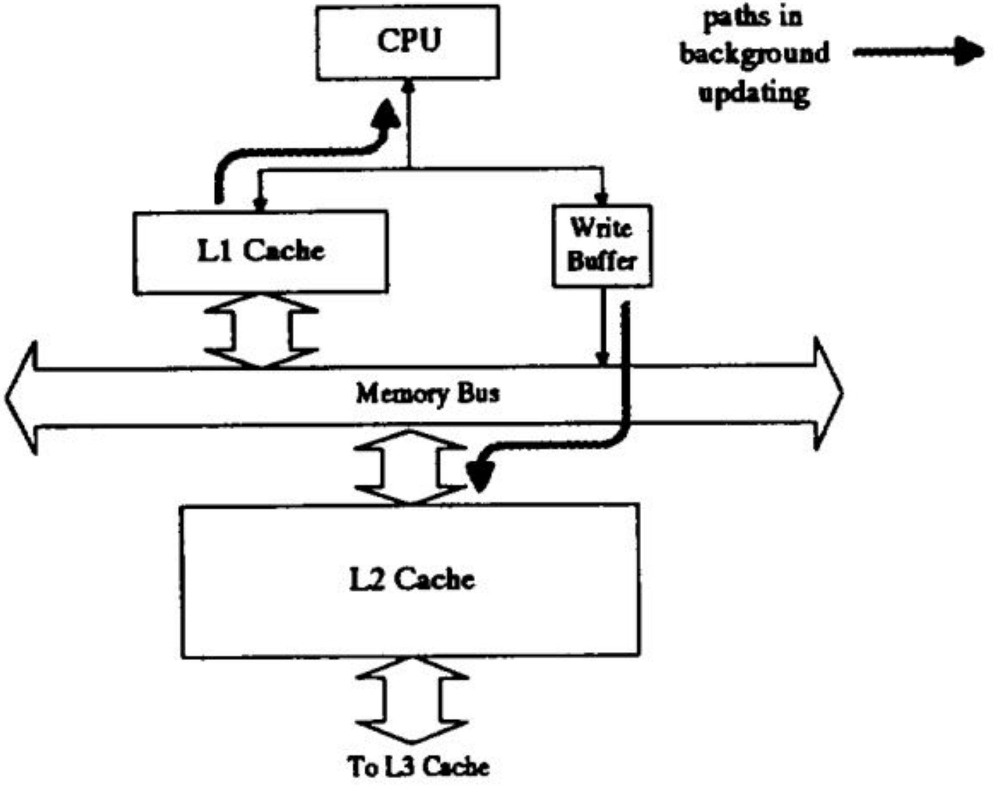


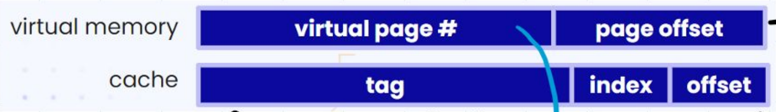
Figure 1: 2-Level Cache with Write Buffer

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.

VIPT

Sunday, February 25, 2024 4:41 PM

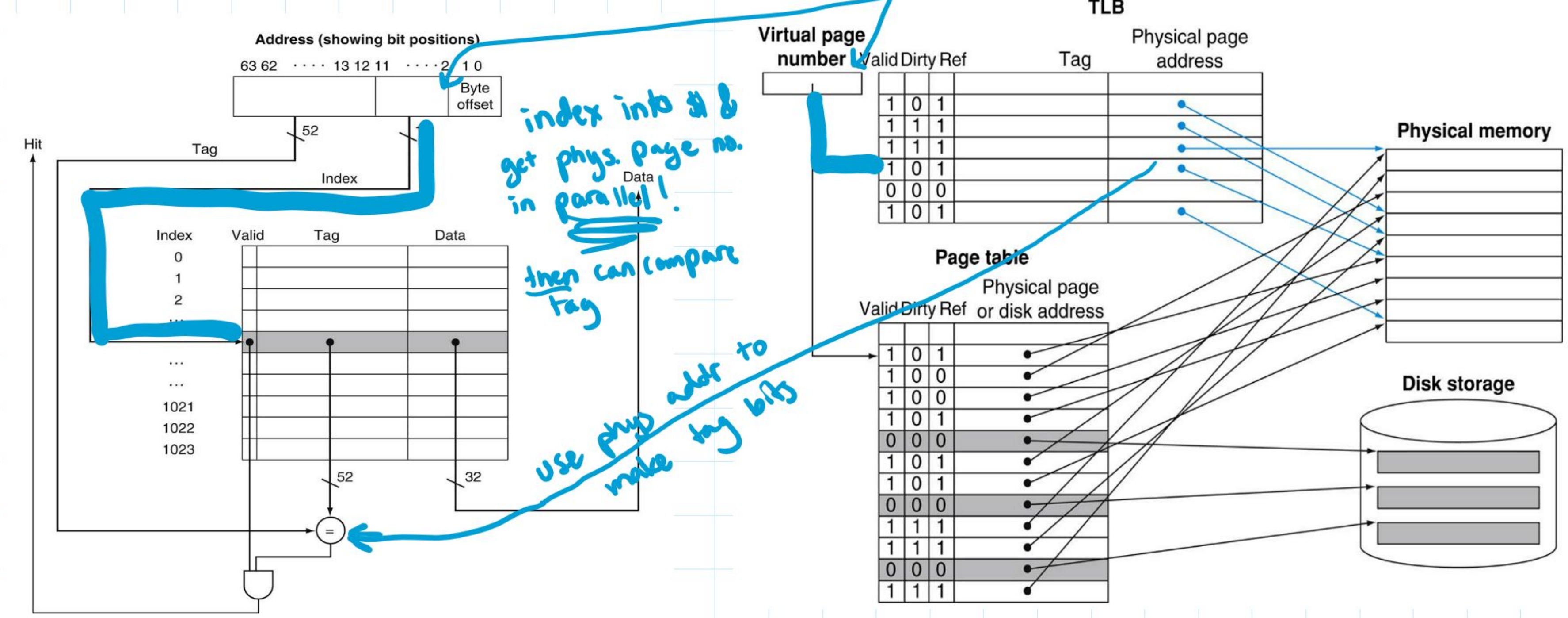
two ways to interpret an address:



same for v & p addrs

need phys. addr for this

if $ size is smaller than page size, these bits will live in the page offset

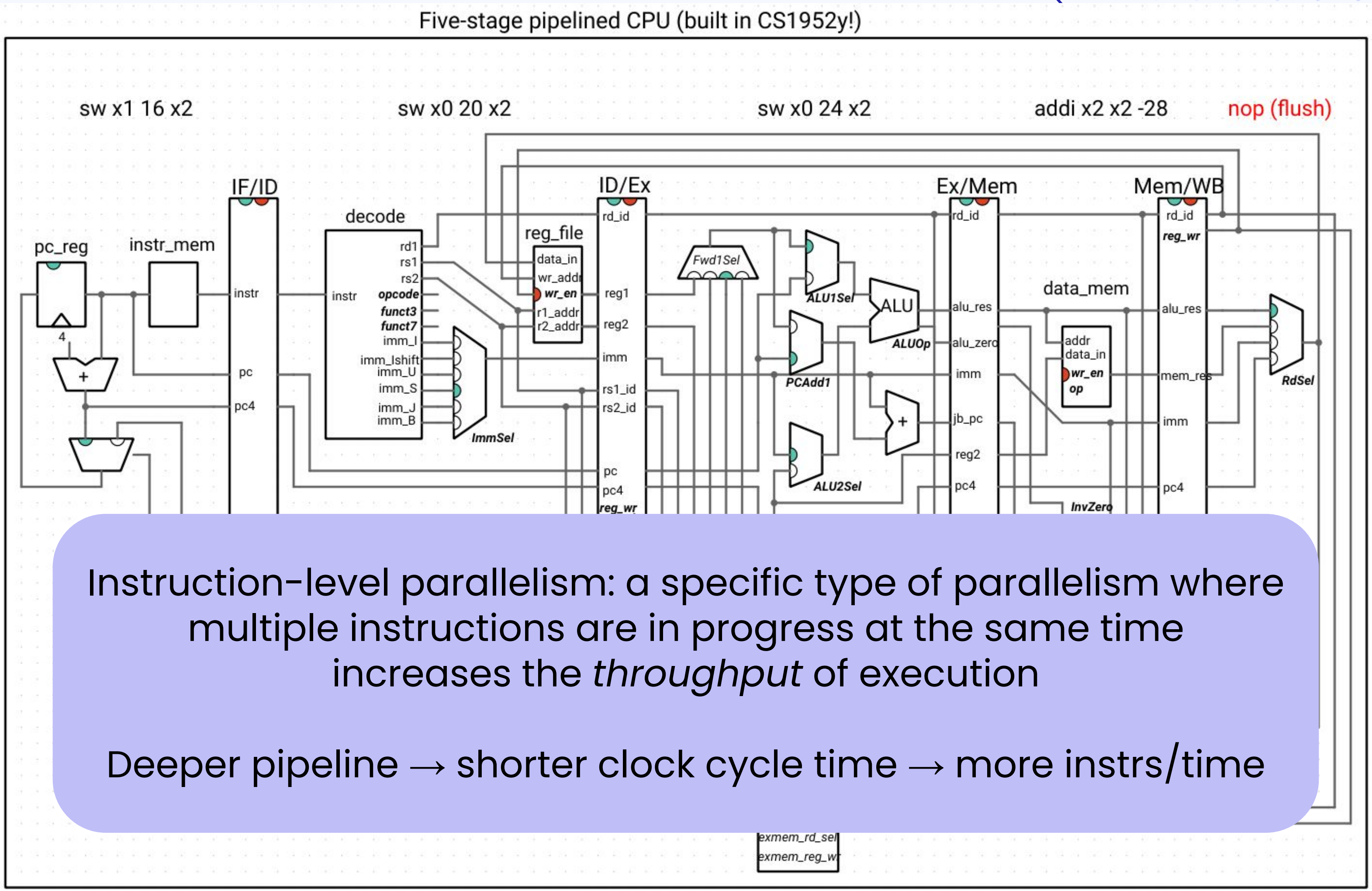


index into $ & get phys. page no. in parallel!

then can compare tag

use phys addr to make tag bits

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



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

? ? ?

What makes ILP challenging?

Basic blocks

```
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
```

Sequence of instructions between branches/jumps

Where’s the hazard?

```
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
```

Same result, lower CPI (1.44/1.35)

```
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
```

```
slli t2, t0, 2
add t3, a0, t2
add t4, a1, t2
lw t2, 0(t4)
lw t4, 0(t3)
addi t0, t0, 1
add t4, t4, t2
sw t4, 0(t3)
```

```
slli t2, t0, 2
add t3, a
add t4, a1,
lw t2, 0(t4)
lw t4, 0(t3)
sw t4, 0(t3
add t4, t
addi t0, t0, 1
```

We could have the compiler do this
or
We could have the CPU do this

Either way: how do we maintain correctness?

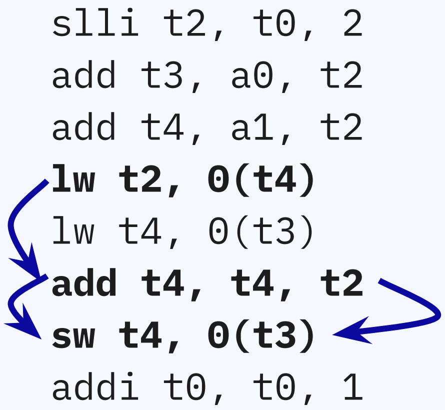
Data dependences (H&P 3.1.2.1)

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*



```
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
```

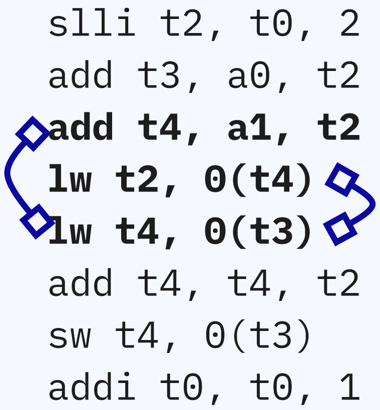
don’t reorder these!!!

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



```
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
```

also don't reorder these!!!

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

Another example

```
div t0, t1, t2
add t3, t0, t4
sw t3, 0(s0)
sub t4, t5, t6
mul t3, t5, t4
```

In-order pipeline will stall to allow div instruction to finish

Can we do better? What are the dependences?

Dynamic scheduling (OOO execution)

Allows executions to be rearranged at runtime (also called Out of Order, or OOO)

Advantages:

- Pipeline-agnostic (code written/compiled for one uarch can work efficiently on OOO uarch)
- Allows handling of dependences that can't be resolved by compiler
- Allows code to execute during delay (e.g. cache miss, div, floating point)

Much more complex! But worth it in modern systems

Details of OOO execution

In-order issue

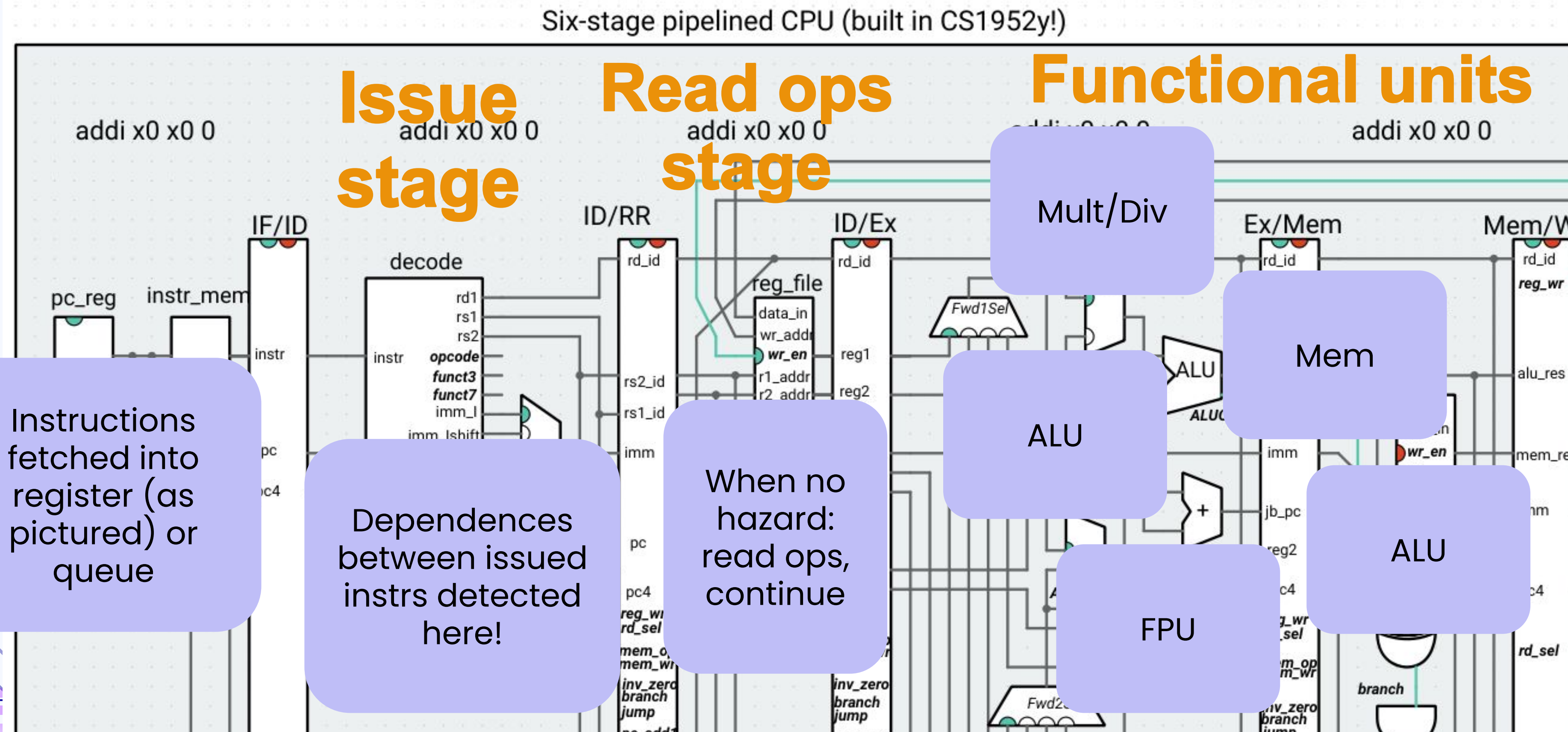
Potentially out-of-order completion

Between the time when an instruction is issued and when it completes, it is *in execution (in flight)*

- Multiple instructions can be in flight at the same time, either due to multiple functional units (ALUs, FPUs, etc) or due to pipelining

How? Split up ID stage

(*we'll redraw this picture later)



Six-stage pipelined CPU (built in CS1952y!)

Issue stage

Read ops stage

Functional units

Instructions fetched into register (as pictured) or queue

Dependences between issued instrs detected here!

When no hazard: read ops, continue

Mult/Div

ALU

Mem

ALU

FPU

Early OOO: scoreboarding

Fascinating history of CDC 6600

Keep track of dependences between in-flight instructions and fetched instructions using dependency matrices

Issue a fetched instruction only when no dependences arise

But this is really restrictive: we can do better

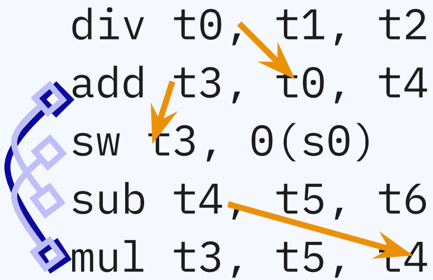
By Jitze Couperus - Flickr: Supercomputer - The Beginnings, CC BY 2.0, link

Tomasulo’s algorithm intuition

Developed by Robert Tomasulo for IBM 360/91

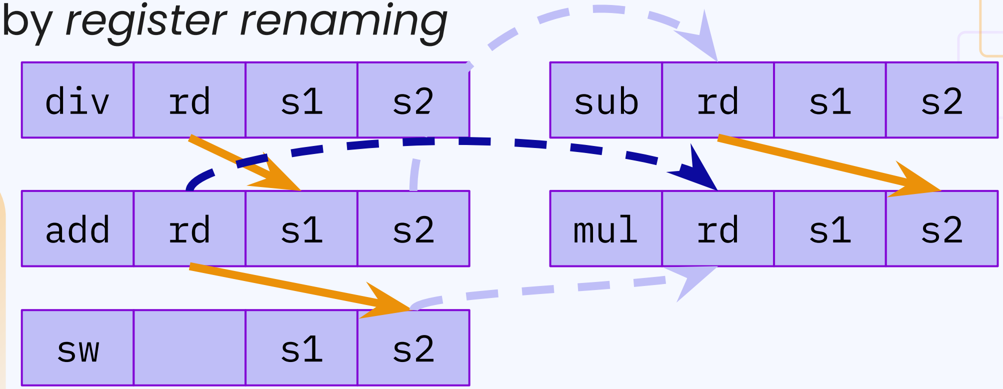
Minimize RAW hazards by tracking data dependences and reordering

Minimize WAR and WAW hazards by *register renaming*



```
div t0, t1, t2
add t3, t0, t4
sw t3, 0(s0)
sub t4, t5, t6
mul t3, t5, t4
```

What if the hardware could track dependences and temporarily store results?



div	rd	s1	s2		sub	rd	s1	s2
add	rd	s1	s2		mul	rd	s1	s2
sw		s1	s2					

now the div might have enough time to write to t0 before add needs it!