Scoreboarding

Managing Dependences for Multiple Functional Units

CDC 6600 - 1964

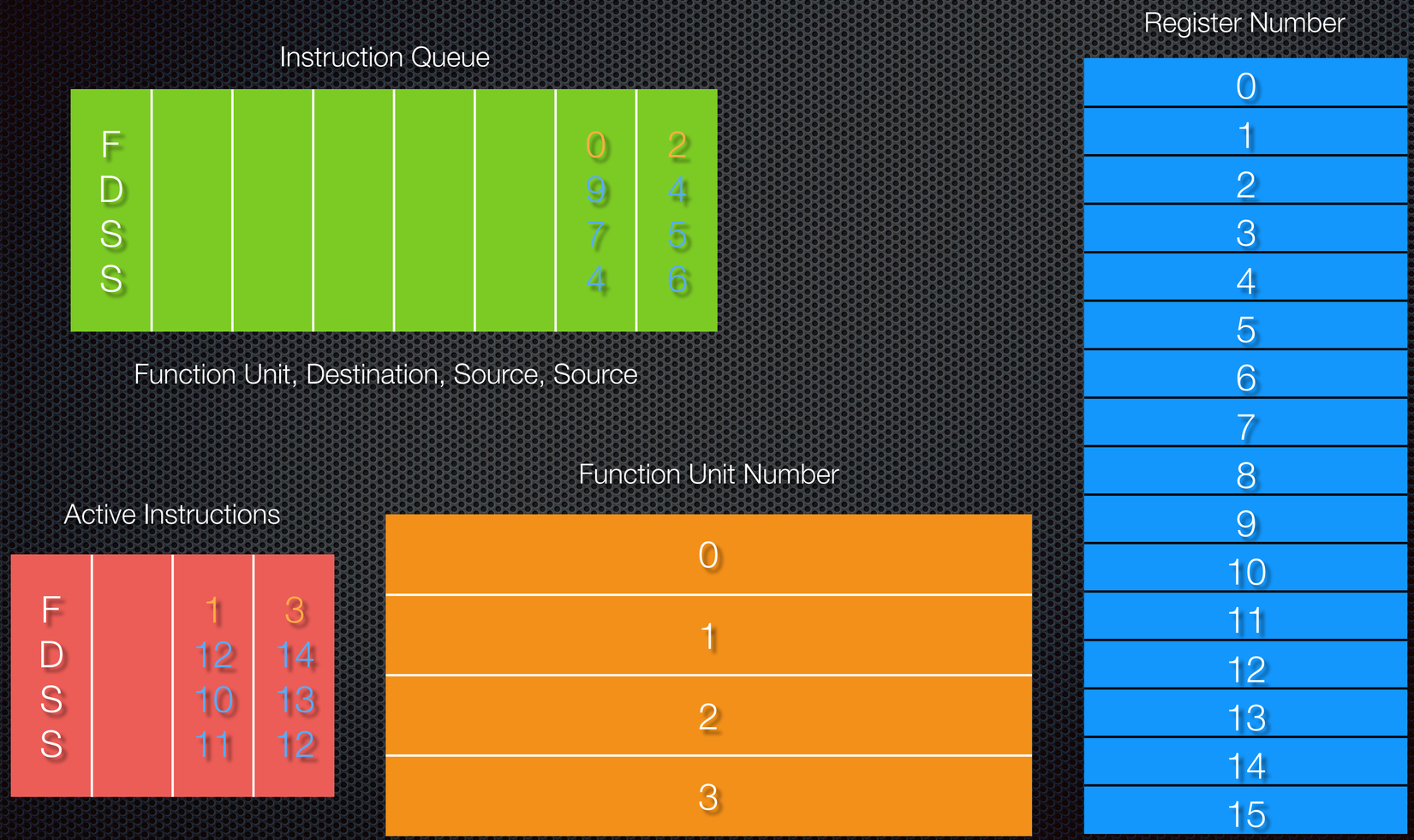
Multiple Functional Units

- Provide parallelism
- Multiple instructions execute simultaneously
- Potential for conflicts on operands

Operand Dependences

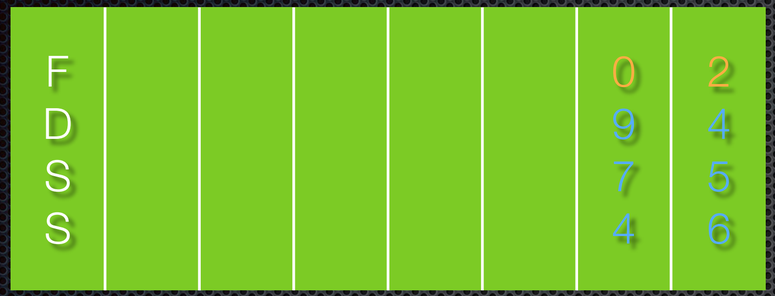
- RAW: Read After Write
  - Need to read (use) a value that has not yet been written
- WAW: Write After Write
  - Writes occur out of order because units vary in timing
- WAR: Write After Read
  - Need to read a value before it gets overwritten

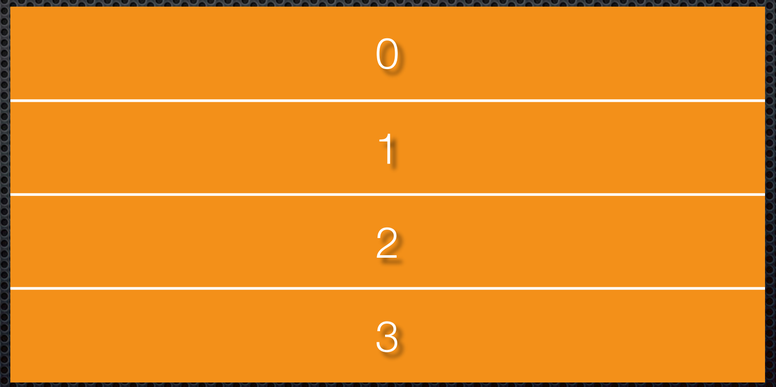
Instructions, Functional Units, and Registers



Result Register Designator

Specifies register is reserved by a given unit



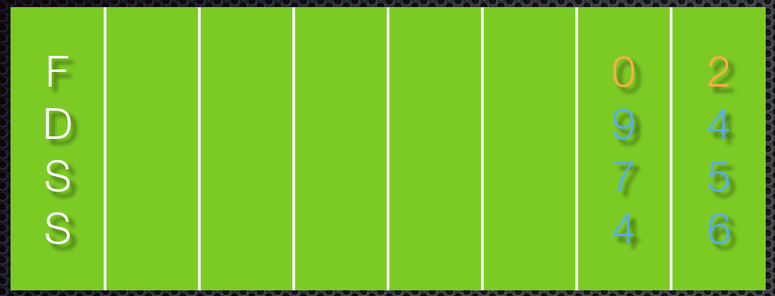


RRD

0	
1	
2	
3	
4	
5	
6	
7	
8	
9	
10	
11	
12	1
13	
14	3
15	

Entry-Operand Register Designator

Specifies operand sources and status for a unit: Valid/Pending, if Pending source unit is given



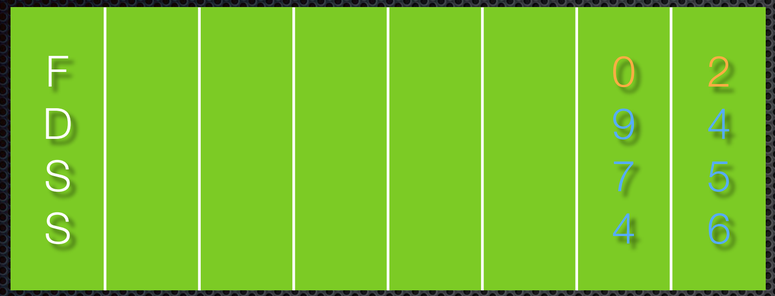
EORD

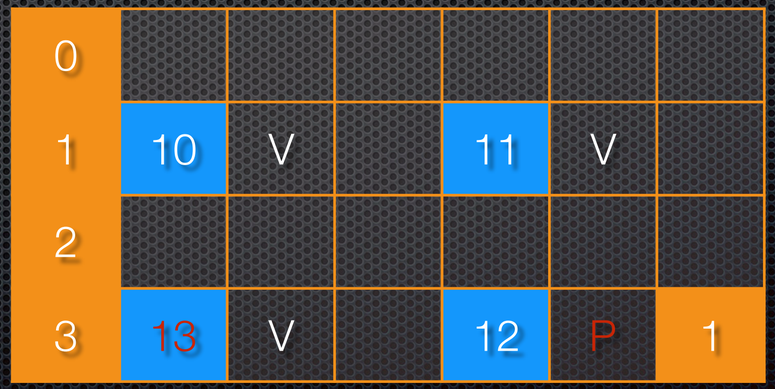
0						
1	10	V		11	V	
2						
3	13	V		12	P	1

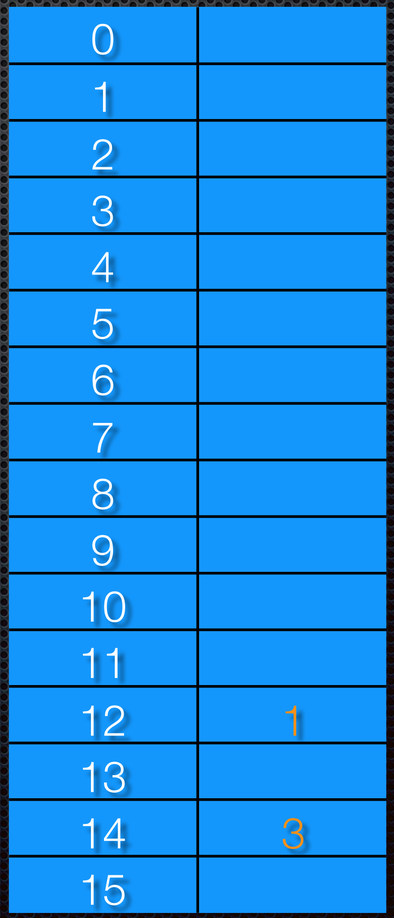
0	
1	
2	
3	
4	
5	
6	
7	
8	
9	
10	
11	
12	1
13	
14	3
15	

Issue Rule

An instruction can issue from the queue when its functional unit is available, and its result register is not reserved by an active instruction. This prevents WAW hazards: it's impossible for two active instructions to be pending with the same destination register.

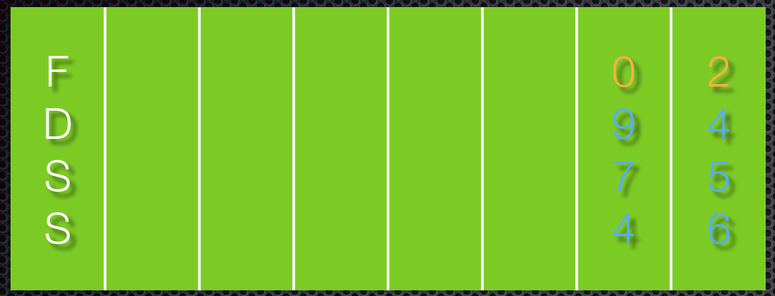


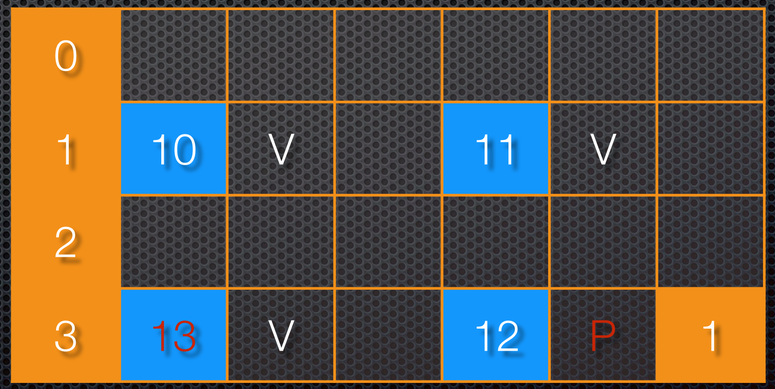


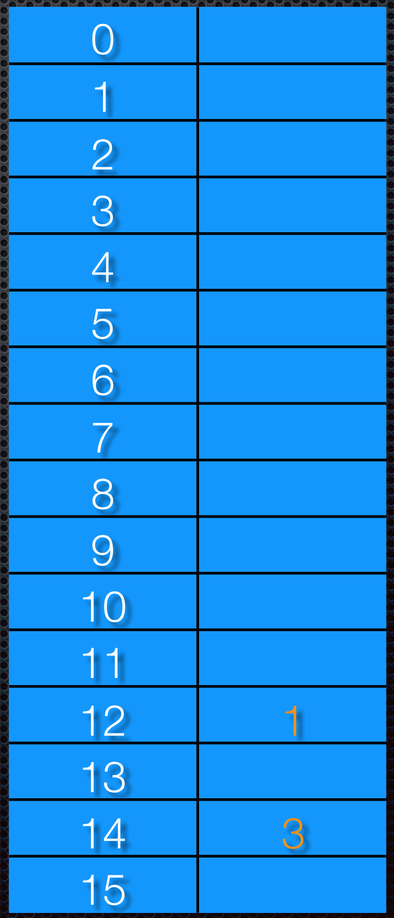


Release Rule

Operands are only released to a functional unit when all of them are valid, preventing RAW hazards.
Release can occur in parallel with register writing, via forwarding.

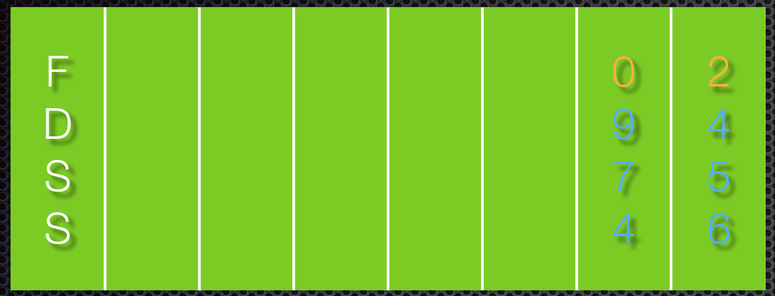


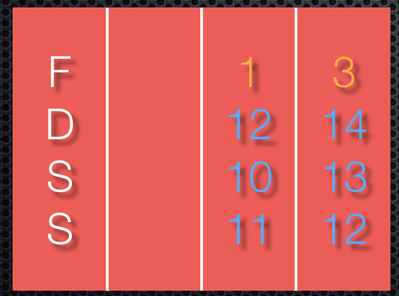


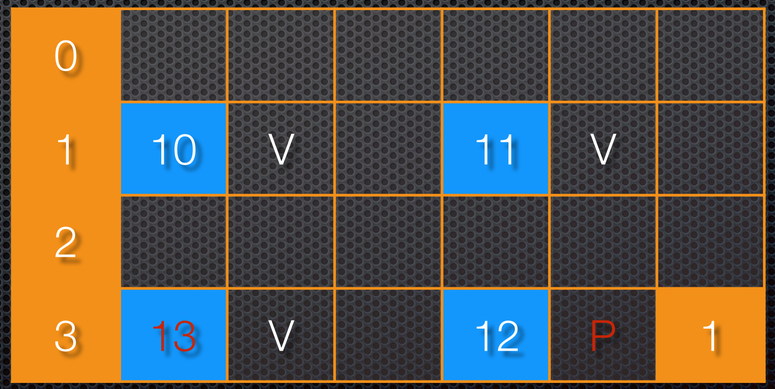


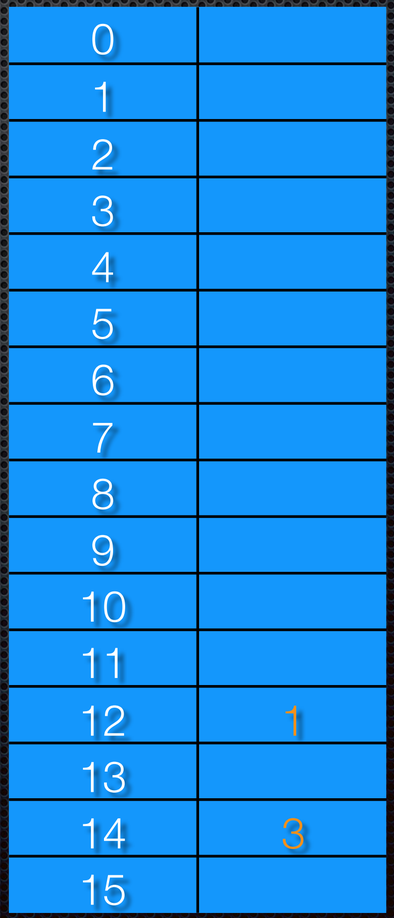
Stall Rule

An instruction will be stalled if its destination is listed as an entry operand for any other pending instruction. This prevents WAR hazards: it's not allowed to write until the value in its destination has been released to all dependent instructions.



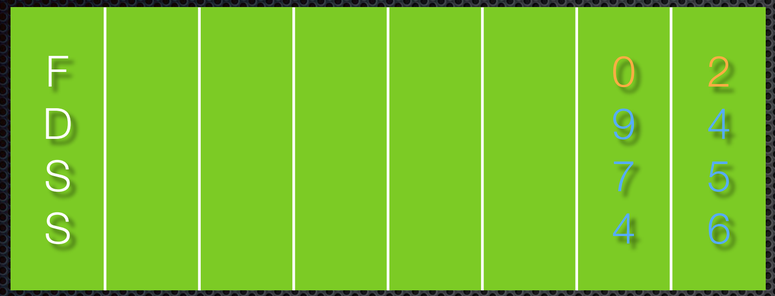


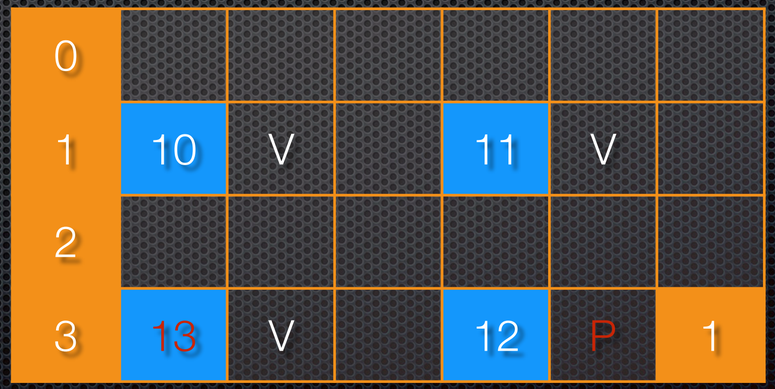


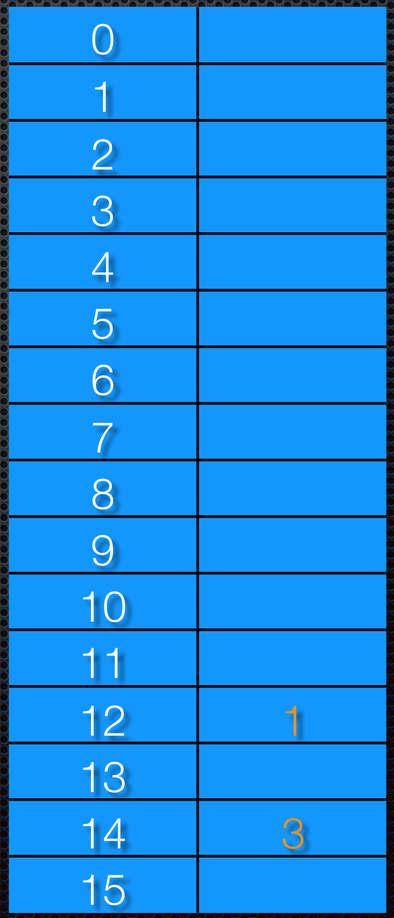


Current status

Instruction in F1 has valid operands. Instruction in F3 is stalled waiting for result from F1 (RAW hazard), and the operand in R13 has not been released to it. The next instruction is free to issue.

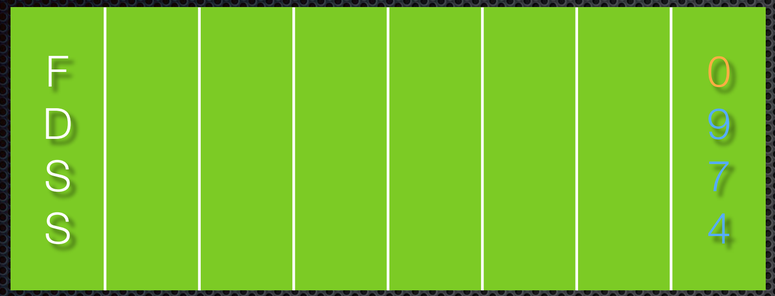






Cycle 2

Result from F1 is valid, and forwarded to F3 while being written in R12. R13 is also released to F3. F1 is busy until result is written. Instruction issues to F2, reserving R4 for result. Valid operands in R5 and R6 are released to F2.



0	
1	
2	
3	
4	2
5	
6	
7	
8	
9	
10	
11	
12	
13	
14	3
15	

F	2	1	3
D	4	12	14
S	5	10	13
S	6	11	12

0						
1	10	V		11	V	
2	5	V		6	V	
3	13	V		12	V	

Cycle 3

New instructions arrive. F1 instruction completes. Result from F3 is valid, and being written in R14. Instruction in F2 is executing. Instruction in queue issues to F0 because F0 and R9 are available, but must wait for result from F2 (RAW hazard).

F					2	3	1
D					13	7	9
S					10	13	13
S					8	12	12

F	0	2	3
D	9	4	14
S	7	5	13
S	4	6	12

0	7	V		4	P	2
1						
2	5	V		6	V	
3	13	V		12	V	

0	
1	
2	
3	
4	2
5	
6	
7	
8	
9	0
10	
11	
12	
13	
14	
15	

Cycle 4

F3 instruction completes. Result from F2 is valid, and being written in R4 and forwarded to F0.
Operands are released to instruction in F0.
Instruction in queue cannot issue to F1 because F0 has the same destination reserved (WAW hazard).

F					2	3	1
D					13	7	9
S					10	13	13
S					8	12	12

F		0	2
D		9	4
S		7	5
S		4	6

0	7	V		4	P	2
1						
2	5	V		6	V	
3						

0	
1	
2	
3	
4	
5	
6	
7	
8	
9	0
10	
11	
12	
13	
14	
15	

Cycle 5

F2 instruction completes. Result from F0 is valid, and being written in R9. Instruction in queue can issue to F1 and receive its operands because F0 is writing in 9 and will finish before F1 starts.

F						2	3
D						13	7
S						10	13
S						8	12

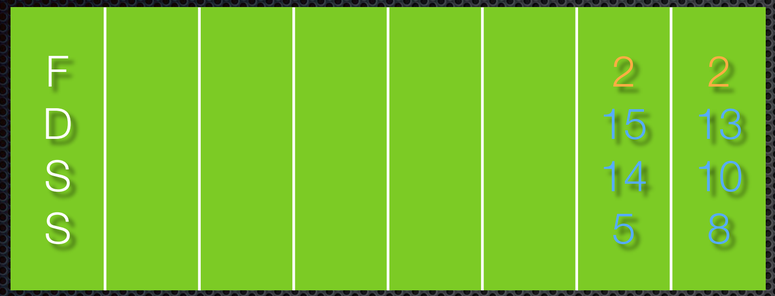
F		1	0
D		9	9
S		13	7
S		12	4

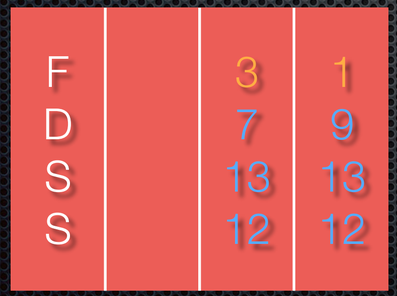
0	7	V		4	V	
1	13	V		12	V	
2						
3						

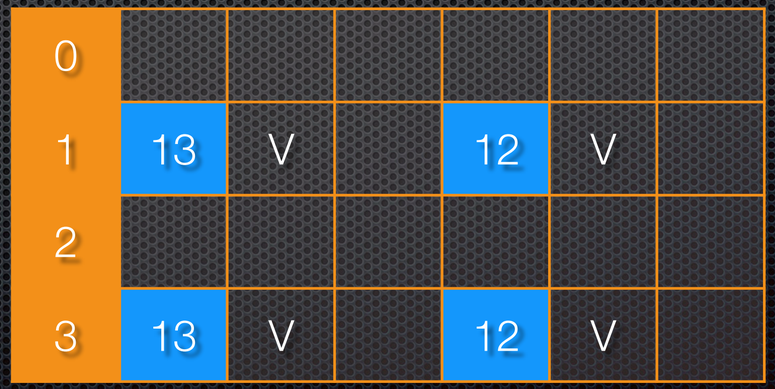
0	
1	
2	
3	
4	
5	
6	
7	
8	
9	1
10	
11	
12	
13	
14	
15	

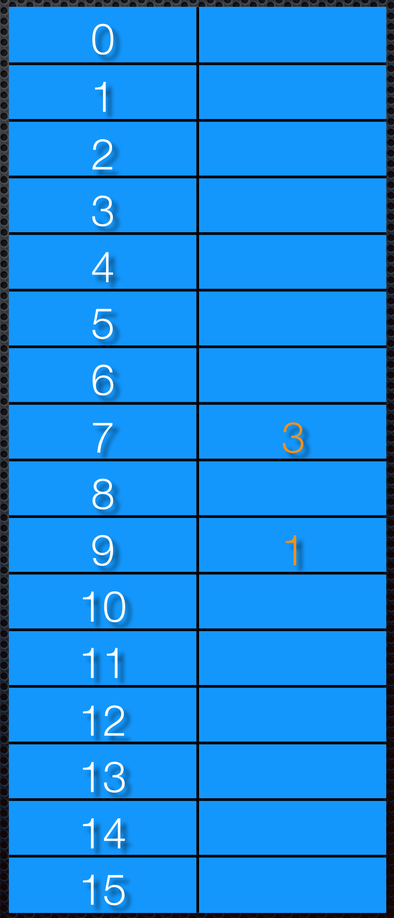
Cycle 6

F0 instruction completes. Instruction in queue can issue to F3 and receive its operands because F3 and R7 are available. It's fine to read the same operands as F0 (RAR isn't a hazard).



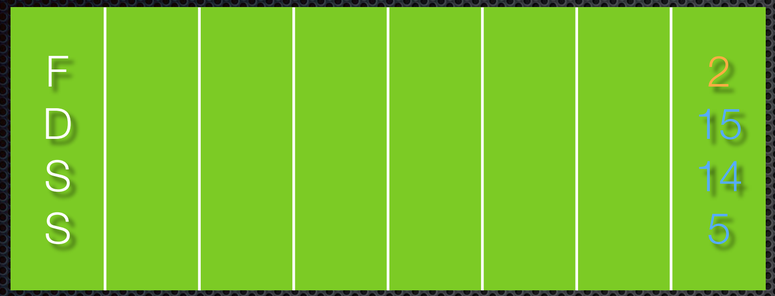






Cycle 7

F1 instruction is writing its result to R9.
Instruction in queue issues and receives operands because F2 and R13 are available and no writes are pending to R10 or R8.
F2 stalls because its destination is an entry operand for F3. (WAR Hazard)



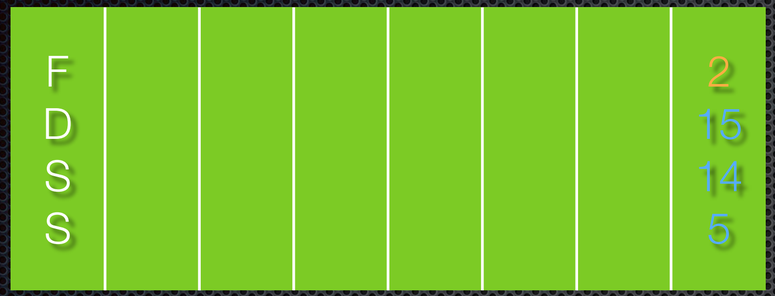
F	2	3	1
D	13	7	9
S	10	13	13
S	8	12	12

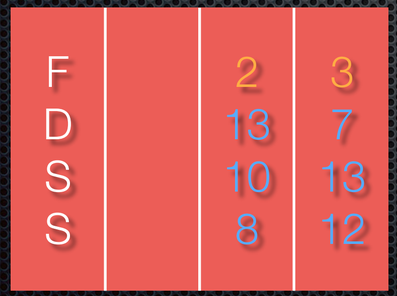
0						
1	13	V		12	V	
2	10	V		8	V	
3	13	V		12	V	

0	
1	
2	
3	
4	
5	
6	
7	3
8	
9	
10	
11	
12	
13	2
14	
15	

Cycle 8

F1 instruction completes. F3 is writing to R7. F2 starts execution.
Instruction in queue cannot issue because F2 is busy (structural hazard).



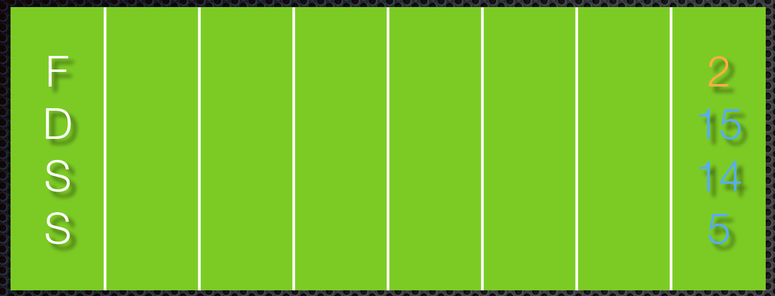


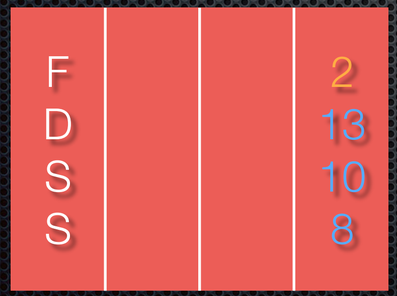
0						
1						
2	10	V		8	V	
3	13	V		12	V	

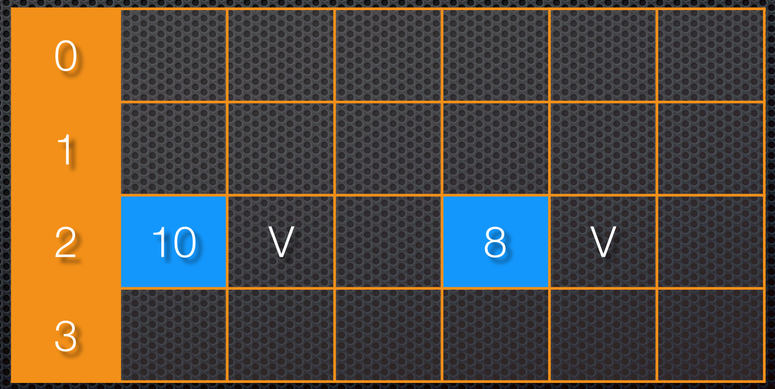
0	
1	
2	
3	
4	
5	
6	
7	
8	
9	
10	
11	
12	
13	2
14	
15	

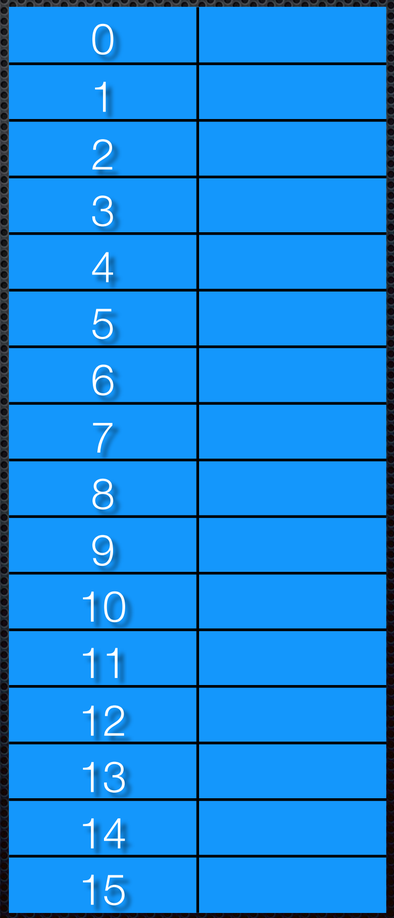
Cycle 9

F3 instruction completes. F2 is writing to R13.
Instruction in queue cannot issue because F2 is busy (structural hazard).









Summary

- Scoreboard avoids dependence hazards
- Centralized control of issue/release/stall
- Overly restrictive
- Simple to implement

Reservation Stations

Tomasulo's Algorithm for Managing Dependence Among Multiple Functional Units

IBM 360/91 - 1966-1967

Reservation Station

- One or more associated with each functional unit
- Contains:
  - Operation
  - Operand values, if already computed
  - Operand source stations, if not yet available
  - Operand and reservation station status

Registers

- Extended with result designator, indicating reservation station that will write to it
- Keeps track of most recently issued instruction that targets this register
- Unlike scoreboard, this can change before the register is updated
- Issuing instructions check the registers to see where each operand is coming from

Station 5 has operation X with valid values, executing on unit 2

Instruction Queue

OP						Z	X
D						9	4
S						7	5
S						4	6

Reservation Stations

1	OP	V1	V2	S1	S2
2						
3						
4						
5	X	3	17	0	0
6						
7						
8						

Function Units

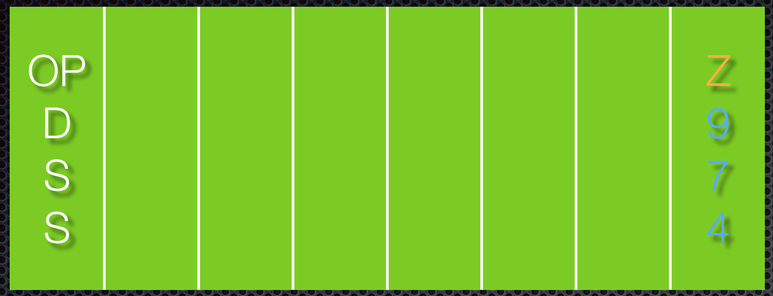
0
1
2
3

Common Data Bus

Reg#	Desig	Value
0	0	14
1	0	25
2	0	84
3	0	63
4	0	22
5	5	
6	0	97
7	0	45
8	0	48
9	0	18
10	0	27
11	0	83
12	0	16
13	0	47
14	0	69
15	0	51

Instruction in queue issues to station 6 with invalid V1 value, and copy of R5 designator for station 5. V2 is a valid copy of R6.

Instruction Queue

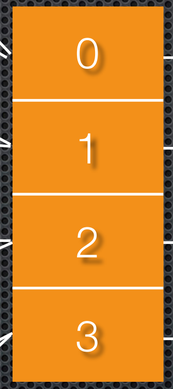


Copying value avoids WAW and WAR hazards. R4 now designates its value is pending from station 6.

Reservation Stations

	OP	V1	V2	S1	S2
1					
2					
3					
4					
5	X	3	17	0	0
6	X	0	97	5	0
7					
8					

Function Units

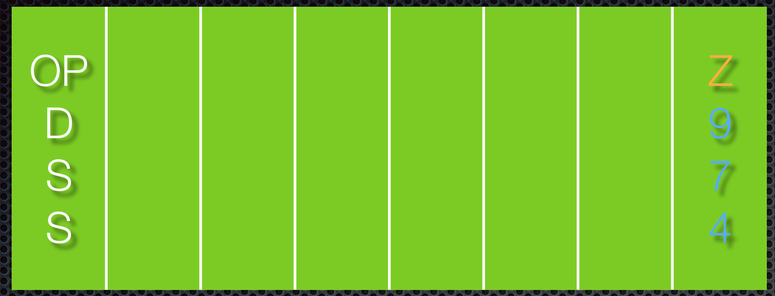


Common Data Bus

Reg#	Desig	Value
0	0	14
1	0	25
2	0	84
3	0	63
4	6	22
5	5	99
6	0	97
7	0	45
8	0	48
9	0	18
10	0	27
11	0	83
12	0	16
13	0	47
14	0	69
15	0	51

Function unit 2 produces a result for station 5 and broadcasts it on the CDB, where the station number will match the S1 designator in station 6

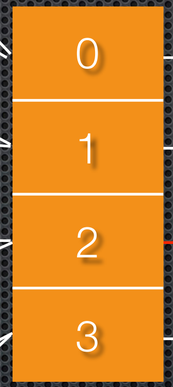
Instruction Queue



Reservation Stations

	OP	V1	V2	S1	S2
1					
2					
3					
4					
5	X	3	17	0	0
6	X	0	97	5	0
7					
8					

Function Units



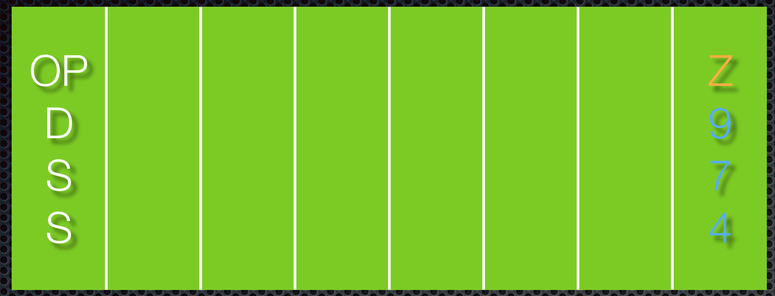
5

Common Data Bus

Reg#	Desig	Value
0	0	14
1	0	25
2	0	84
3	0	63
4	6	22
5	5	99
6	0	97
7	0	45
8	0	48
9	0	18
10	0	27
11	0	83
12	0	16
13	0	47
14	0	69
15	0	51

Station 6 snoops the value, and copies it, clearing the S1 designator. It is now ready to execute. This avoids RAW hazards.

Instruction Queue

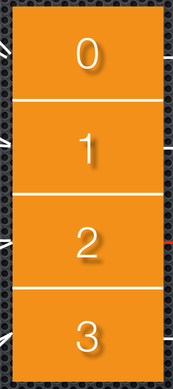


Register 5 now has a valid result, and a 0 designator

Reservation Stations

	OP	V1	V2	S1	S2
1					
2					
3					
4					
5	X	3	17	0	0
6	X	77	97	0	0
7					
8					

Function Units



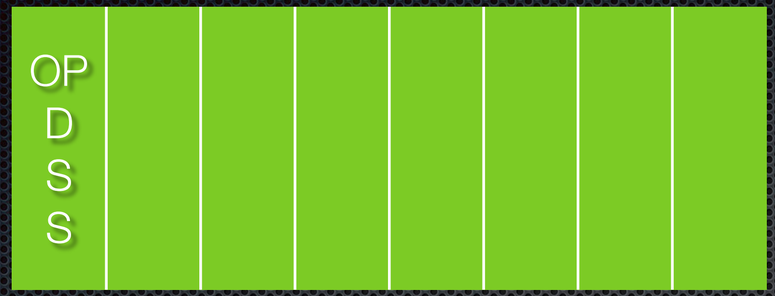
5

Common Data Bus

Reg#	Desig	Value
0	0	14
1	0	25
2	0	84
3	0	63
4	6	22
5	0	77
6	0	97
7	0	45
8	0	48
9	0	18
10	0	27
11	0	83
12	0	16
13	0	47
14	0	69
15	0	51

Instruction in queue issues to station 3, copying value from R7, and designator from R4 that result will come from station 6

Instruction Queue

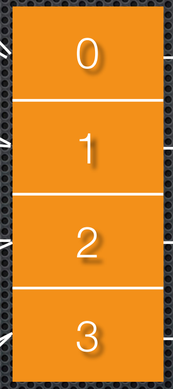


Register 9 designates that 3 will write it.

Reservation Stations

	OP	V1	V2	S1	S2
1					
2					
3	Z	45	0	0	6
4					
5					
6	X	77	97	0	0
7					
8					

Function Units



Common Data Bus

Reg#	Desig	Value
0	0	14
1	0	25
2	0	84
3	0	63
4	6	22
5	0	77
6	0	97
7	0	45
8	0	48
9	3	18
10	0	27
11	0	83
12	0	16
13	0	47
14	0	69
15	0	51

Register Renaming

- Reservation stations act as extra registers
- If a register has a designator other than 0 that differs from the station that is about to write it, the write is cancelled
  - An instruction is pending that will overwrite it
  - All consumers of this value already have copies
- Result designator acts as a pointer to the “station register” that has taken this logical register’s name

Comparison with Scoreboard

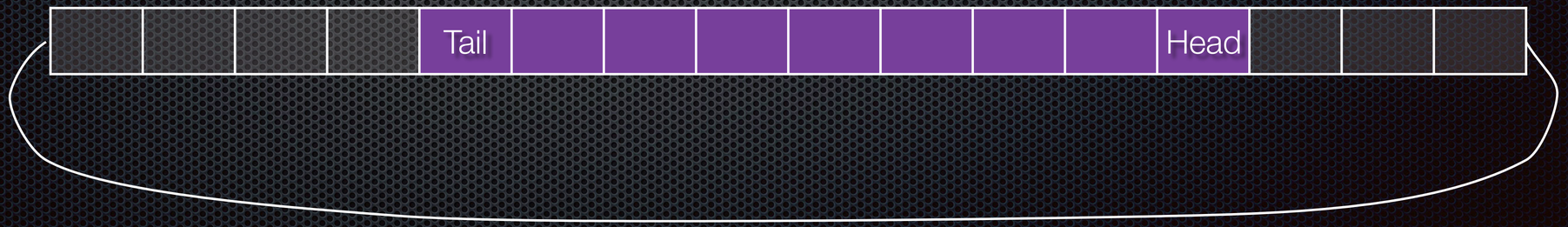
- Distributed (vs. centralized)
- Allows out-of-order execution
- Allows more instructions to proceed
- More complex to implement
- Adds a one cycle delay for forwarding
- Neither can execute past a branch

Sohi IEEE TC 1990

Instruction Issue Logic for High Performance Interruptible, Multiple Functional Unit, Pipelined Computers

Register Update Unit

- Similar to reservation stations, but arranged in a queue
- Head points to next instruction to commit
- Tail is next available issue slot
- If Head = Tail, then RUU is full and no issue allowed



RUU Issue

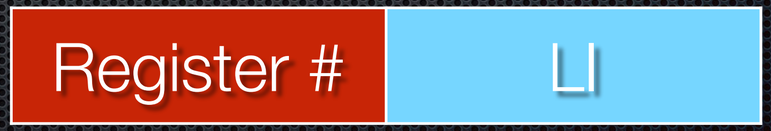
- Instruction goes into current tail pointer station
- Tail pointer is updated
- If source register values are available, copied to station
- If value(s) not available, station gets tag(s)

RUU Register Extensions

- Registers can have multiple instances
- Queue structure keeps instances in order
- Registers extended with Number of Instances and Last Instance fields (NI, LI)
- When an instruction with destination R_i issues, NI_i and LI_i are incremented, LI is modulo n, NI is max 2^n-1
- When NI is max, issue is blocked
- Committing instruction with R_i destination decrements NI_i

RUU Tag

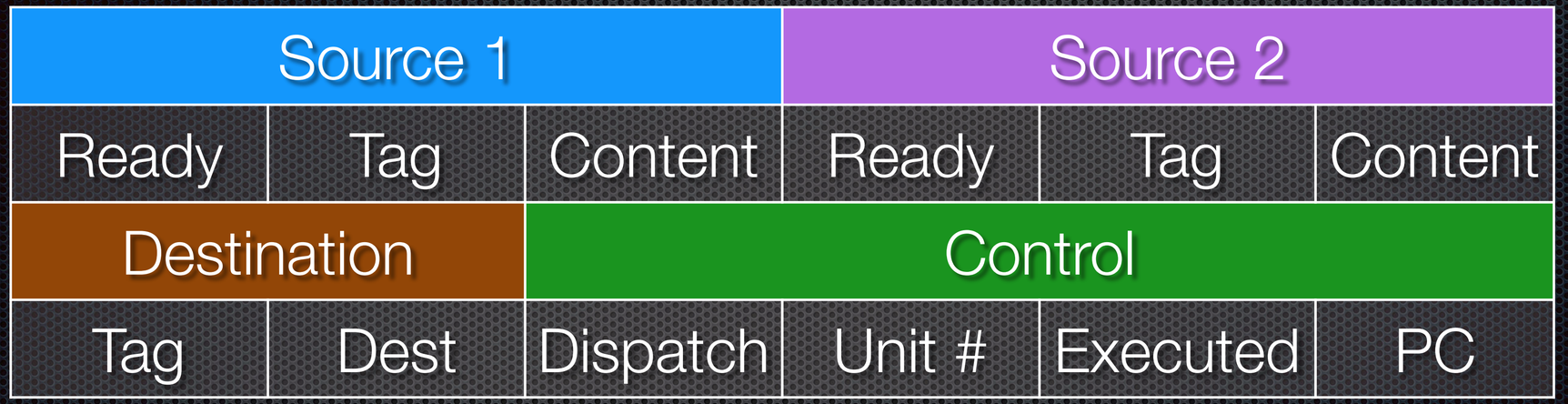
- Tags in the RUU are R_i concatenated with LI_i
- A tag thus indicates the most recent destination instance for the register
- Up to 7 instances of a register are allowed in this implementation



RUU Entry Source Fields

- Ready bit
- Tag subfield
- Content subfield
- If not ready, then watch result bus for matching tag
- When tag is seen, copy result bus into content subfield and switch to Ready

RUU Entry Structure



Determining Issue

- When operands are all ready, an instruction can be issued to a functional unit
- Loads and stores get priority (other schemes possible)
- Others issued in order received

Commit

- When Executed is set, instruction is done
- Results go to register file
- NI for destination register is decremented
- Head pointer is updated

Results

Number of Entries in RUU	Relative Speedup	Instruction Issue Rate
3	0.853	0.380
4	0.940	0.419
6	1.079	0.481
8	1.248	0.557
10	1.383	0.617
12	1.508	0.673
15	1.584	0.706
20	1.649	0.735
25	1.682	0.750
30	1.700	0.758
40	1.735	0.774
50	1.737	0.775

With bypass

Number of Entries in RUU	Relative Speedup	Instruction Issue Rate
3	0.824	0.366
4	0.912	0.407
6	1.031	0.460
8	1.082	0.483
10	1.103	0.492
12	1.216	0.542
15	1.225	0.546
20	1.295	0.578
25	1.330	0.593
30	1.393	0.621
40	1.439	0.642
50	1.471	0.656

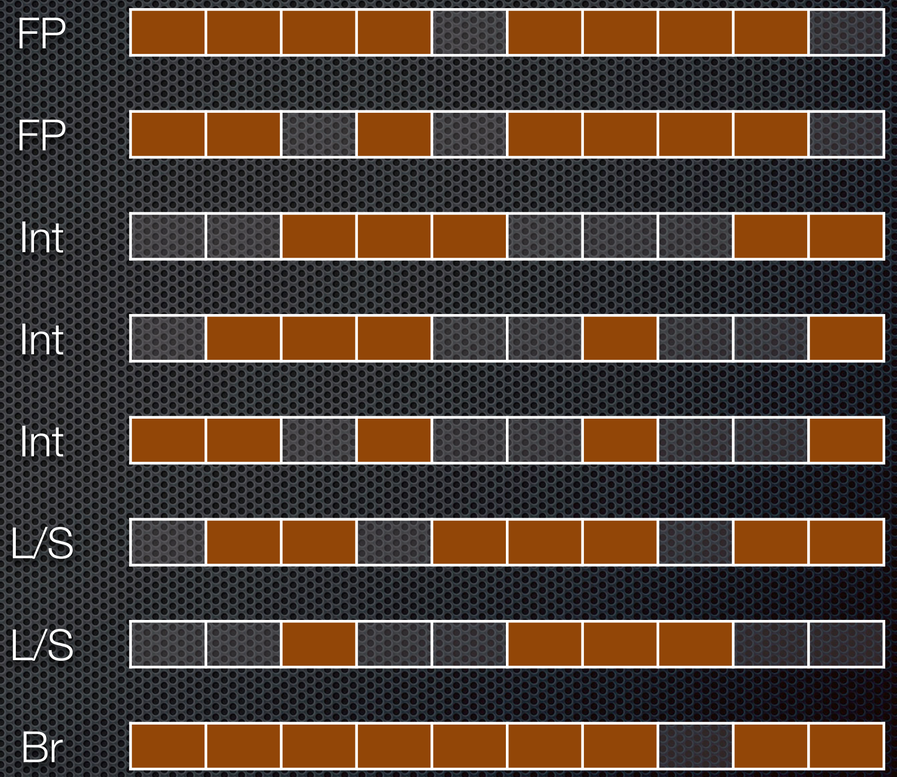
Without bypass

Branch Prediction

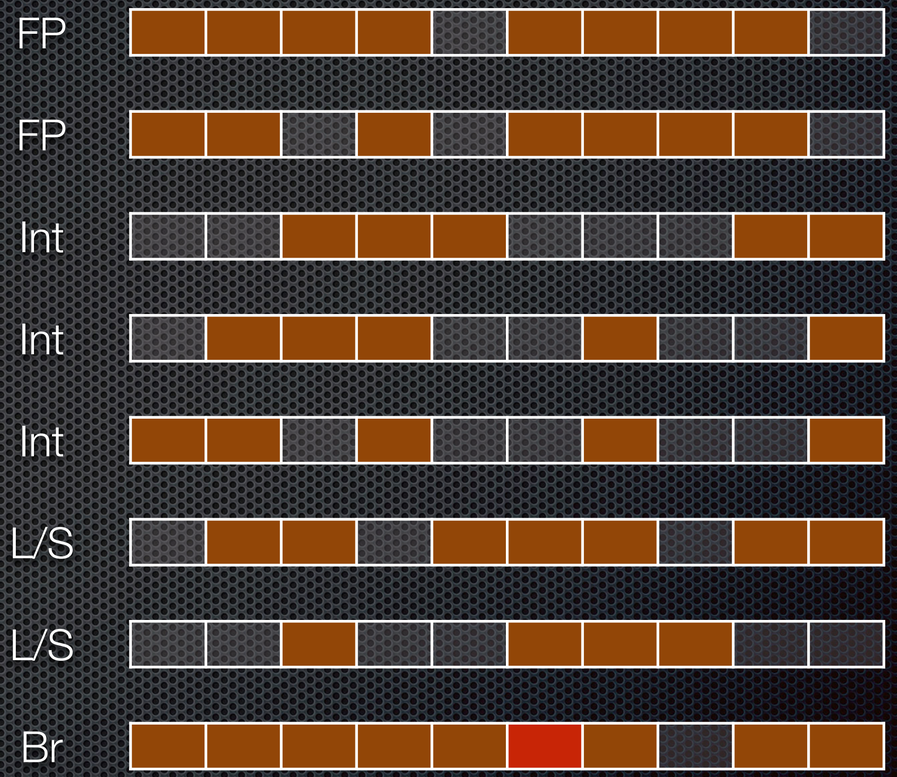
Reducing Pipeline Bubbles

The Problem

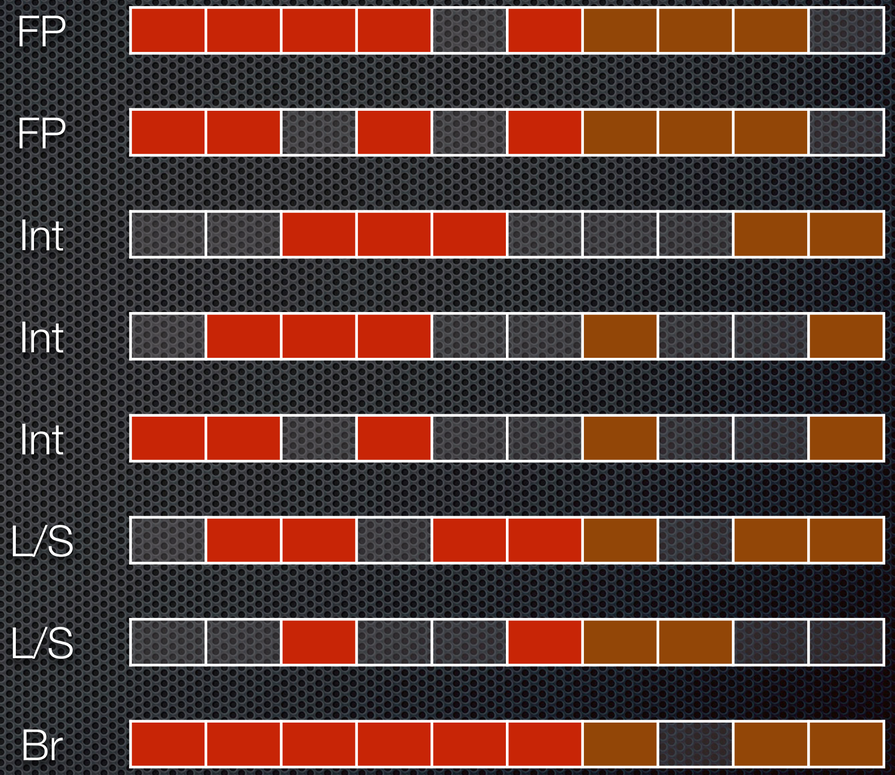
- Branches occur approximately every 5 instructions
- Superscalar pipelines have on the order of 100 instructions in flight
- To get instructions in to the pipelines we have to predict the ways that branches will go
- Branch outcome may not be computed until several stages into the pipeline
- A mispredicted branch means wrong instructions are in the pipes, so they must be flushed



A branch occurs with most issue slots



Misprediction of a branch may be detected late



Many dependent instructions must be flushed

Pipeline Efficiency = $1/(1+P_j\ P_t\ P)$

$P_j = 0.2$ (20% jumps)
$P_t = 0.1$ (10% mispredicts)
$P = 30$ (penalty)
Efficiency = 62.5%

Predictor Context

- Local: Specific to a particular branch, or its history
- Global: Using information from other branches (correlations between branches can be more predictive)
- Hybrid: Selecting different prediction mechanisms for different branches

Static Predictions

- Predictions that do not depend on run time
- Always-taken (or always not taken)
- Backward taken (loop return) and forward not (loop exit) -- uses sign of relative address in branch instruction

Dynamic Prediction

- Uses information from branch history
- For example, predict same as last outcome
  - Need to record state of the branch