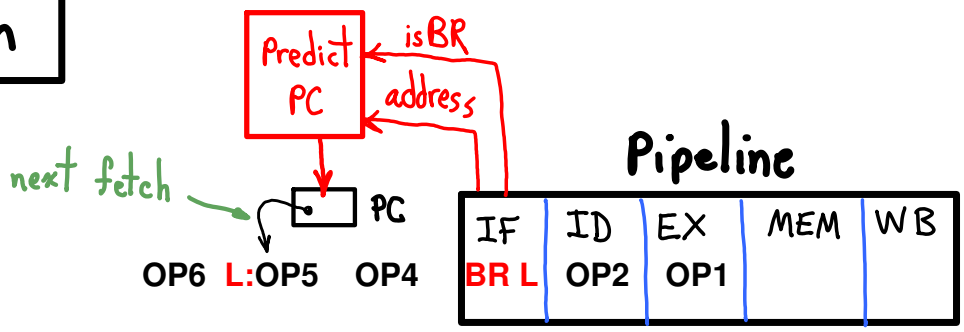
Branch Prediction

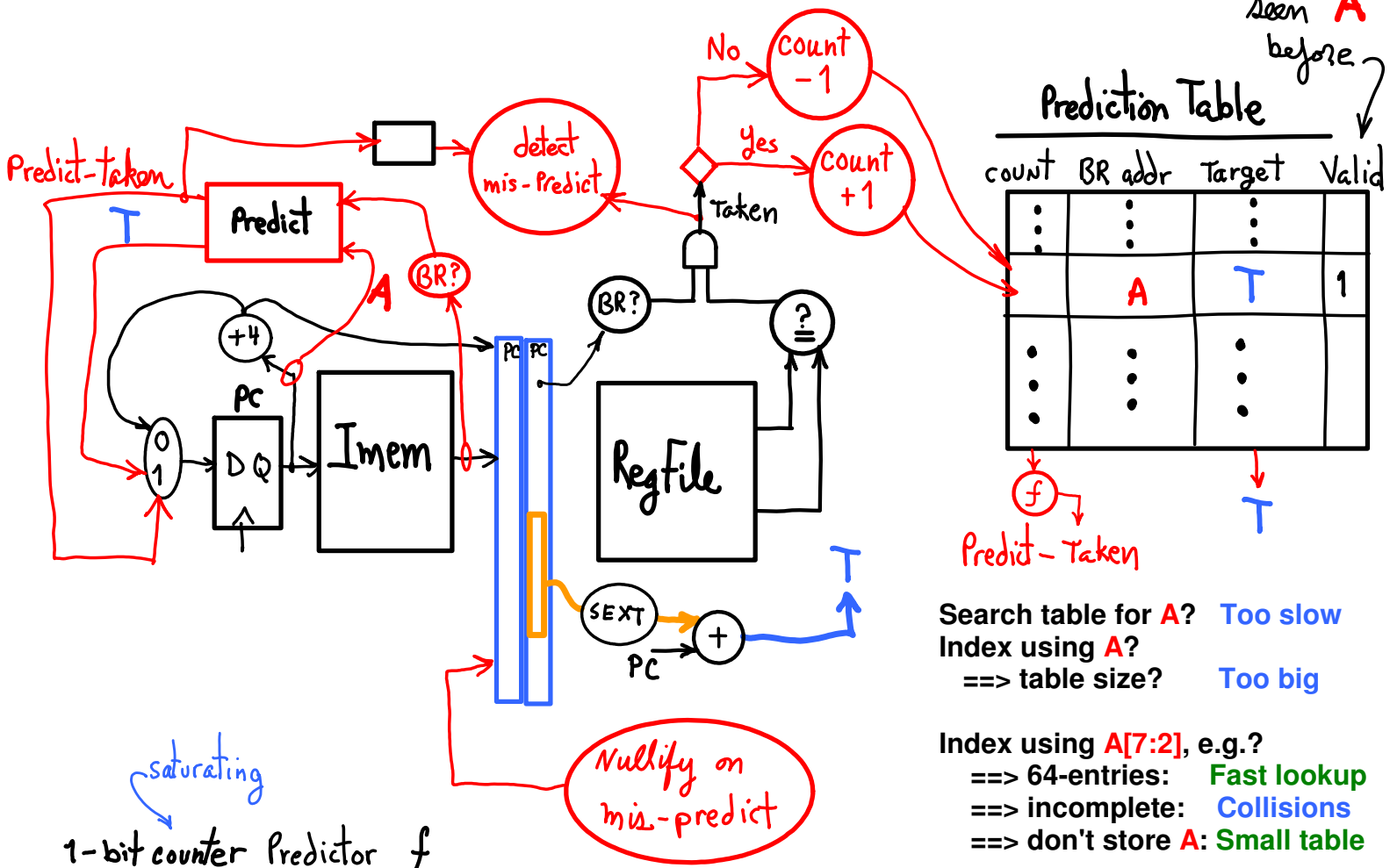
Address	instructions
	OP1
	OP2
A	BR T
	OP4
	T:
	OP5



Set PC according to prediction – need target address T

(1st encounter: store T, valid ← 1)

(2nd occurrence: use T from table)



Search table for A? Too slow

Index using A?

==> table size? Too big

Index using A[7:2], e.g.?

==> 64-entries: Fast lookup

==> incomplete: Collisions

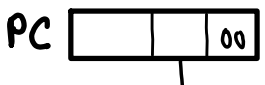
==> don't store A: Small table

↳ ignore collisions

saturating

1-bit counter Predictor f

$$\text{Predicted-Taken} = \begin{cases} 1, & \text{count} > 0 \\ 0, & \text{count} \le 0 \end{cases}$$



↳ Table index

count == 0

entry into loop:

Predict-taken = 0, WRONG (nullify instr.), count <== 1

last iteration:

Predict-taken = 1, WRONG (nullify instr.), count <== 0

Guaranteed to mis-Predict twice per loop entry

n-bit Predictor

Predicted-Taken = $\begin{cases} 1, & count > max/2 \\ 0, & count < max/2 \end{cases}$

COUNT
Saturating
No rollover/under

first entry: count <== max/2 (+e)
next k BR's:
Predict-taken = 1, count++
last iteration:
Predict-taken = 1, WRONG (nullify instr.), count--
next entry into loop:
Predict-taken = 1, count++

How many bits for good results?
→ depends on workload.
→ experiment

How big a table?

2-bit, 4k → 0 – 20% missed (SPEC)

what about this?

correlated BRs

```
if (x)
    y ← 0
else
    y ← 1
if (y)
```

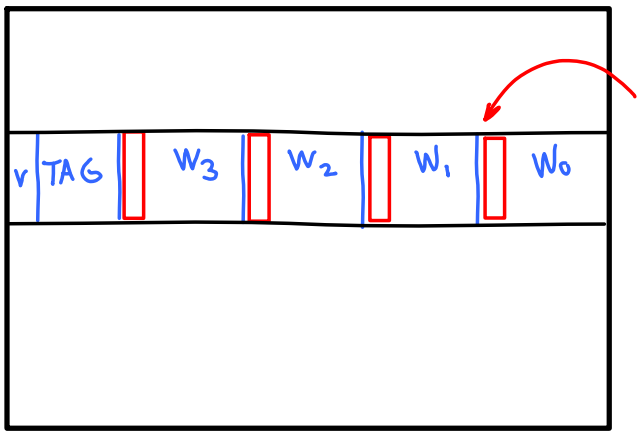
⇒ Context-based prediction

[010010] = history of recent branches
shift in latest BR-Taken bit

⇒ BR pattern table
STore patterns in table w/counts

Look up pattern to get prediction
Adjust count for that pattern.

Cache



Put prediction bits in cache for each word when loading cache block.

On fetch, no lookup
⇒ of prediction bits needed.
Update BR-taken bits in cache? Load Table on replacement?

Some other influences to consider:
--- Multiple threads of execution? Multiple processes?
--- Different states.
--- Different prediction tables?

Scheduling

```
SW $1, 16($8)
LW $1, 4($3)
```

Looking for independent instructions. Schedule to avoid stalls.

Speculate that SW and LW are independent. Reorder freely.

HW: Check addresses at runtime: where they independent? If so, fix.

SW: Insert code to check, and code to do the fixing up if wrong.

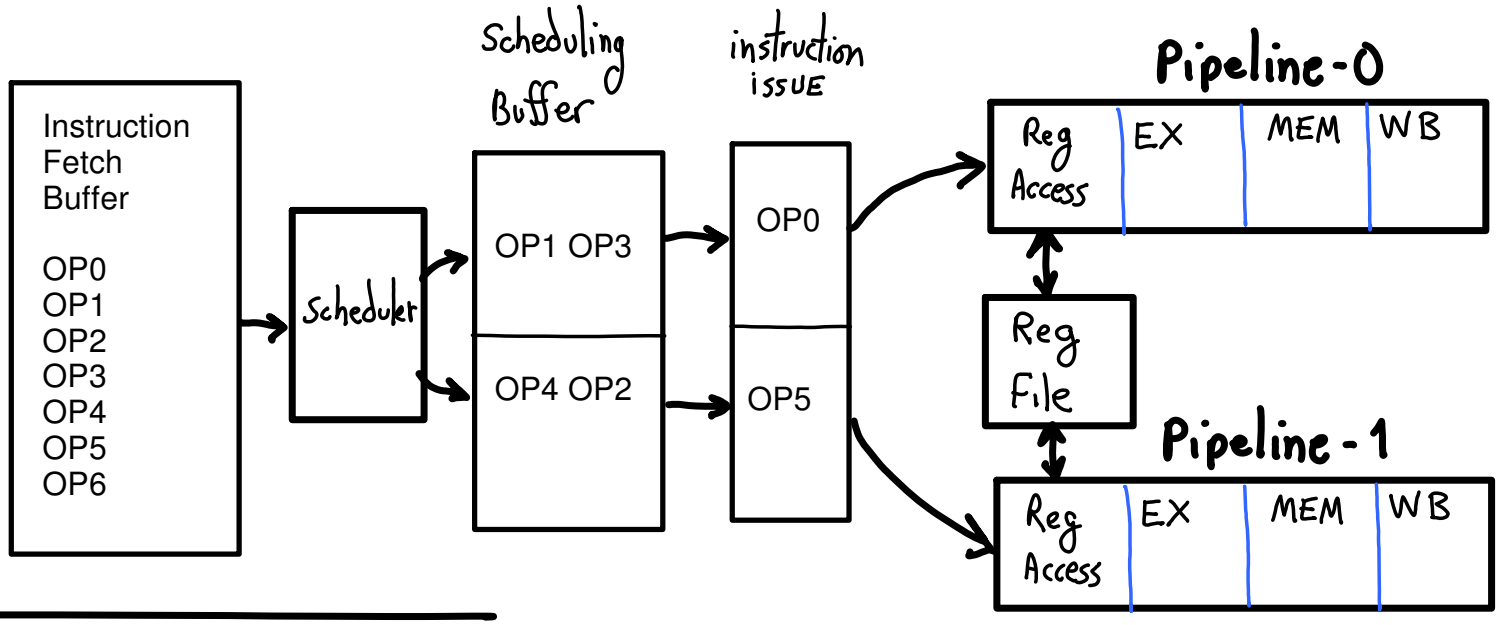
Reorder code to avoid Load-Use stalls.

Reorder code to avoid BR bubbles.

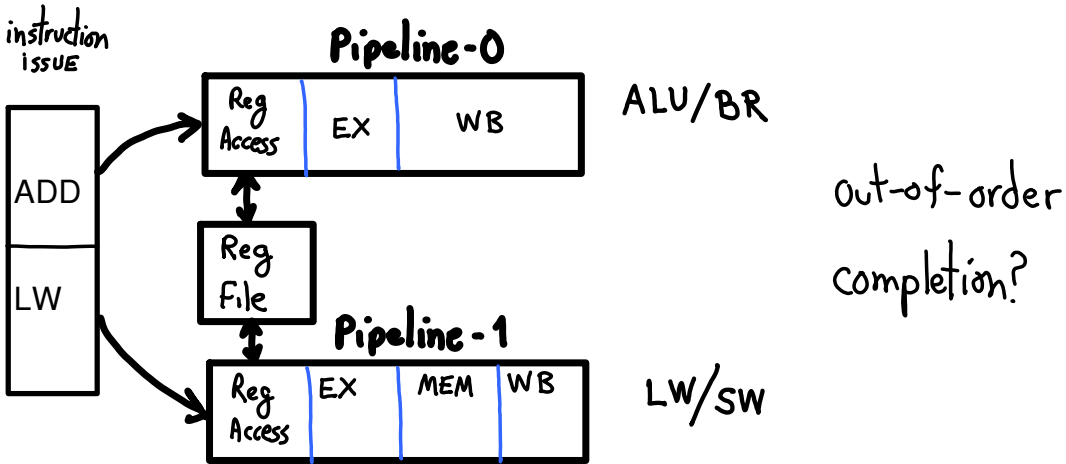
Reorder code into delayed BR slots.

Rename registers to avoid false dependencies.

Multi-issue Scheduling



hetero-Pipelines



Loop unrolling

```
L:
   LW     $1, 0( $2 )
   ADD    $1, $1, $3
   SW     $1, 0( $2 )
   ADDi   $2, $2, -4
   BNE    $2, $0, L
```

```
L:
$1 <== A[ $2 ]
$1 <==  $3 + $1
A[ $2 ] <== $1
$2--
until( $2 == 0 )
```

Unrolled loop shows "naming dependency", aka, "anti-dependency"

```
$1 <== A[ $2 ]
$1 <==  $3 + $1
A[ $2 ]  <== $1
$2--
$1 <== A[ $2 ]
$1 +=  $3
A[ $2 ] <== $1
$2--    ...
```

$2==8

independent

$2==4

Data dependencies exist ($1) between three instructions. Cannot schedule out-of-order nor in parallel. (Also $2 dependency across loop iterations.)

Independent except that same register is used ($1). Compiler can rename to exploit existing ILP.

Scheduling

index pre-calculated

renamed $1

```
$1 <== A[ $2 ]
$4 <== A[ $2 - 4 ]
$1 <==  $3 + $1
$4 <==  $3 + $4
...
$2 <== $2 - 8
Loop
```

new loop control

```
L:
   LW     $1, 0( $2 )
   LW     $4, -4( $2 )
   ADD    $1, $1, $3
   ADD    $4, $4, $3
   SW     $1, 0( $2 )
   SW     $4, -4( $2 )
   ADDi   $2, $2, -8
   BNE    $2, 0, L
```

reads grouped

operations w/o load-use stalls

writes grouped

Unrolling for static-issue Multi-pipeline

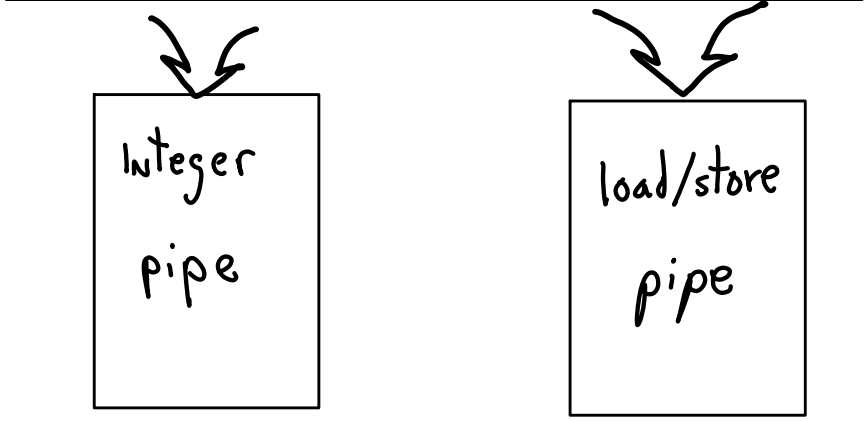
```
L:
    LW   $1, 0($2)
    ADDU $1, $1, $3
    ADDI $2, $2, -4
    BNE  $2, $0, L
```

Unrolled

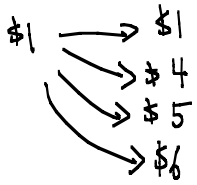
```
L:  addi $2, $2, -16     ——   LW  $1, 0($2)
    nop                  ——   LW  $4, 12($2)
    addu $1, $1, $3      ——   LW  $5, 8($2)
    addu $4, $4, $3      ——   LW  $6, 4($2)
    addu $5, $5, $3      ——   SW  $1, 16($2)
    addu $6, $6, $3      ——   SW  $4, 12($2)
    nop                  ——   SW  $5, 8($2)
    bne  $2, $0, L       ——   SW  $6, 4($2)
```

uses old value of $2

uses forwarded value of $2



Register Renaming

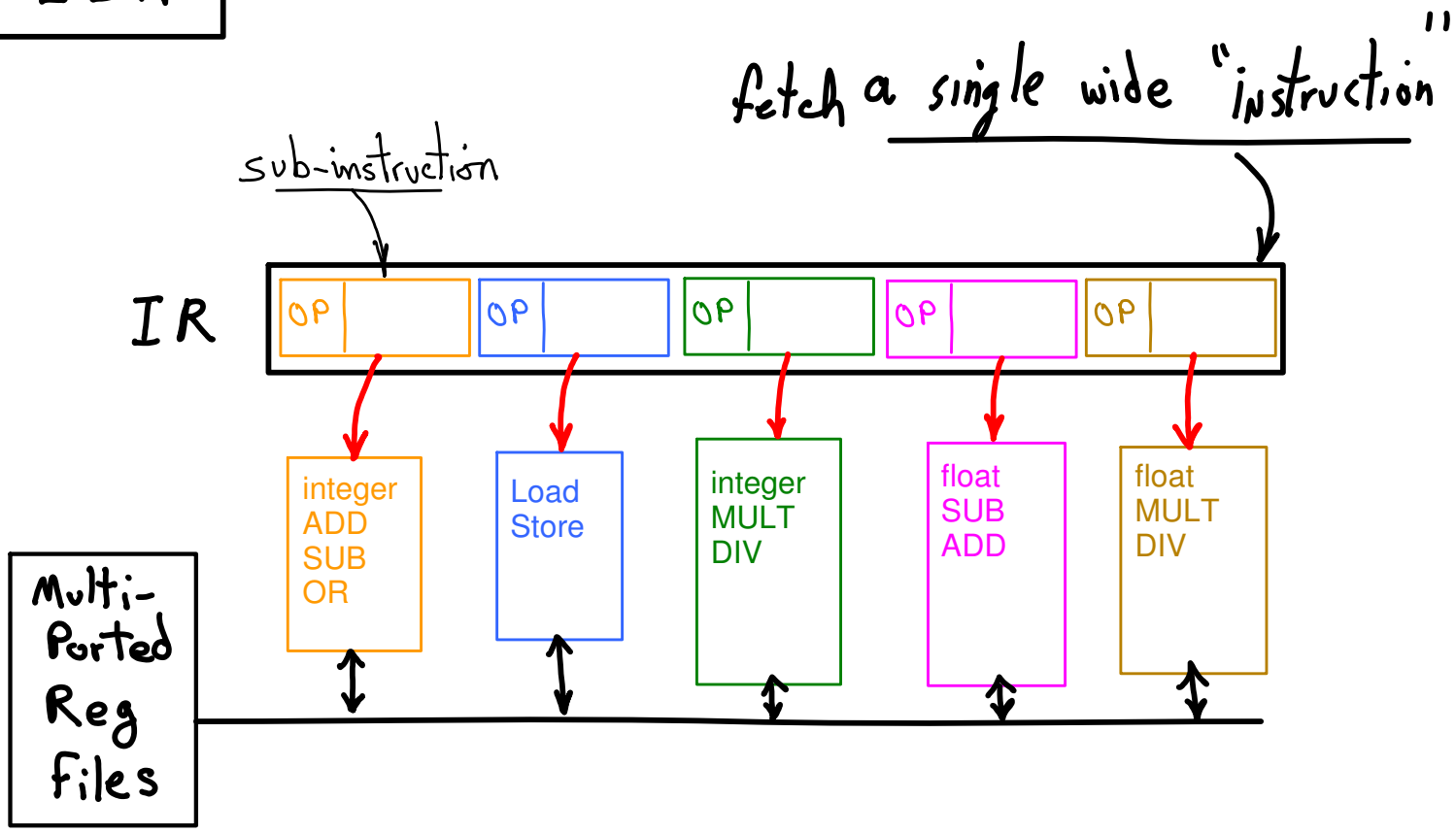


CPI < 1

Static issue,
Instruction pairs issued together

Compiler scheduling

VLIW



Can we fill w/ useful operations?

How do we discover ILP to build VLIW instructions w/ few NOPs?

How big is a program?

Multiple functional units

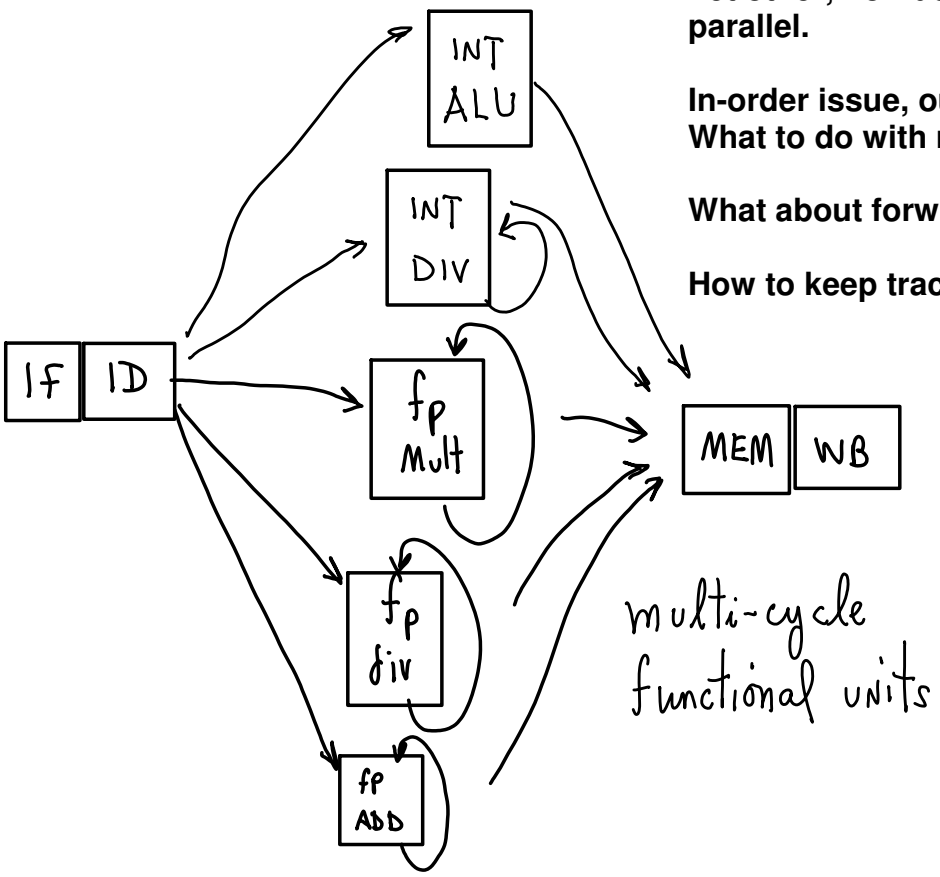
Some operations take a long time (FP divide, e.g.)

Let other, non-dependent instructions flow by in parallel.

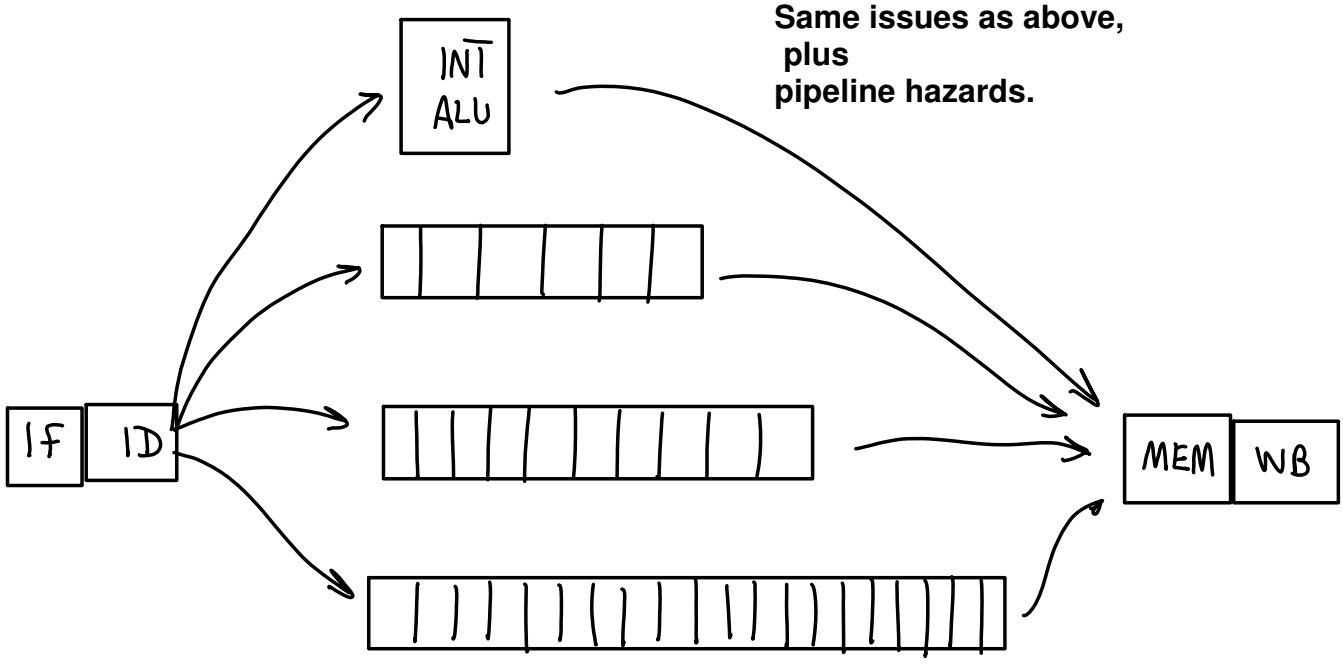
In-order issue, out-of-order completion.
What to do with multiple completions in same cycle?

What about forwarding between units?

How to keep track of dependencies?

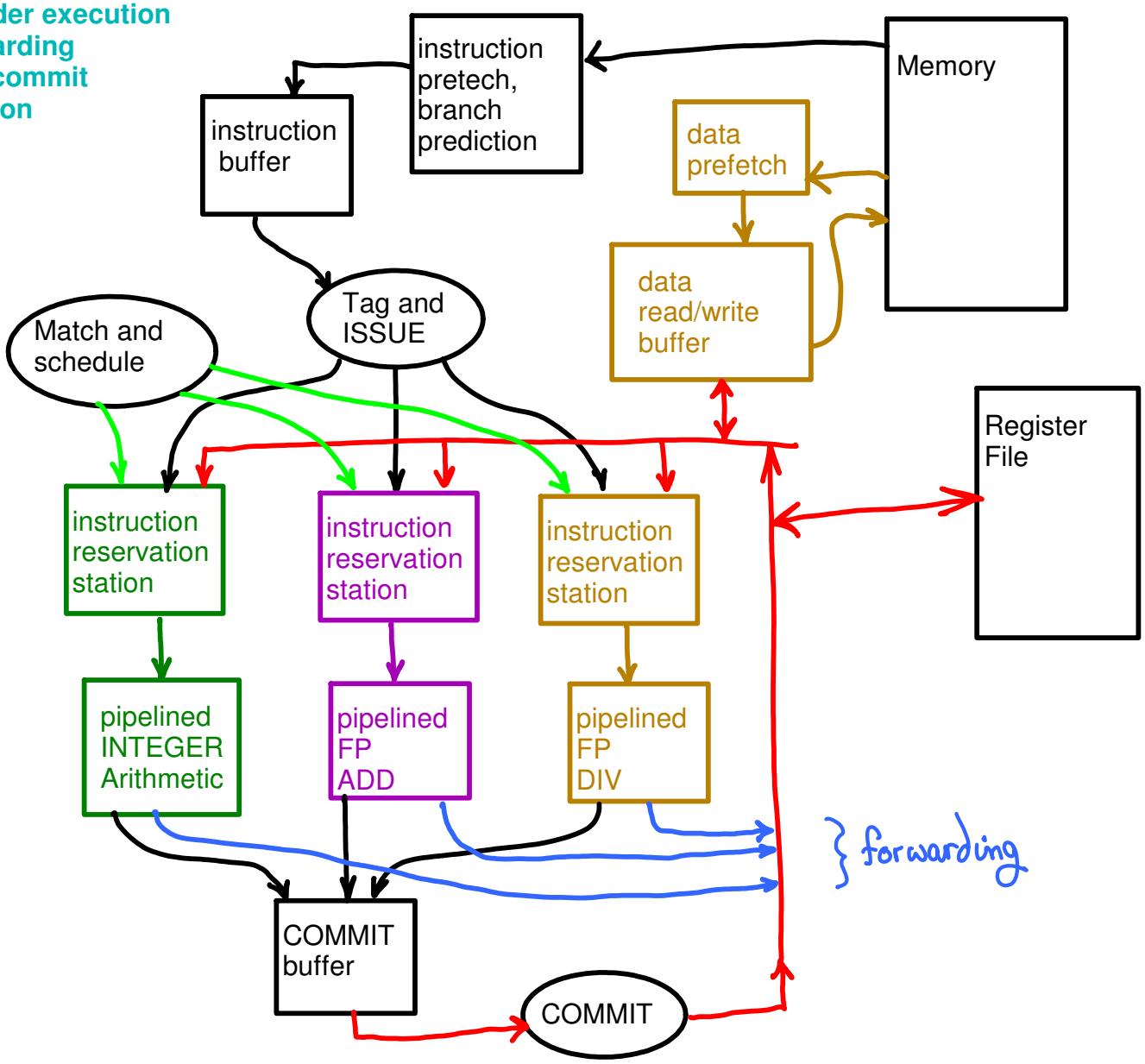


Multiple pipelined functional units.
Same issues as above,
plus
pipeline hazards.



Could have combination
pipelined
and
non-pipelined, multi-cycle units.

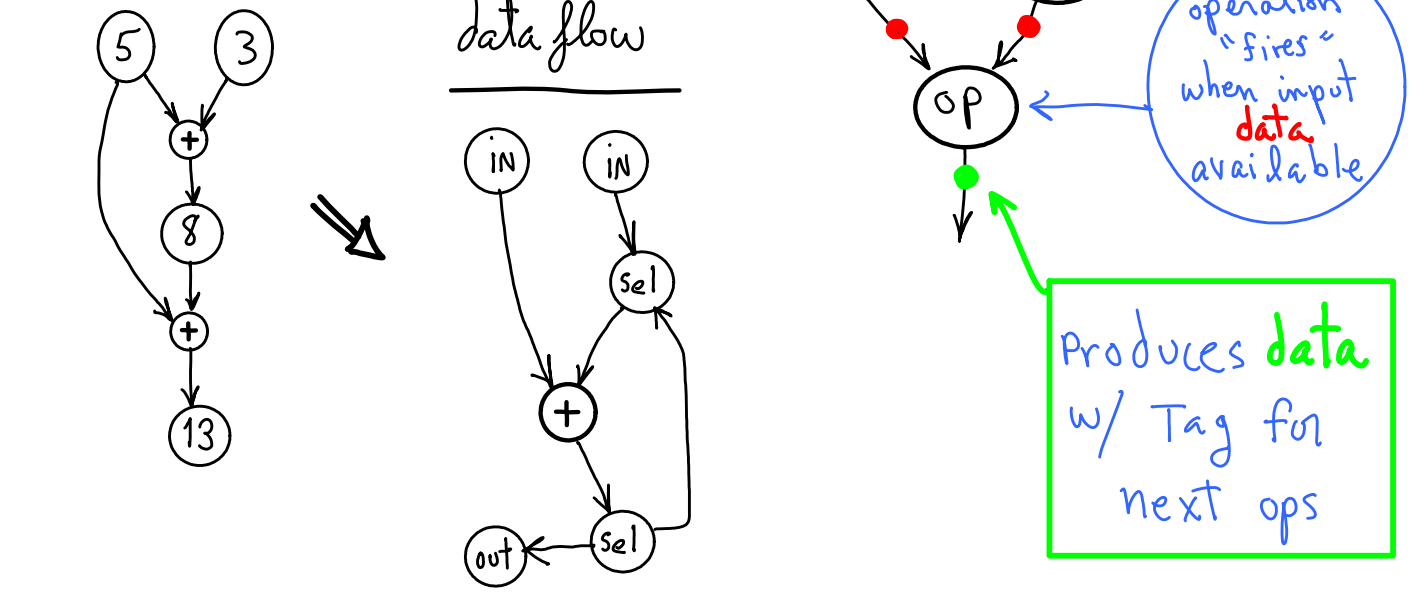
Super-Scalar OOO
speculation
dependency analysis
dataflow scheduling
out-of order execution
data fowarding
in-order commit
nullification



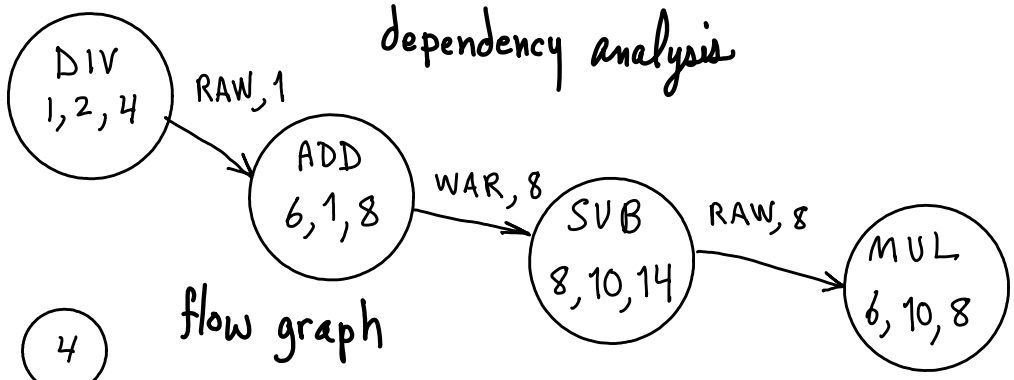
In general ILP:

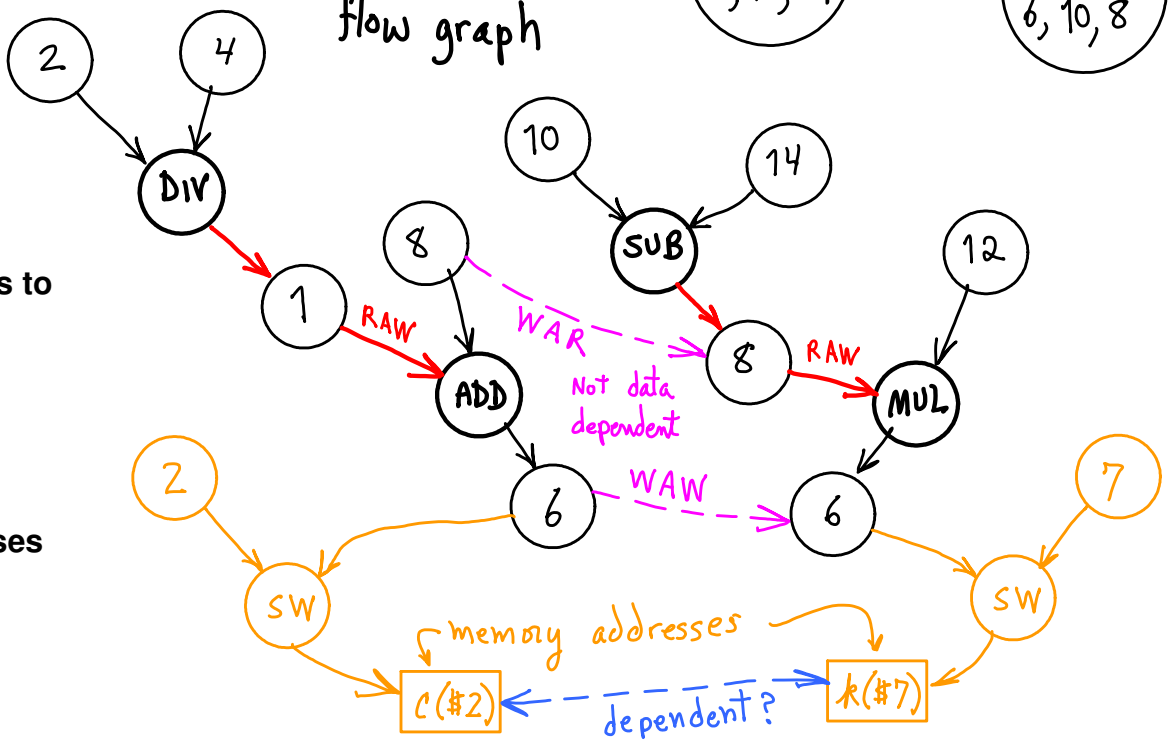
--- Dataflow approximation
--- discover instructions to run in parallel (arch, compiler, programmer)
--- handle data dependencies
--- hide latencies due to control dependencies

Dataflow, dependency analysis



DIV $1, $2, $4
ADD $6, $1, $8
SUB $8, $10, $14
MUL $6, $10, $8





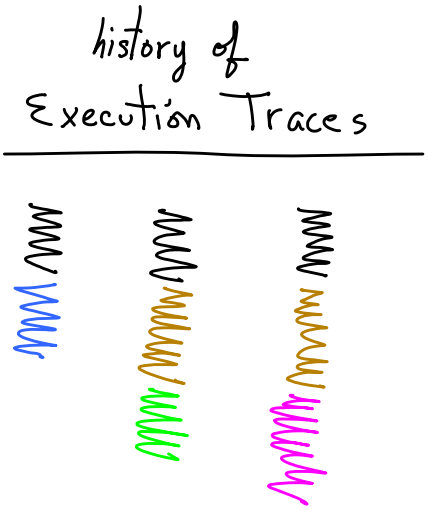
Tagging to match register content w/ operation.

Renaming registers to avoid false dependencies.

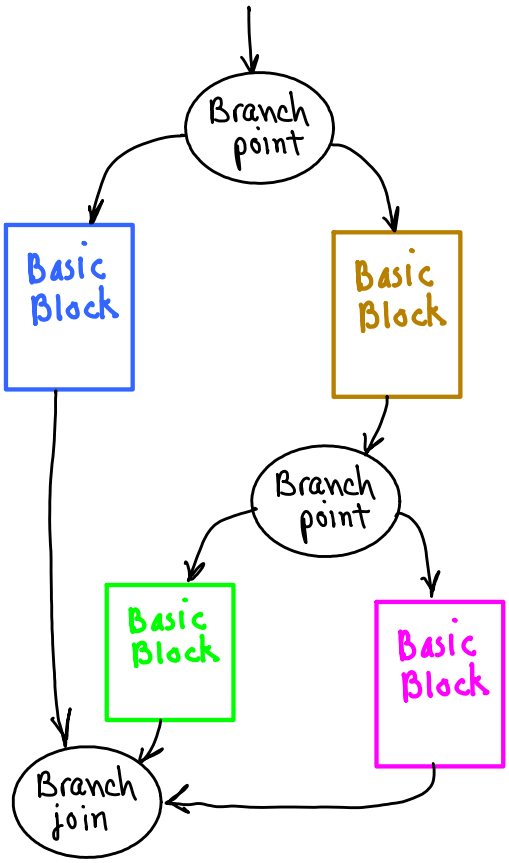
Forwarding per dataflow.

Evaluating addresses to check on dependencies.

Trace speculation, scheduling



compose code from traces



HW Speculation

--- Keep track of traces (cache).

--- Speculate on taken trace (nullify as needed).

--- Execute multiple traces in parallel (nullify as needed).

SW Speculation

--- Rewrite code with multiple traces.

--- Add code to undo bad speculation.

--- Profile benchmarks to pick most probable trace as first executed.