Speculation in the pipeline

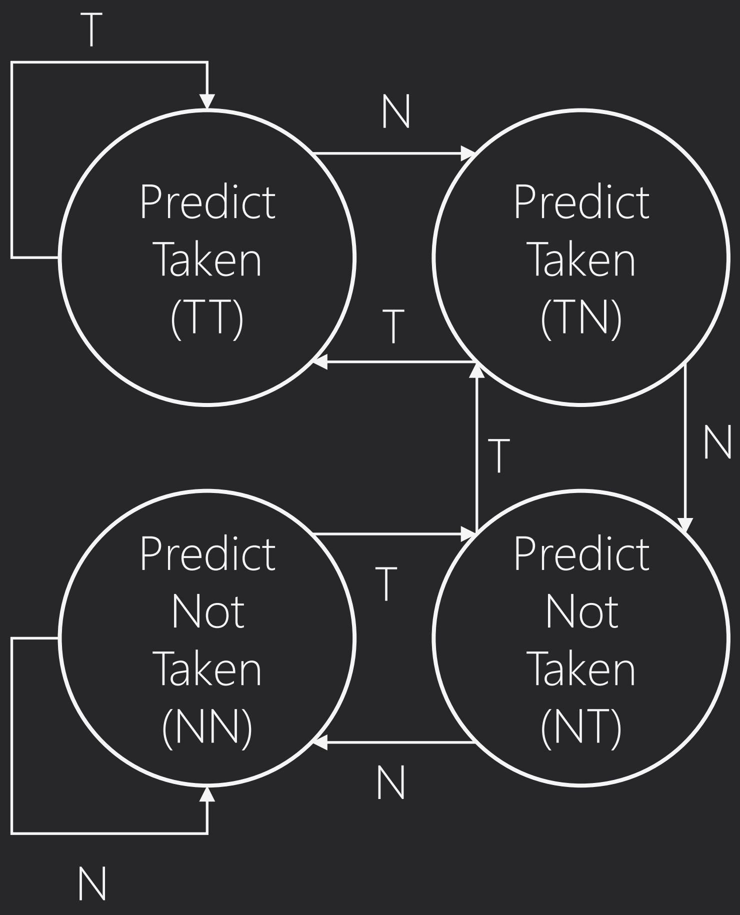
Dynamic branch predictors and speculative execution

The two-bit saturating counter

- Store two bits rather than 1

- Two bits differentiate between four states
  - TT: Strongly taken
  - TN: Weakly taken
  - NT: Weakly not taken
  - NN: Strongly not taken

- Starting state: NN



The nested loop (2-bit saturating counters)

```
for(i =0; i < n; i++)
     for(j = 0; j < m; j++)
          // loop body
```

- Suppose n is very large (100)
- Suppose m is small (3)
- Let Taken (T) mean "enter loop body"

Value of i	Value of j	State / Prediction	Outcome
0	0	NN	T
0	1	NT	T
0	2	TN	T
0	3	TT	N
1	0	TN	T
1	1	TT	T
1	2	TT	T
1	3	TT	N
2	0	TN	T
2	1	TT	T
2	2	TT	T
2	3	TT	N

Branch prediction

Approaches (data structures, mechanisms) for dynamic branch predictors

Where is the branch predictor data structure?

- One option is to store it in the instruction cache

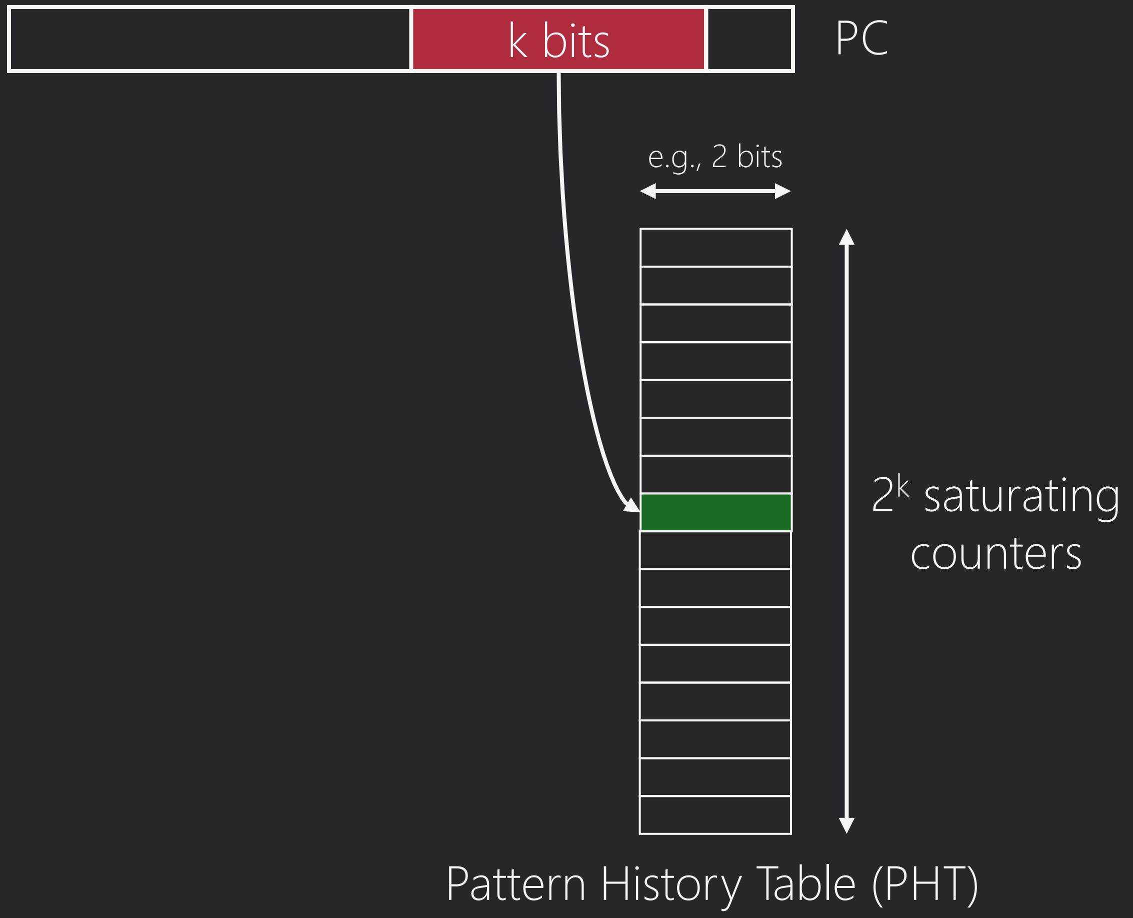
- Examples
  - Alpha 21264 used one counter per instruction
  - AMD K5 used one counter per cache line
    - Recall variable-length instructions with x86

V	*Tag*	*Data*	Counter (2 bits)
			🟩
			🟩
			🟩
			🟩
			🟩
			🟩
			🟩
			🟩

What is a bimodal branch predictor?

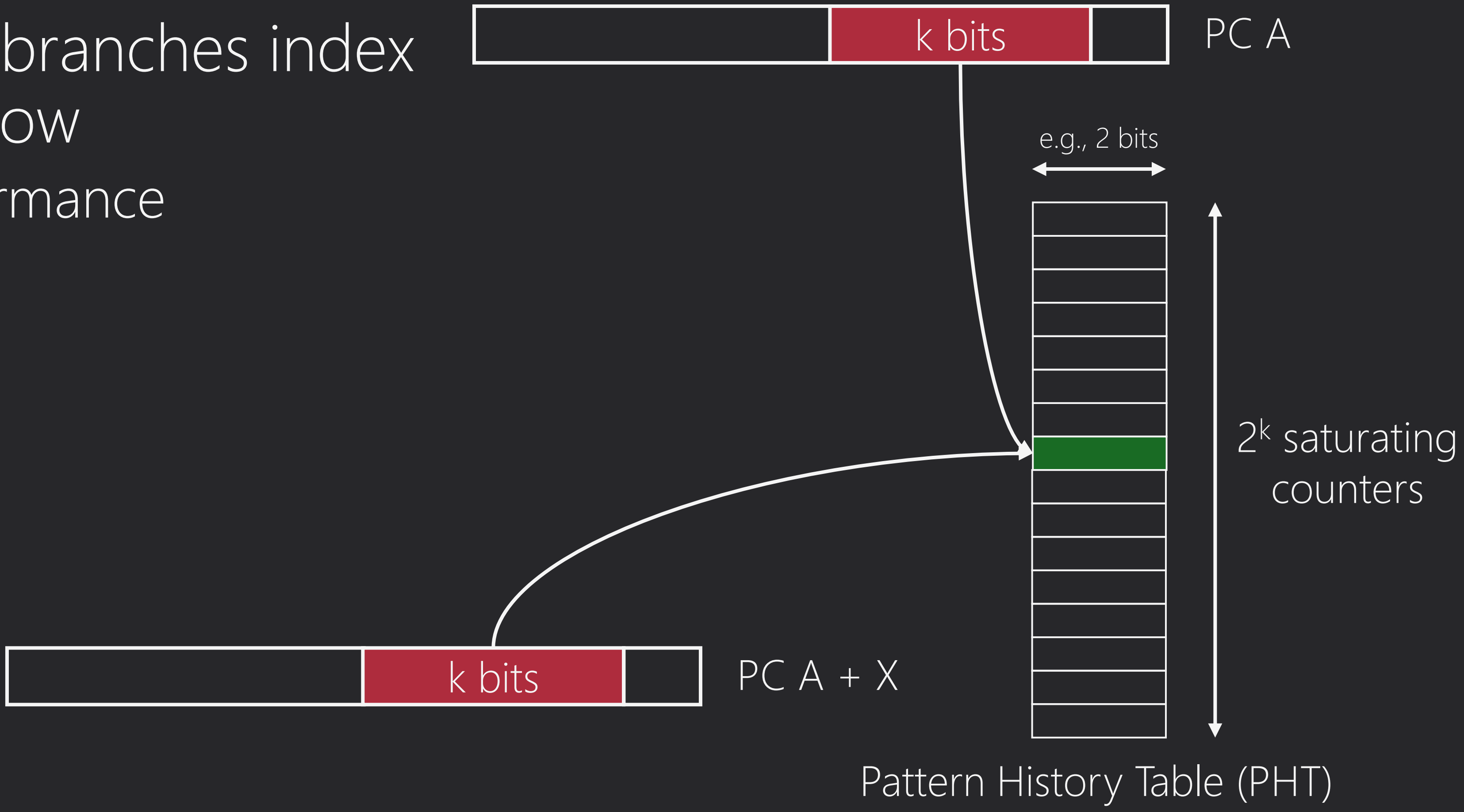
- Separate structure: PHT
  - Indexed by k bits from PC
  - Has 2^k rows
  - Each row is a saturating counter

- Examples
  - MIPS R10000 (512 counters)
  - IBM PowerPC 620 (2,048 counters)



What is aliasing?

- When two different branches index into the same PHT row
  - Can degrade performance



Can *other* branches help with prediction?

```
if a == 4:
    a = 0
if b == 4:
    b == 0

if a != b:
    # more code
```

```
        ADDI r3, r1, -4
        BNEZ r3, SKIP_IF1    # b1
        ADD r1, zero, zero
SKIP_IF1:
        ADDI r3, r2, -4
        BNEZ r3, SKIP_IF2    # b2
        ADD r2, zero, zero
SKIP_IF2:
        SUB r3, r1, r2
        BEQZ r3, SKIP_IF3    # b3
```

How are b1, b2, and b3 correlated?

- When neither b1 nor b2 are taken
  - a = 0, b = 0
  - b3 is taken (since a == b)

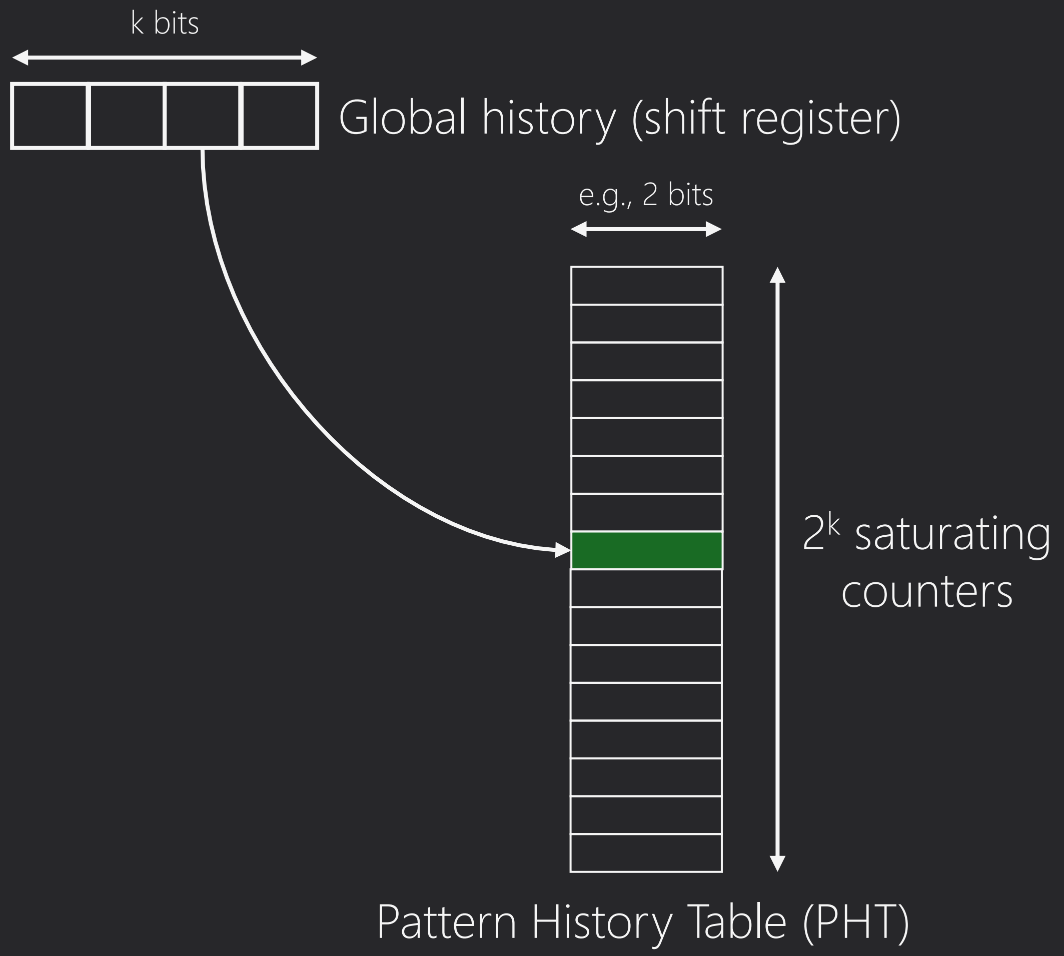
- Consider the "global" history:

b1	b2	b3
N	N	T
N	T	?
T	N	?
T	T	?

```
        ADDI r3, r1, -4
        BNEZ r3, SKIP_IF1    # b1
        ADD r1, zero, zero
SKIP_IF1:
        ADDI r3, r2, -4
        BNEZ r3, SKIP_IF2    # b2
        ADD r2, zero, zero
SKIP_IF2:
        SUB r3, r1, r2
        BEQZ r3, SKIP_IF3    # b3
```

What is the correlator predictor*? *with global history register

- The history register is a shift register
  - Insert (at right) whether branch is taken
  - Stores the most recent k branch outcomes
- Use the global history to index into PHT


k bits
Global history (shift register)
e.g., 2 bits
2^k saturating counters
Pattern History Table (PHT)

The nested loop (correlated predictor) (1)

```
for(i =0; i < n; i++)
     for(j = 0; j < m; j++)
          // loop body
```

- Suppose n is very large (100)
- Suppose m is small (3)
- Let Taken (T) mean “enter loop body”

- Use 2 bits for global history register (GHR)
- Use 1-bit “counter” for each row in PHT

		GHR				
i	j	NN	NT	TN	TT	Outcome
0	0	N	N	N	N	T
0	1					T
0	2					T
0	3					N
1	0					T
1	1					T
1	2					T
1	3					N
2	0					T
2	1					T
2	2					T
2	3					N

The nested loop (correlated predictor) (4)

```
for(i =0; i < n; i++)
     for(j = 0; j < m; j++)
          // loop body
```

- Suppose n is very large (100)
- Suppose m is small (3)
- Let Taken (T) mean "enter loop body"

- Use 2 bits for global history register (GHR)
- Use 1-bit "counter" for each row in PHT

		GHR				
i	j	NN	NT	TN	TT	Outcome
0	0	N	N	N	N	T
0	1	T	N	N	N	T
0	2	T	T	N	N	T
0	3	T	T	N	T	N
1	0					T
1	1					T
1	2					T
1	3					N
2	0					T
2	1					T
2	2					T
2	3					N

Is global history good in practice?

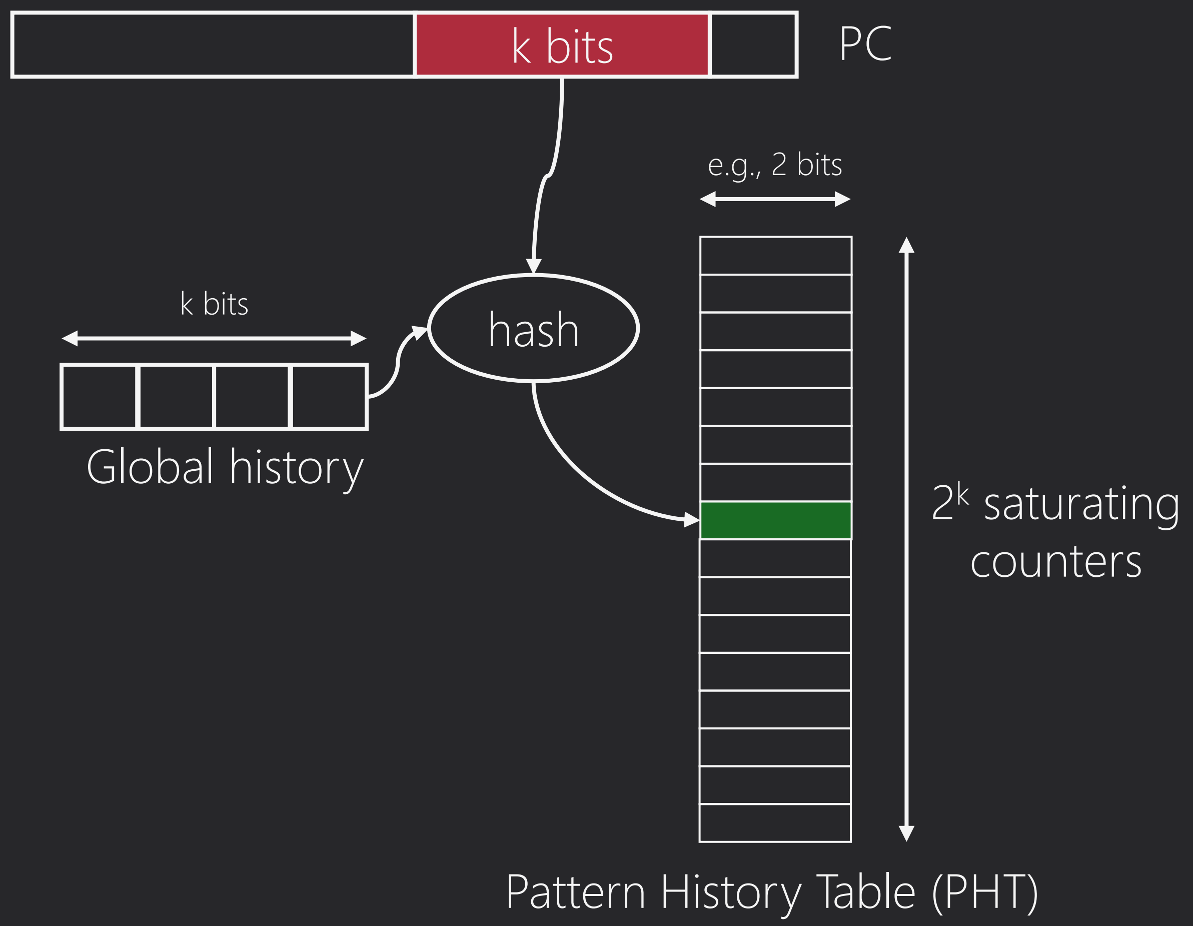
- Global predictors are worse (than local) when…
  - Number of counters is small, or (equivalently)
  - Global history is short

- Can we combine the benefits of local and global?
  1. Combine PC and history bits (e.g., concatenation, has function)
  2. Create separate predictors for each, then select between them (hybrid)

What is the g-share predictor?

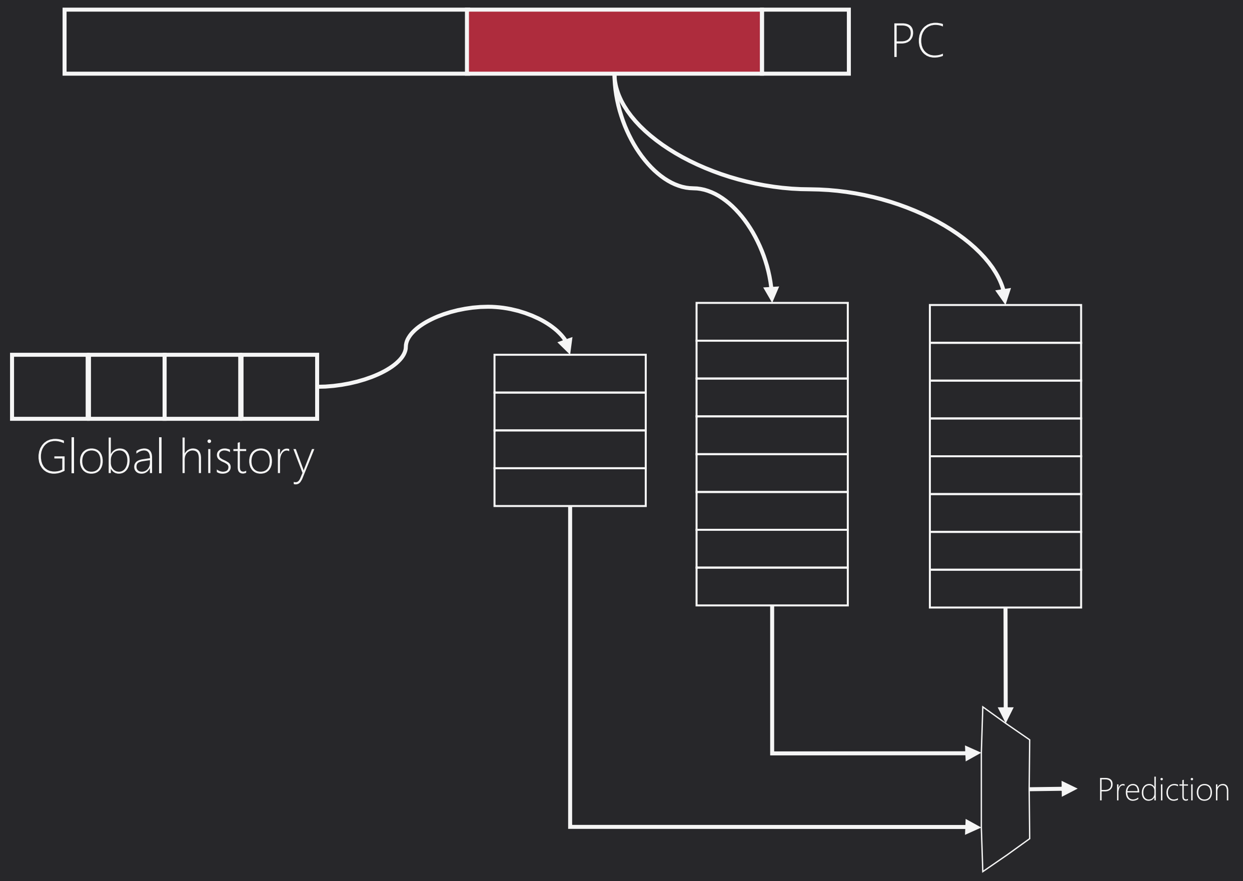
- The *gshare* predictor XORs bits from the PC and global register
  - The resulting bits index into the PHT

- Examples
  - AMD K6 Altheon
  - Sun UltraSPARC III
  - IBM Power4



What is a tournament predictor?

- Branch address (from PC) updates
  - The *selector* counters
  - The *local* PHT
- The branch history updates the *global* PHT
  - Can be smaller
- The *selector* chooses a PHT for the prediction
  - Start in local; use global if correlation is found
- In practice, very accurate! (90%+)

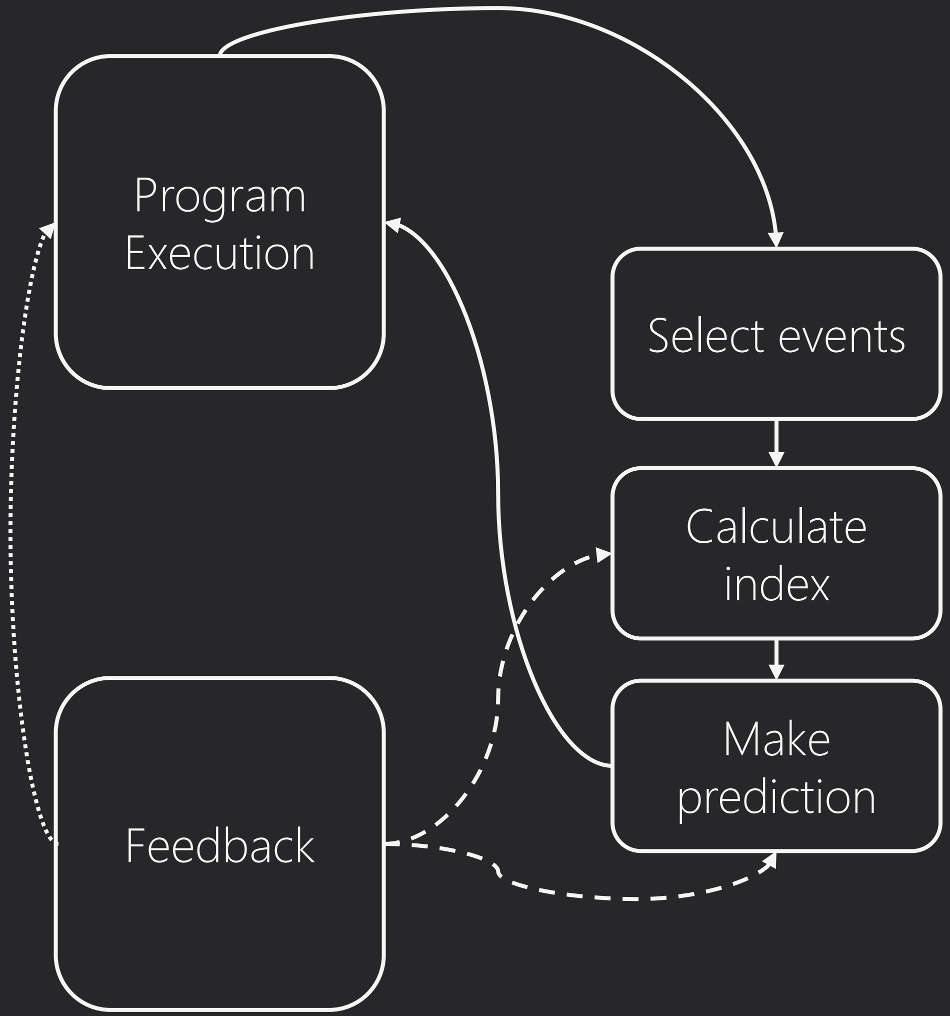


A summary of branch predictors

- We can track branch history in hardware with some overhead
  - Counters, PHTs, shift registers
- There are many forms of hybrid “two-level” branch predictors
  - Tournament is only one example
- Modern branch predictors are quite advanced
  - Machine learning based predictors: “perceptron predictor”
  - Tagged hybrid predictors (many global histories with different lengths)

The flows of branch prediction

- The front end uses the branch predictor
  - Events: all instructions? Branches only?
- Feedback is needed when the branch outcome is calculated
  - Need to update branch predictor
  - Need to update processor; recover from mispredictions



Speculative execution

How to execute speculatively with dynamic scheduling

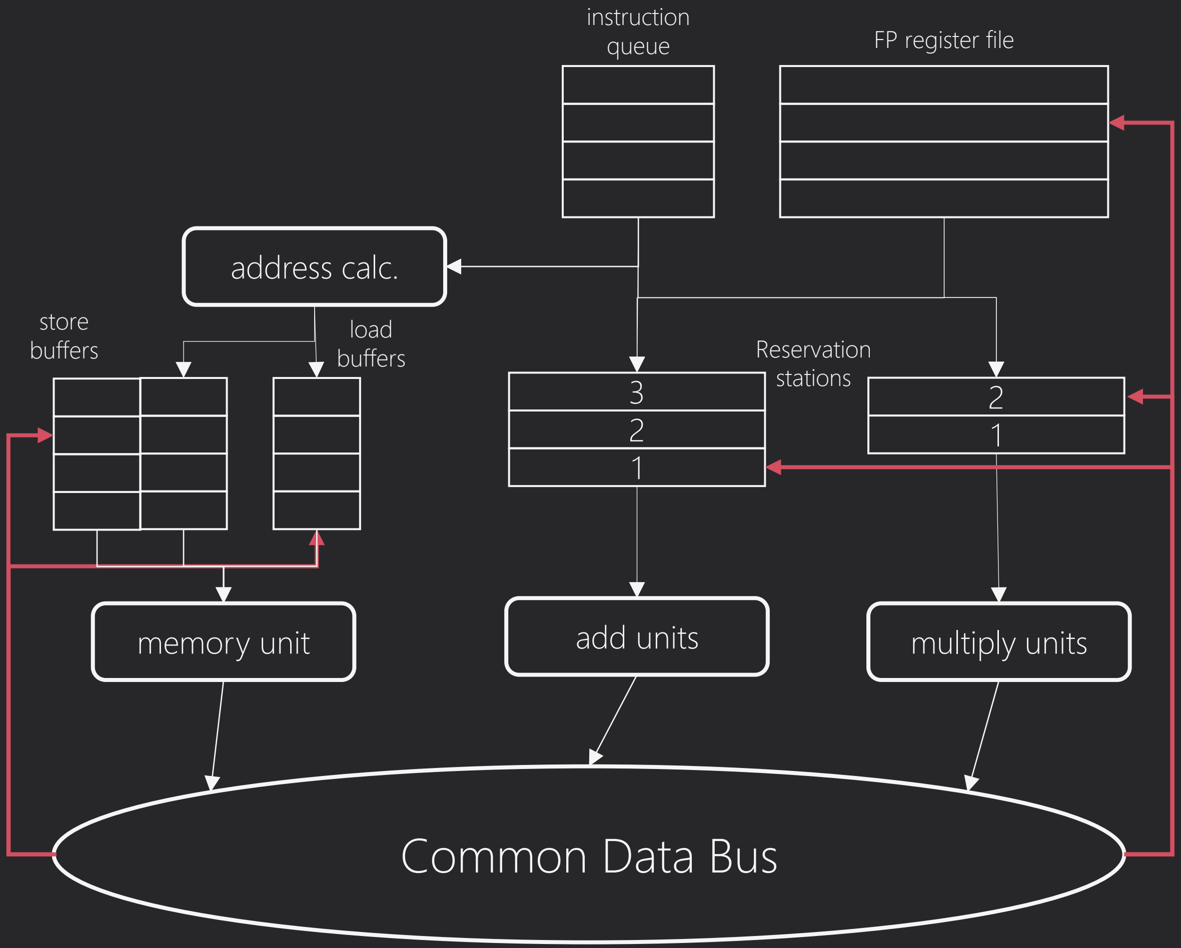
The loop example

```
loop:
    FL.D f0, 0(r1)
    FMUL.D f4, f0, f2
    FS.D f4, 0(r1)
    ADDI r1, r1, -8
    BNE r1, r2, loop
```

- The code multiplies an array of floats by some scalar value (f2)
- Let’s consider two dynamic scheduling pipelines
  - Without Reorder Buffer (ROB)
  - With ROB
- Assume our branch predictor predicts the loop is taken

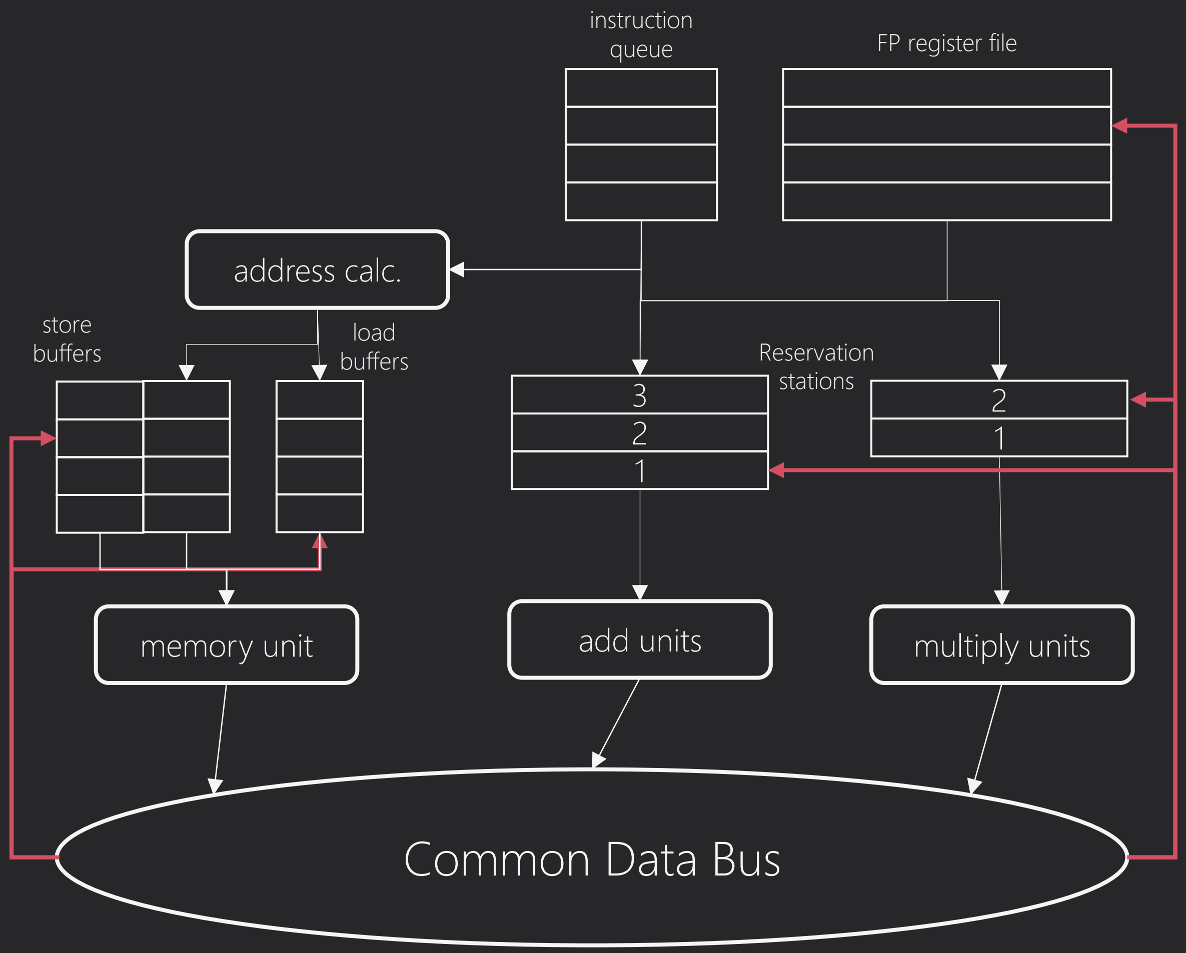
Architecture 1 organization (IBM System/360)

- FIFO instruction queue
- Reservation stations
  - 2 for load, 2 for store
  - 3 for add, 2 for multiply
- Buffers for load and store instructions
- A common data bus (CDB) for broadcasting values
  - From loads, adds, multiplies



Architecture 1 scheduling

1. Issue
   - Dispatch instruction to corresponding reservation station
   - Stall on structural hazard
2. Execute
   - If all operands are available (data), start execution in functional unit
   - Otherwise, wait for data to be broadcast on CDB
3. Write result
   - When functional unit is done, broadcast result on CDB



The reservation station and register file

- Reservation station (RS)
  - The operation to perform (Op)
  - Names of reservation stations that produce a source operand (Q_j, Q_k)
    - 0 means operand value is available
  - Values of the source operands (V_j, V_k)
  - The effective memory address (A)
- Register file
  - The name of the reservation station writing to this register (Q_i)
    - 0 means no write is pending

Op	Q_j	Q_k	V_j	V_k	A	Busy

	Value	Q_i
R0		
R1		
R2		
…		
R31		

The loop example (iteration 0)

```
loop:
      FL.D f0, 0(r1)
      FMUL.D f4, f0, f2
      FS.D f4, 0(r1)
      ADDI r1, r1, -8
      BNE r1, r2, loop
```

Name	Op	Q_j	Q_k	V_j	V_k	A	Busy
Load1							No
Load2							No
Store1							No
Store 2							No
Add1							No
Add2							No
Add3							No
Mul1							No
Mul2							No

The loop example (iteration 1)

```
loop:
         FL.D f0, 0(r1)      # execute
         FMUL.D f4, f0, f2   # issue
         FS.D f4, 0(r1)      # issue
         ADDI r1, r1, -8     # done
         BNE r1, r2, loop
```

Name	*Op*	Q_j	Q_k	V_j	V_k	*A*	*Busy*
Load1	LD					0 + [r1]	Yes
Load2							No
Store1	ST	0	Mul1	[r1]		0	Yes
Store 2							No
Add1							No
Add2							No
Add3							No
Mul1	MUL	Load1	0		[f2]		Yes
Mul2							No

	f0	f2	f4	…	f30
Q_i	Load1		MUL1		

Notes
- Register f0 renamed to Load1
- Register f4 renamed to Mul1

The loop example (iteration 2)

```
FL.D f0, 0(r1)          # execute
FMUL.D f4, f0, f2       # issue
FS.D f4, 0(r1)          # issue
ADDI r1, r1, -8         # done
BNE r1, r2, loop        # predict T
FL.D f0, 0(r1)          # execute
FMUL.D f4, f0, f2       # issue
FS.D f4, 0(r1)          # issue
ADDI r1, r1, -8         # done
BNE r1, r2, loop
```

Name	*Op*	Q_j	Q_k	V_j	V_k	*A*	*Busy*
Load1	LD					[r1] + 0	Yes
Load2	LD					[r1] - 8	Yes
Store1	ST	0	Mul1	[r1]		0	Yes
Store 2	ST	0	Mul2	[r1] - 8		0	Yes
Add1							No
Add2							No
Add3							No
Mul1	MUL	Load1	0		[f2]		Yes
Mul2	MUL	Load2	0		[f2]		Yes

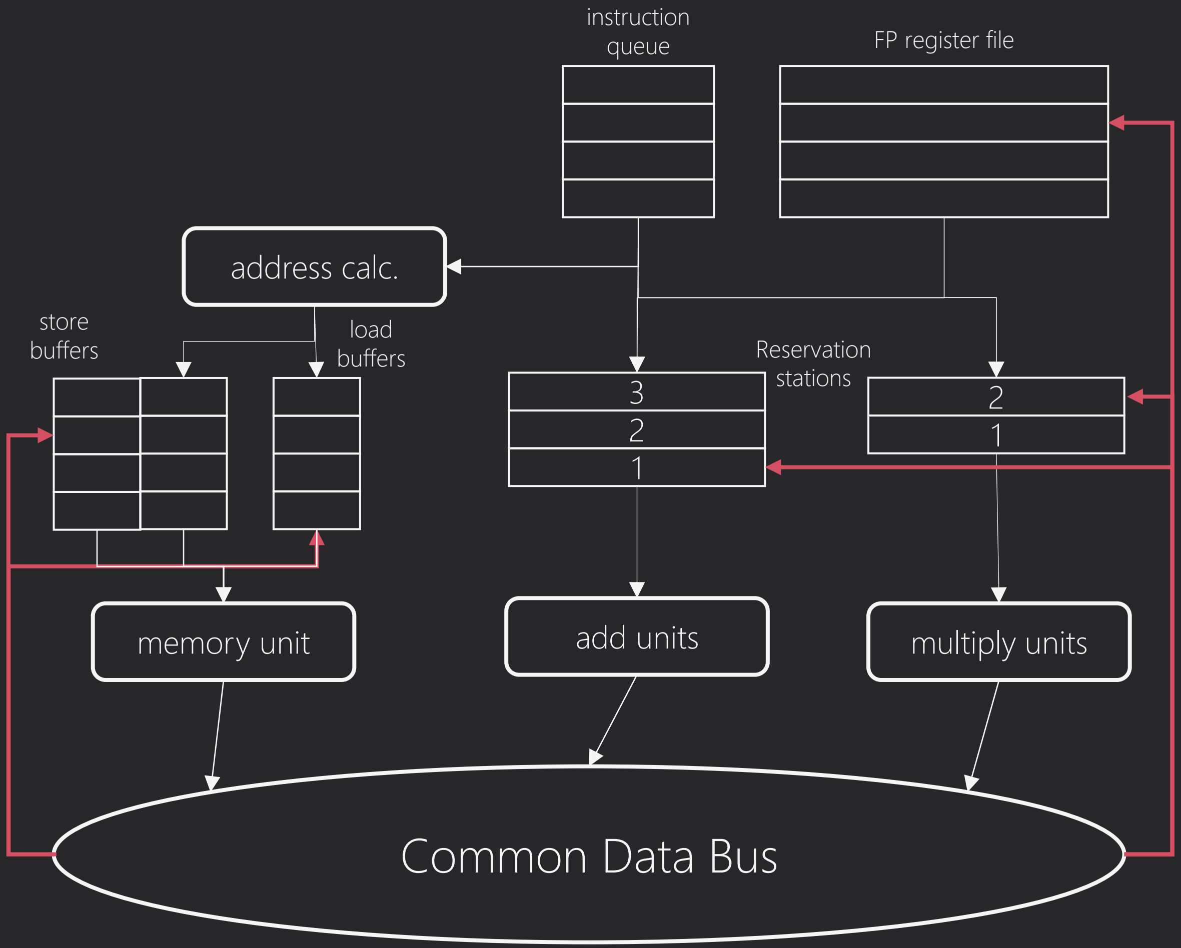
	f0	f2	f4	…	f30
Q_i	Load2		MUL2		

Notes
- Register renaming *dynamically* unrolled the loop!
  - Iteration 1's f0 comes from Load1
  - Iteration 1's f4 comes from Mul1
  - Iteration 2's f0 comes from Load2
  - Iteration 2's f4 comes from Mul2

Speculation in execute for Architecture 1

- Instructions cannot execute until a preceding branch has completed
  - i.e., need to verify branch prediction before instruction enters functional unit

- Instructions wait in reservation stations
  - But there are only a finite number of reservation stations



Architecture 1 summary

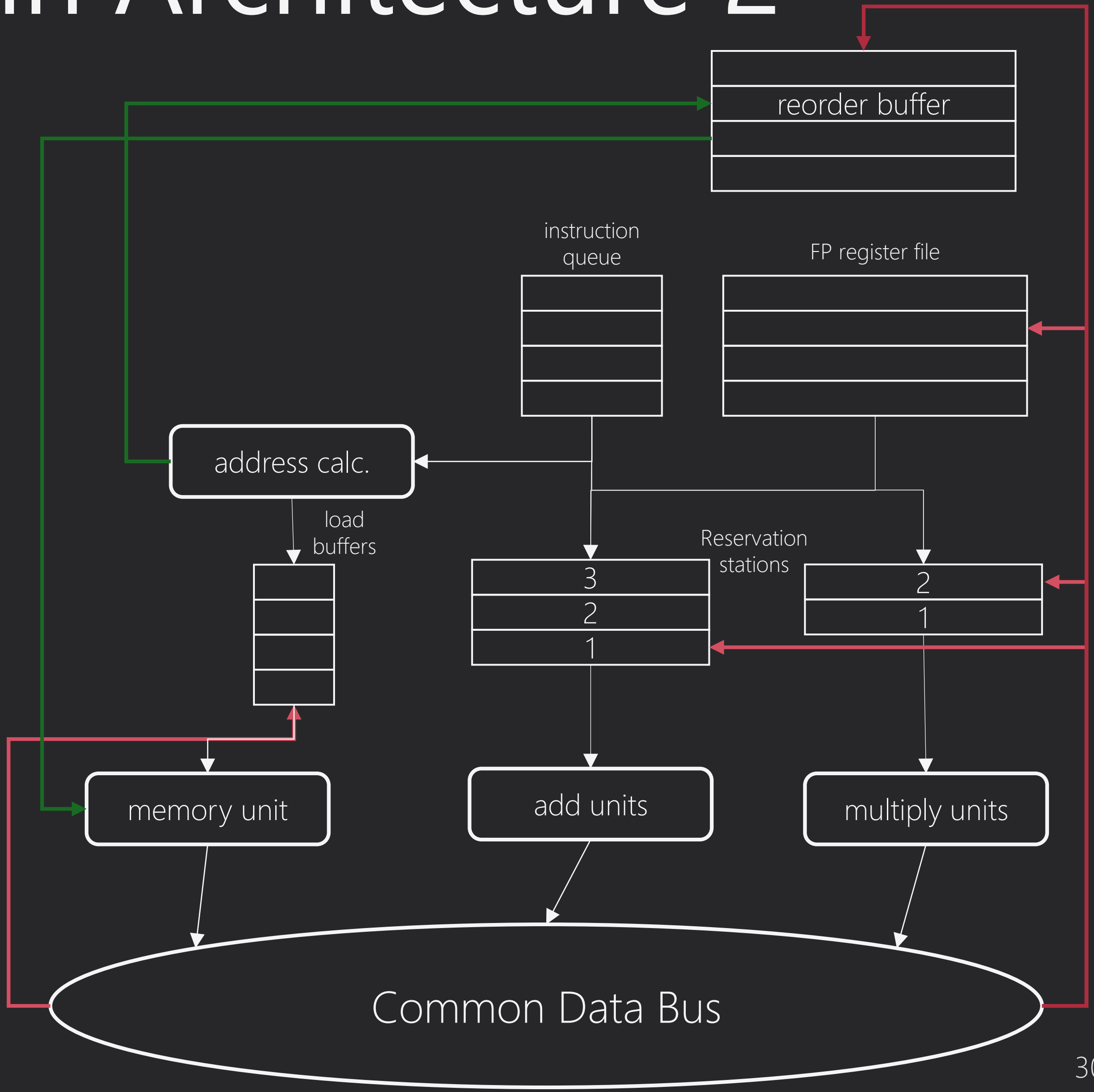
- Dynamic scheduling can exploit instruction-level parallelism
  - Register renaming
  - Reservation stations

- Values can be distributed to many physical locations at once

- But execution is limited by control hazards
  - Functional units cannot begin execution until branch outcomes are determined
  - We don’t have infinite reservation stations

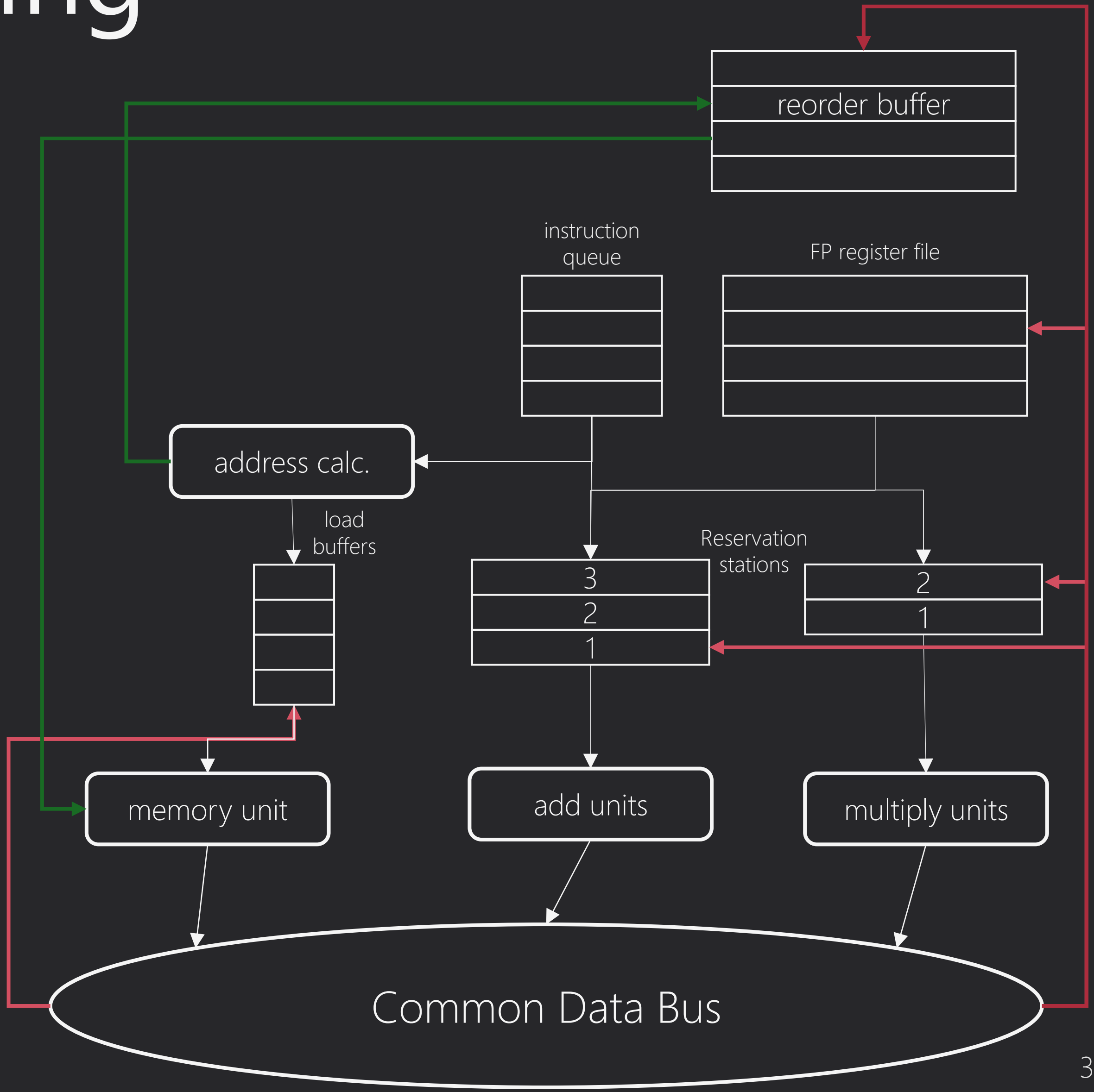
Speculative execution in Architecture 2

- The reorder buffer (ROB) decouples result bypassing from instruction completion
- Using bypassed values can be considered speculative
  - We don't know if the result written on the CDB came from a speculated instruction
- Store buffer is now integrated into ROB



Architecture 2 scheduling

1. Issue
   - Now needs empty reservation station *and* ROB slot
   - Save pointer to ROB slot in RS (new column)
2. Execute
   - As before
3. Write result
   - Broadcast on CDB *with* ROB slot
4. Commit
   - If result is available, instruction is at head, and speculated correctly, update register/memory
   - If speculation was incorrect, flush ROB



The loop example with an ROB

Entry	Busy	Instruction	State	Destination	Value
1	No	FL.D f0, 0(r1)	Commit	`f0`	`Mem[0 + Reg[r1]]`
2	No	FMUL.D f4, f0, f2	Commit	`f4`	`ROB[1] * Reg[f2]`
3	Yes	FS.D f4, 0(r1)	Write result	`Reg[r1] + 0`	`ROB[2]`
4	Yes	ADDI r1, r1, -8	Write result	`r1`	`Reg[r1] – 8`
5	Yes	BNE r1, r2, loop	Write result		
6	Yes	FL.D f0, 0(r1)	Write result	`f0`	`Mem[0 + ROB[4]]`
7	Yes	FMUL.D f4, f0, f2	Write result	`f4`	`ROB[6] * Reg[f2]`
8	Yes	FS.D f4, 0(r1)	Write result	`0 + ROB[4]`	`ROB[7]`
9	Yes	ADDI r1, r1, -8	Write result	`r1`	`ROB[4] – 8`
10	Yes	BNE r1, r2, loop	Write result		

	f0	f2	f4	...	f30
ROB #	6				

Architecture 2 summary

- Like reservation stations, slots in the ROB acts as a new name and physical location for values

- We can support speculative execution with an ROB because
    - Writing result to physical location is decoupled from…
    - Instruction commit (writing to the final destination)

- The ROB can be flushed on mispredictions
    - Hopefully, branch outcomes are resolved quickly to minimize wasted work

Conclusion

Recapping the important points

Is the cost of speculation worth it?

The emphatic yes

- In-order cores are very limited by dynamic events
    - e.g., cache misses
    - e.g., branch mispredictions
- Aggressive out-of-order cores can extract a lot of ILP with
    - Good branch predictors
    - Intricate support for speculative execution

The emphatic no

- A lot of hardware support is required
    - Area overhead
    - Design overhead
    - Out-of-order cores are power hungry
- Speculative execution CPU vulnerabilities
    - Spectre
    - Meltdown

What is value prediction?

- So far we have focussed on patterns with:
  - Branches
  - Memory addresses

- What about the actual data?
  - Load value prediction: hardware that predicts the actual values being loaded from main memory
  - Load value approximation: hardware that supports “approximate computing” (i.e., when the algorithm can tolerate inexactness of the value)

How much ILP is there?

- Processors became more aggressive to exploit ILP
  - Wider pipelines
  - Dynamic scheduling
  - Support for speculative execution

- But research and experience shows ILP is limited
  - Coupled with the power wall, we next transition to other forms of parallelism!