Lecture 18: Instruction Level Parallelism -- Dynamic Scheduling, Multiple Issue, and Speculation

CSE 564 Computer Architecture Summer 2017

Department of Computer Science and Engineering
Yonghong Yan
yan@oakland.edu
www.secs.oakland.edu/~yan

Topics for Instruction Level Parallelism

- ILP Introduction, Compiler Techniques and Branch Prediction
  - 3.1, 3.2, 3.3
- Dynamic Scheduling (OOO)
  - 3.4, 3.5 and C.5, C.6 and C.7 (FP pipeline and scoreboard)
- Hardware Speculation and Static Superscalar/VLIW
  - 3.6, 3.7
- Dynamic Scheduling, Multiple Issue and Speculation
  - 3.8, 3.9, 3.13
- ILP Limitations and SMT
  - 3.10, 3.11, 3.12

Acknowledge and Copyright

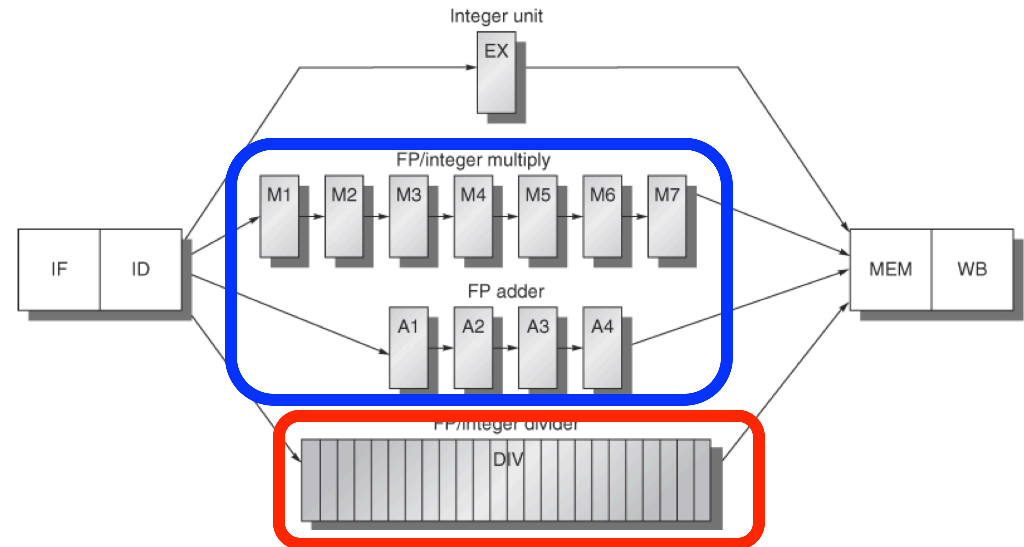
- **Slides adapted from**
  - UC Berkeley course "Computer Science 252: Graduate Computer Architecture" of David E. Culler Copyright(C) 2005 UCB
  - UC Berkeley course Computer Science 252, Graduate Computer Architecture Spring 2012 of John Kubiatowicz Copyright(C) 2012 UCB
  - Computer Science 152: Computer Architecture and Engineering, Spring 2016 by Dr. George Michelogiannakis from UC Berkeley

- **https://passlab.github.io/CSE564/copyrightack.html**

REVIEW

Not Every Stage Takes only one Cycle

- **FP EXE Stage**
  - **Multi-cycle Add/Mul**
  - **Nonpiplined for DIV**



- **MEM Stage**

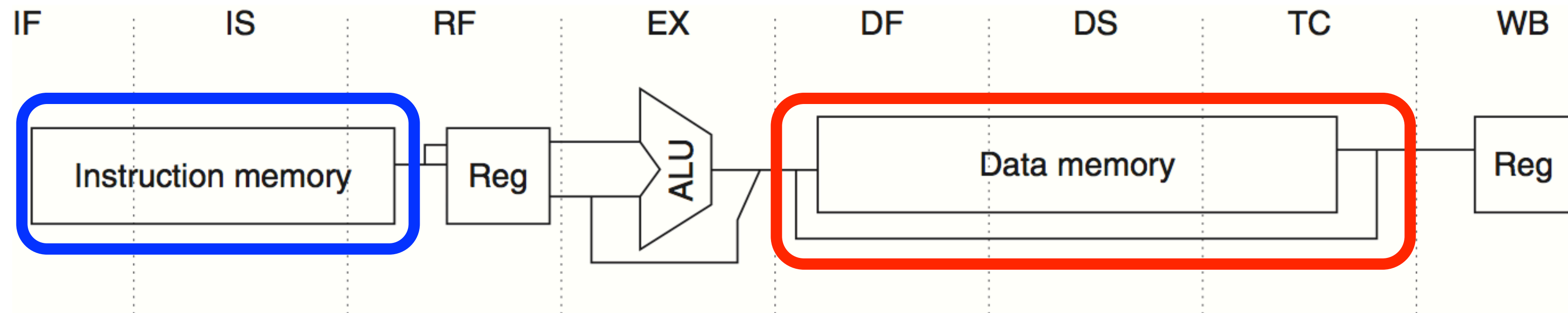


Figure C.41 The eight-stage pipeline structure of the R4000 uses pipelined instruction and data caches. The pipe stages are labeled and their detailed function is described in the text. The vertical dashed lines represent the stage boundaries as well as the location of pipeline latches. The instruction is actually available at the end of IS, but the tag check is done in RF, while the registers are fetched. Thus, we show the instruction memory as operating

Issues of Multi-Cycle in Some Stages

- The divide unit is not fully pipelined
  - structural hazards can occur
    - » need to be detected and stall incurred.
- The instructions have varying running times
  - the number of register writes required in a cycle can be > 1
- Instructions no longer reach WB in order
  - Write after write (WAW) hazards are possible
    - » Note that write after read (WAR) hazards are not possible, since the register reads always occur in ID.
- Instructions can complete in a different order than they were issued (out-of-order complete)
  - causing problems with exceptions
- Longer latency of operations
  - stalls for RAW hazards will be more frequent.

Hardware Solution for Addressing Data Hazards

- **Dynamic Scheduling**
  - **Out-of-order execution and completion**
- **Data Hazard via Register Renaming**
  - **Dynamic RAW hazard detection and scheduling in data-flow fashion**
  - **Register renaming for WRW and WRA hazard (name conflict)**

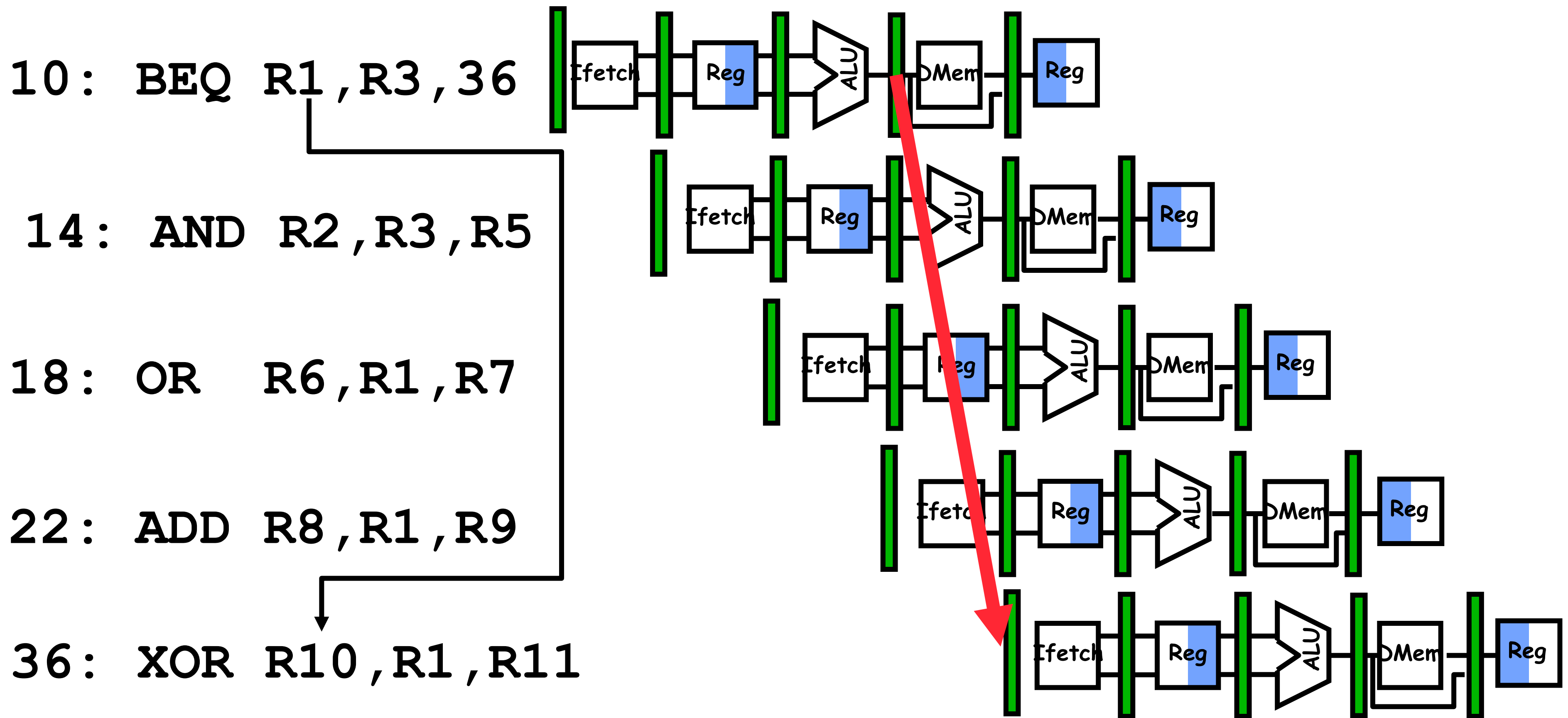
- **Implementations**
  - **Scoreboard (CDC 6600 1963)**
    - » **Centralized register renaming**
  - **Tomasulo's Approach (IBM 360/91, 1966)**
    - » **Distributed control and renaming via reservation station, load/ store buffer and common data bus (data+source)**

Register Renaming Summary

- **Purpose of Renaming: removing "Anti-dependencies"**
  - Get rid of WAR and WAW hazards, since these are not "real" dependencies
- **Implicit Renaming: i.e. Tomasulo**
  - Registers changed into values or response tags
  - We call this "implicit" because space in register file may or may not be used by results!
- **Explicit Renaming: more physical registers than needed by ISA.**
  - Rename table: tracks current association between architectural registers and physical registers
  - Uses a translation table to perform compiler-like transformation on the fly
- **With Explicit Renaming:**
  - All registers concentrated in single register file
  - Can utilize bypass network that looks more like 5-stage pipeline
  - Introduces a register-allocation problem
    - » Need to handle branch misprediction and precise exceptions differently, but ultimately makes things simpler

HARDWARE SPECULATION: ADDRESSING CONTROL HAZARDS

Control Hazard from Branches: Three Stage Stall if Taken



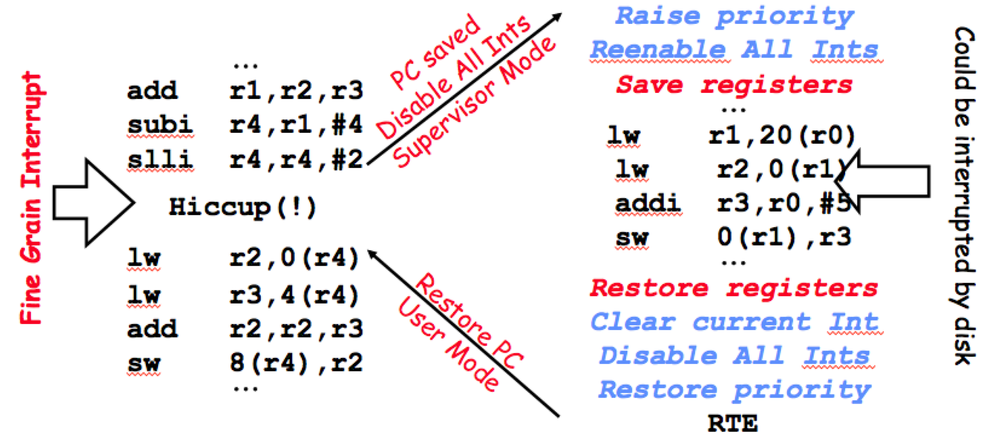
What do you do with the 3 instructions in between?

How do you do it?

Control Hazards

- Break the instruction flow

- Unconditional Jump
- Conditional Jump
- Function call and return
- Exceptions



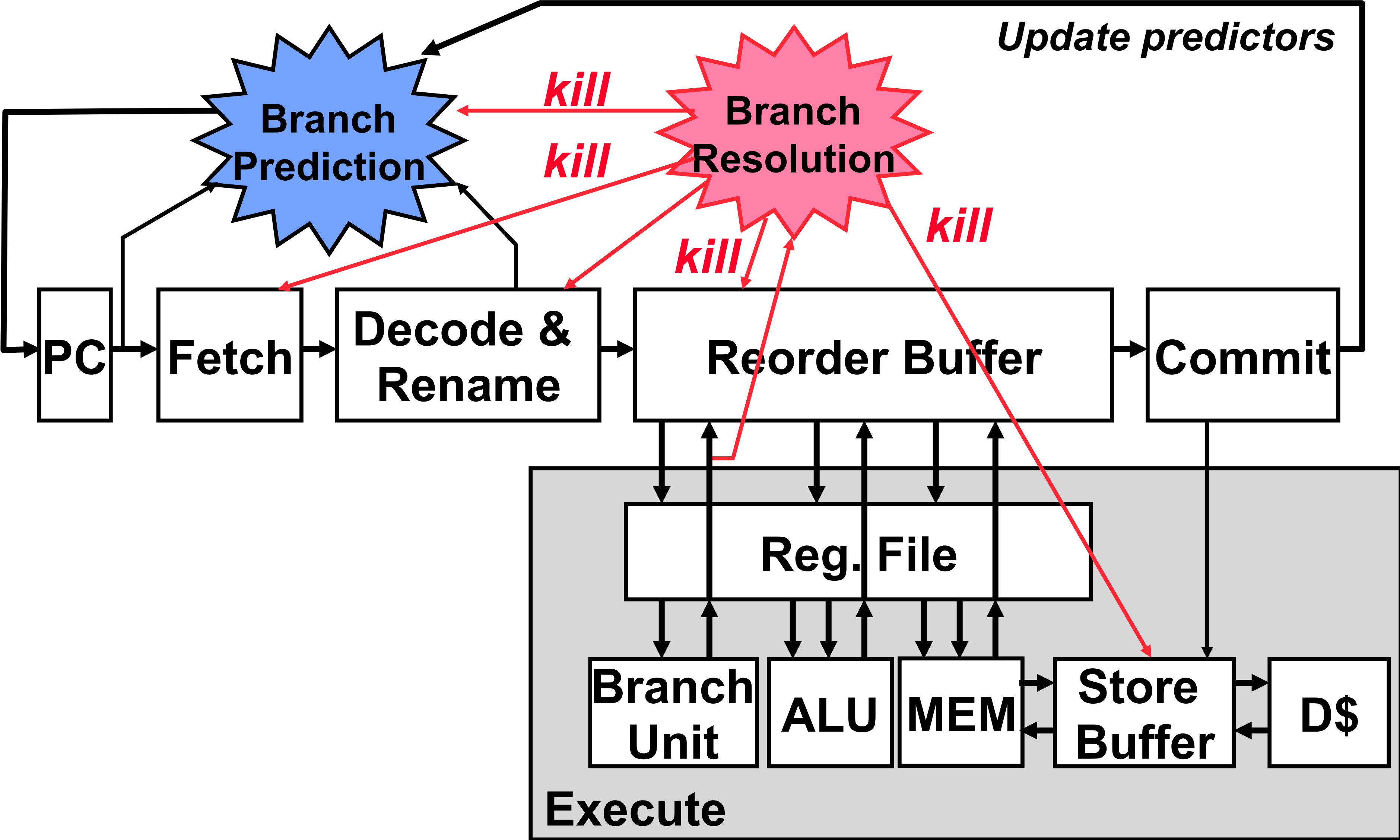
- Multiple instructions following branch in program order can complete before branch resolves

Branch Prediction + Speculation (Misprediction Recovery)

- **Branch Prediction:**
  - Modern branch predictors have high accuracy: (>95%) and can reduce branch penalties significantly
  - Required hardware support
    - » Branch history tables (Taken or Not)
    - » Branch target buffers, etc. (Target address)

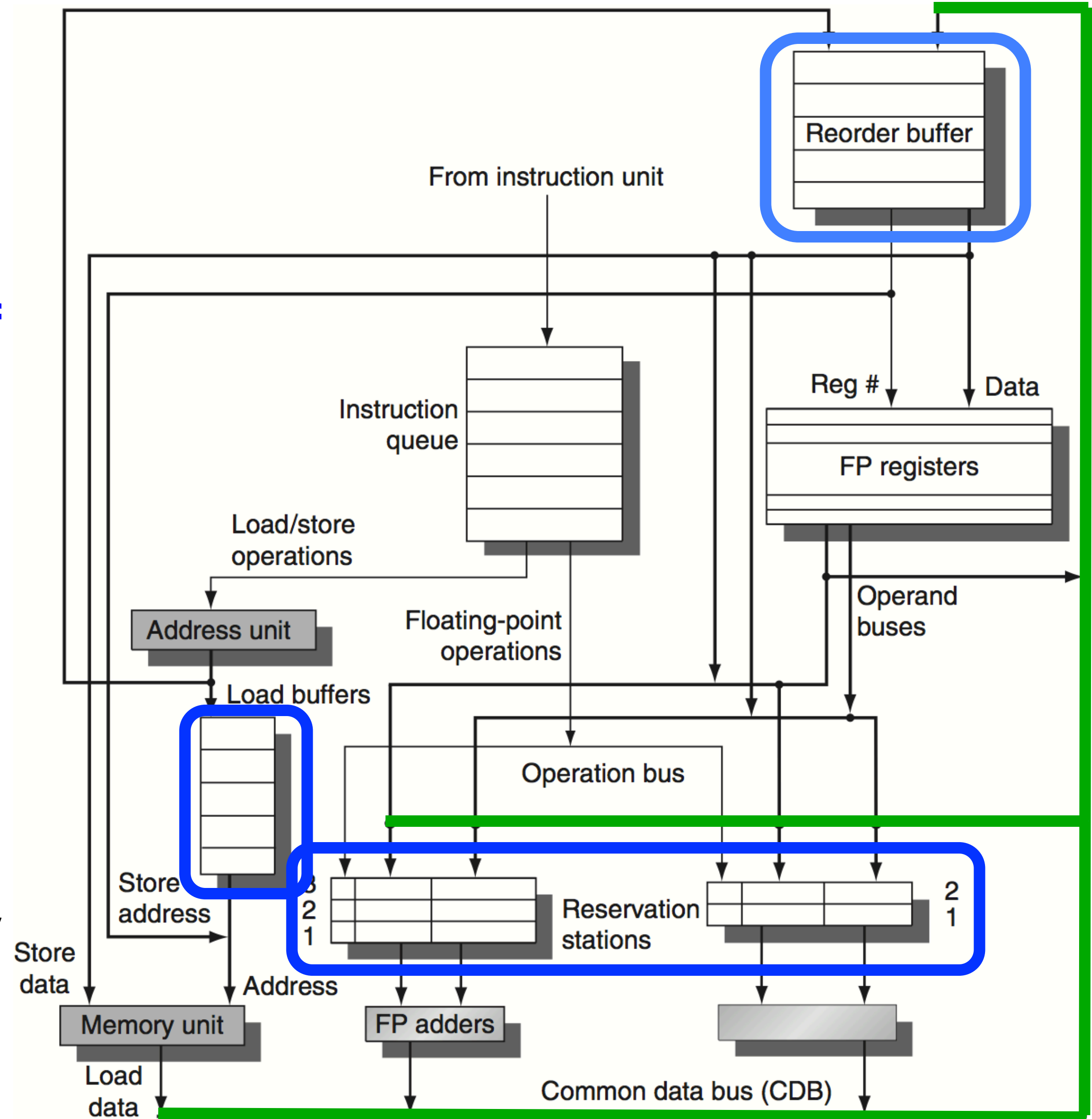
- **In-order commit for out-of-order execution:**
  - Instructions fetched and decoded into instruction reorder buffer in-order
  - Execution is out-of-order ($\Rightarrow$ out-of-order completion)
  - Commit (write-back to architectural state, i.e., regfile & memory) is in-order

Branch Prediction/Speculation



Hardware Speculation in Tomasulo Algorithm

- **Reservation Station and Load Buffer**
  - Register renaming
  - For dynamic scheduling and out-of order execution
- **Reorder Buffer**
  - Register renaming
  - For in-order commit
- **Common Data Bus**
  - Data forwarding
- **Also handle memory data hazard**



Four Steps of *Speculative* Tomasulo

1. **Issue—get instruction from FP Op Queue**

   If reservation station and reorder buffer slot free, issue instr & send operands & reorder buffer no. for destination (this stage sometimes called "dispatch")

2. **Execution—operate on operands (EX)**

   When both operands ready then execute; if not ready, watch CDB for result; when both in reservation station, execute; checks RAW (sometimes called "issue")

3. **Write result—finish execution (WB)**

   Write on Common Data Bus to all awaiting FUs & reorder buffer; mark reservation station available.

4. **Commit—update register with reorder result**

   When instr. at head of reorder buffer & result present, update register with result (or store to memory) and remove instr from reorder buffer. Mispredicted branch flushes reorder buffer (sometimes called "graduation")

Instruction In-order Commit

- Also called completion or graduation
- In-order commit
  - In-order issue
  - Out-of-order execution
  - Out-of-order completion
- Three cases when an instr reaches the head of ROB
  - Normal commit: when an instruction reaches the head of the ROB and its result is present in the buffer
    - » The processor updates the register with the result and removes the instruction from the ROB.
  - Committing a store:
    - » is similar except that memory is updated rather than a result register.
  - A branch with incorrect prediction
    - » indicates that the speculation was wrong.
    - » The ROB is flushed and execution is restarted at the correct successor of the branch.

In-order Commit with Branch

```
Loop:   L.D     F0,0(R1)
        MUL.D   F4,F0,F2
        S.D     F4,0(R1)
        DADDIU  R1,R1,#-8
        BNE     R1,R2,Loop     ;branches if R1¦|
```

Reorder buffer

Entry	Busy	Instruction		State	Destination	Value
1	No	L.D	F0,0(R1)	Commit	F0	Mem[0 + Regs[R1]]
2	No	MUL.D	F4,F0,F2	Commit	F4	#1 × Regs[F2]
3	Yes	S.D	F4,0(R1)	Write result	0 + Regs[R1]	#2
4	Yes	DADDIU	R1,R1,#-8	Write result	R1	Regs[R1] – 8
5	Yes	BNE	R1,R2,Loop	Write result		
6	Yes	L.D	F0,0(R1)	Write result	F0	Mem[#4]
7	Yes	MUL.D	F4,F0,F2	Write result	F4	#6 × Regs[F2]
8	Yes	S.D	F4,0(R1)	Write result	0 + #4	#7
9	Yes	DADDIU	R1,R1,#-8	Write result	R1	#4 – 8
10	Yes	BNE	R1,R2,Loop	Write result		

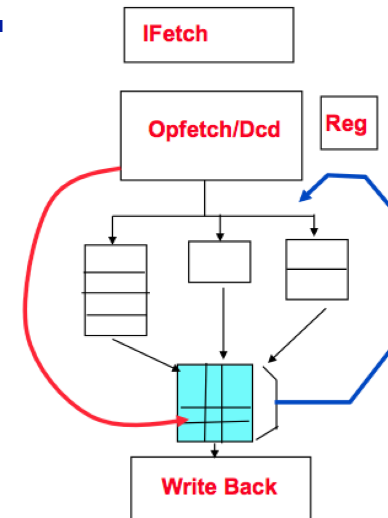
FLUSHED

IF Misprediction

FP register status

Field	F0	F1	F2	F3	F4	F5	F6	F7	F8
Reorder #	6				7				
Busy	Yes	No	No	No	Yes	No	No	...	No

Dynamic Scheduling and Speculation



- ILP Maximized (a restricted data-flow)
  - In-order issue
  - Out-of-order execution
  - Out-of-order completion
  - In-order commit
- Data Hazards
  - Input operands-driven dynamic scheduling for RAW hazard
  - Register renaming for handling WAR and WAW hazards
- Control Hazards (Branching, Precision Exception)
  - Branch prediction and in-order commit (speculation)
  - Branch prediction without speculation
    - » Cannot do out-of-order execution/complete for branch
- Implementation: Tomasulo
  - Reservation stations and Reorder buffer
  - Other solutions as well (scoreboard, history table)

MULTIPLE ISSUE VIA VLIW/ STATIC SUPERSCALAR

Multiple Issue

- "Flynn bottleneck"
  - single issue performance limit is CPI = IPC = 1
  - hazards + overhead ⇒ CPI >= 1 (IPC <= 1)
  - diminishing returns from superpipelining [Hrishikesh paper!]
- Solution: issue multiple instructions per cycle

	1	2	3	4	5	6	7
inst0	F	D	X	M	W		
inst1	F	D	X	M	W		
inst2		F	D	X	M	W	
inst3		F	D	X	M	W	

- 1st superscalar: IBM America → RS/6000 → POWER1

Multiple Issue

- Issue multiple instructions in one cycle
- Three major types (VLIW and superscalar)
  - Statically scheduled superscalar processors
  - VLIW (very long instruction word) processors
  - Dynamically scheduled superscalar processors
- Superscalar
  - Variable # of instr per cycle
  - In-order execution for static superscalar
  - Out-of-order execution for dynamic superscalar
- VLIW
  - Issue a fixed number of instructions formatted either as one large instruction or as a fixed instruction packet with the parallel- ism among instructions explicitly indicated by the instruction.
  - Inherently statically scheduled by the compiler
  - Intel/HP IA-64 architecture, named EPIC—explicitly parallel instruction computer
    - » Appendix H,

IF	ID	EX	MEM	WB			
IF	ID	EX	MEM	WB			
	IF	ID	EX	MEM	WB		
	IF	ID	EX	MEM	WB		
		IF	ID	EX	MEM	WB	
		IF	ID	EX	MEM	WB	
			IF	ID	EX	MEM	W
			IF	ID	EX	MEM	W
				IF	ID	EX	ME
				IF	ID	EX	ME

VLIW and Static Superscalar

- Very similar in terms of the requirements for compiler and hardware support
- We will discuss VLIW/Static superscalar

- Very Long Instruction Word (VLIW)
  - packages the multiple operations into one very long instruction

VLIW: Very Large Instruction Word

- **Each "instruction" has explicit coding for multiple operations**
  - In IA-64, grouping called a "packet"
  - In Transmeta, grouping called a "molecule" (with "atoms" as ops)
- **Tradeoff instruction space for simple decoding**
  - The long instruction word has room for many operations
  - By definition, all the operations the compiler puts in the long instruction word are independent => execute in parallel
  - E.g., 1 integer operation/branch, 2 FP ops, 2 Memory refs
    - » 16 to 24 bits per field => 5*16 or 80 bits to 5*24 or 120 bits wide
  - Need compiling technique that schedules across several branches

Recall: Unrolled Loop that Minimizes Stalls for Scalar

```
1  Loop: L.D    F0,0(R1)
2        L.D    F6,-8(R1)
3        L.D    F10,-16(R1)
4        L.D    F14,-24(R1)
5        ADD.D  F4,F0,F2
6        ADD.D  F8,F6,F2
7        ADD.D  F12,F10,F2
8        ADD.D  F16,F14,F2
9        S.D    0(R1),F4
10       S.D    -8(R1),F8
11       S.D    -16(R1),F12
12       DSUBUI R1,R1,#32
13       BNEZ   R1,LOOP
14       S.D    8(R1),F16      ; 8-32 = -24
```

L.D to ADD.D: 1 Cycle
ADD.D to S.D: 2 Cycles

```
for (i=999; i>=0; i=i-1)
    x[i] = x[i] + s;
```

Instruction producing result	Instruction using result	Latency in clock cycles
FP ALU op	Another FP ALU op	3
FP ALU op	Store double	2
Load double	FP ALU op	1
Load double	Store double	0

14 clock cycles, or 3.5 per iteration

Loop Unrolling in VLIW

Unrolled 7 times to avoid delays

- **7 results in 9 clocks, or 1.3 clocks per iteration (1.8X)**
- **Average: 2.5 ops per clock, 50% efficiency**

Memory reference 1	Memory reference 2	FP operation 1	FP operation 2	Integer operation/branch
L.D F0,0(R1)	L.D F6,-8(R1)			
L.D F10,-16(R1)	L.D F14,-24(R1)			
L.D F18,-32(R1)	L.D F22,-40(R1)	ADD.D F4,F0,F2	ADD.D F8,F6,F2	
L.D F26,-48(R1)		ADD.D F12,F10,F2	ADD.D F16,F14,F2	
		ADD.D F20,F18,F2	ADD.D F24,F22,F2	
S.D F4,0(R1)	S.D F8,-8(R1)	ADD.D F28,F26,F2		
S.D F12,-16(R1)	S.D F16,-24(R1)			DADDUI R1,R1,#-56
S.D F20,24(R1)	S.D F24,16(R1)			
S.D F28,8(R1)				BNE R1,R2,Loop

Figure 3.16 VLIW instructions that
cycles assuming no branch delay; norr
ations in 9 clock cycles, or 2.5 operati
operation, is about 60%. To achieve thi
this loop. The VLIW code sequence abc
MIPS processor can use as few as two l

Instruction producing result	Instruction using result	Latency in clock cycles
FP ALU op	Another FP ALU op	3
FP ALU op	Store double	2
Load double	FP ALU op	1
Load double	Store double	0

Loop Unrolling in VLIW

- Unroll 8 times
  - Enough registers

8 results in 9 clocks, or 1.125 clocks per iteration

Average: 2.89 (26/9) ops per clock, 58% efficiency (26/45)

Memory reference 1	Memory reference 2	FP operation 1	FP operation 2	Integer operation/branch
L.D F0,0(R1)	L.D F6,-8(R1)			
L.D F10,-16(R1)	L.D F14,-24(R1)			
L.D F18,-32(R1)	L.D F22,-40(R1)	ADD.D F4,F0,F2	ADD.D F8,F6,F2	
L.D F26,-48(R1)	L.D	ADD.D F12,F10,F2	ADD.D F16,F14,F2	
		ADD.D F20,F18,F2	ADD.D F24,F22,F2	
S.D F4,0(R1)	S.D F8,-8(R1)	ADD.D F28,F26,F2	ADD.D	
S.D F12,-16(R1)	S.D F16,-24(R1)			DADDUI R1,R1,#-56
S.D F20,24(R1)	S.D F24,16(R1)			
S.D F28,8(R1)	S.D			BNE R1,R2,Loop

Figure 3.16 VLIW instructions that cycles assuming no branch delay; norr ations in 9 clock cycles, or 2.5 operati operation, is about 60%. To achieve thi this loop. The VLIW code sequence abc MIPS processor can use as few as two l

Instruction producing result	Instruction using result	Latency in clock cycles
FP ALU op	Another FP ALU op	3
FP ALU op	Store double	2
Load double	FP ALU op	1
Load double	Store double	0

Loop Unrolling in VLIW

Instruction producing result	Instruction using result	Latency in clock cycles
FP ALU op	Another FP ALU op	3
FP ALU op	Store double	2
Load double	FP ALU op	1
Load double	Store double	0

- Unroll 10 times
  - Enough registers

10 results in 10 clocks, or 1 clock per iteration

Average: 3.2 ops per clock (32/10), 64% efficiency (32/50)

Memory reference 1	Memory reference 2	FP operation 1	FP operation 2	Integer operation/branch
L.D F0,0(R1)	L.D F6,-8(R1)			
L.D F10,-16(R1)	L.D F14,-24(R1)			
L.D F18,-32(R1)	L.D F22,-40(R1)	ADD.D F4,F0,F2	ADD.D F8,F6,F2	
L.D F26,-48(R1)	**L.D**	ADD.D F12,F10,F2	ADD.D F16,F14,F2	
L.D	**L.D**	ADD.D F20,F18,F2	ADD.D F24,F22,F2	
S.D F4,0(R1)	S.D F8,-8(R1)	ADD.D F28,F26,F2	**ADD.D**	
S.D F12,-16(R1)	S.D F16,-24(R1)	**ADD.D**	**ADD.D**	
S.D F20,24(R1)	S.D F24,16(R1)			DADDUI R1,R1,#-56
S.D F28,8(R1)	**S.D**			
S.D	**S.D**			BNE R1,R2,Loop

Problems with 1st Generation VLIW

- Increase in code size
  - generating enough operations in a straight-line code fragment requires ambitiously unrolling loops
  - whenever VLIW instructions are not full, unused functional units translate to wasted bits in instruction encoding
- Operated in lock-step; no hazard detection HW
  - a stall in any functional unit pipeline caused entire processor to stall, since all functional units must be kept synchronized
  - Compiler might prediction function units, but caches hard to predict
- Binary code compatibility
  - Pure VLIW => different numbers of functional units and unit latencies require different versions of the code

- **Itanium/EPIC/VLIW is not a breakthrough in ILP**
  - **If anything, it is as complex or more so than a dynamic processor**

Very Important Terms

- Dynamic Scheduling → Out-of-order Execution
- Speculation → In-order Commit
- Superscalar → Multiple Issue

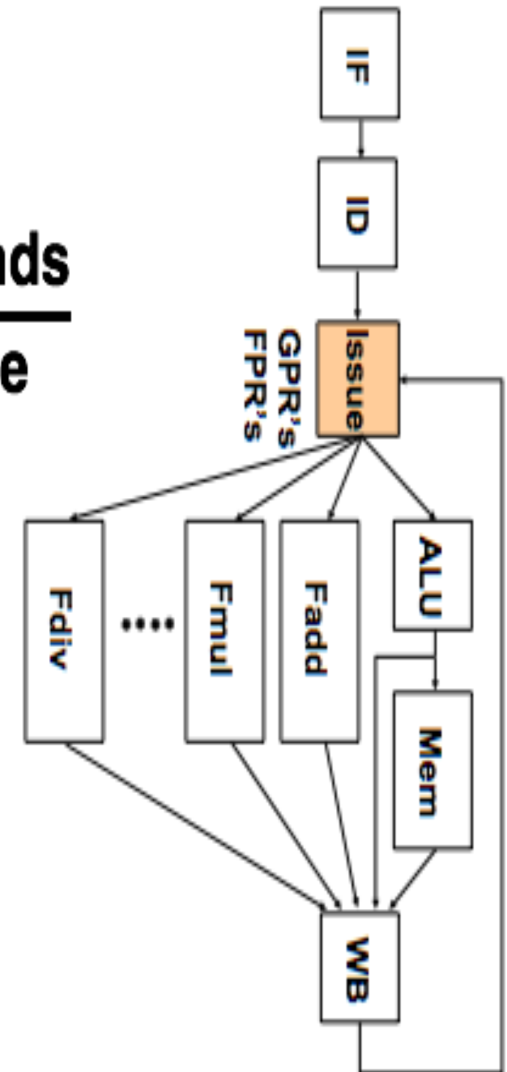
Techniques	Goals	Implementation	Addressing	Approaches
Dynamic Scheduling	Out-of-order execution	Reservation Stations, Load/Store Buffer and CDB	Data hazards (RAW, WAW, WAR)	Register renaming
Speculation	In-order commit	Reorder Buffer	Control hazards	Prediction and misprediction recovery
Superscalar/ VLIW	Multiple issue	Software and Hardware	To Increase CPI	By compiler or hardware

DYNAMIC SCHEDULING, MULTIPLE ISSUE (DYNAMIC SUPERSCALAR), AND SPECULATION

Dynamic Scheduling, Multiple Issue, and Speculation

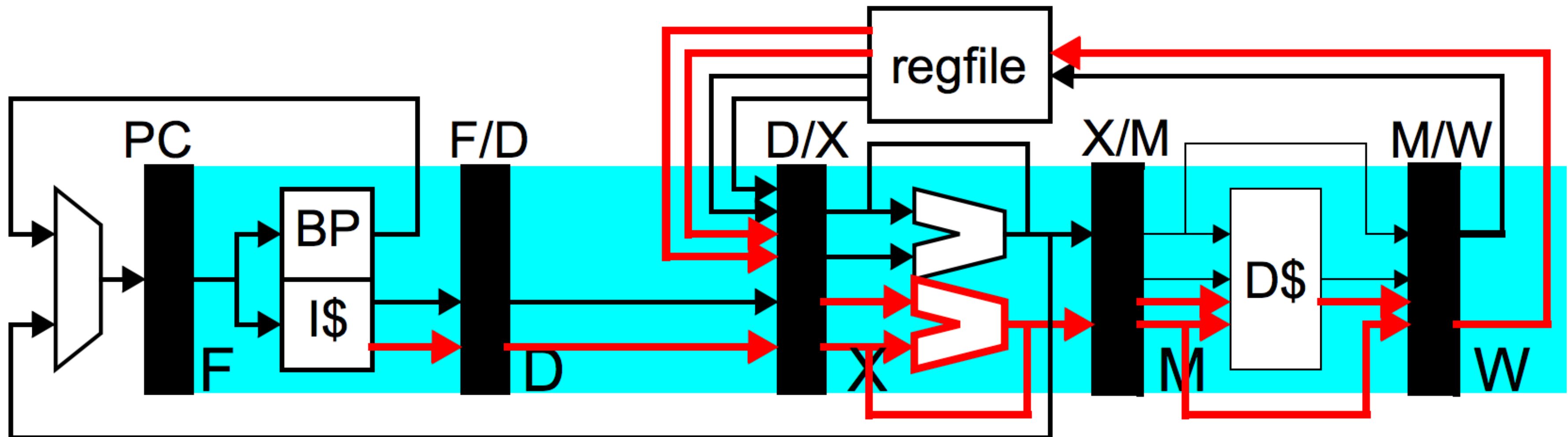
- Microarchitecture quite similar to those in modern microprocessors
  - Real

$$\frac{\text{Seconds}}{\text{Program}} = \frac{\text{Instructions}}{\text{Program}} \; \frac{\text{Cycles}}{\text{Instruction}} \; \frac{\text{Seconds}}{\text{Cycle}}$$



- Consider two issue per clock
  - Example: CPU with floating point ALUs: Issue 1 FP + 1 Integer instruction per cycle.
    - » Save at least 1 cycle than the pipeline
  - Challenges
    - » Find the right instructions
    - » Dependency between instructions

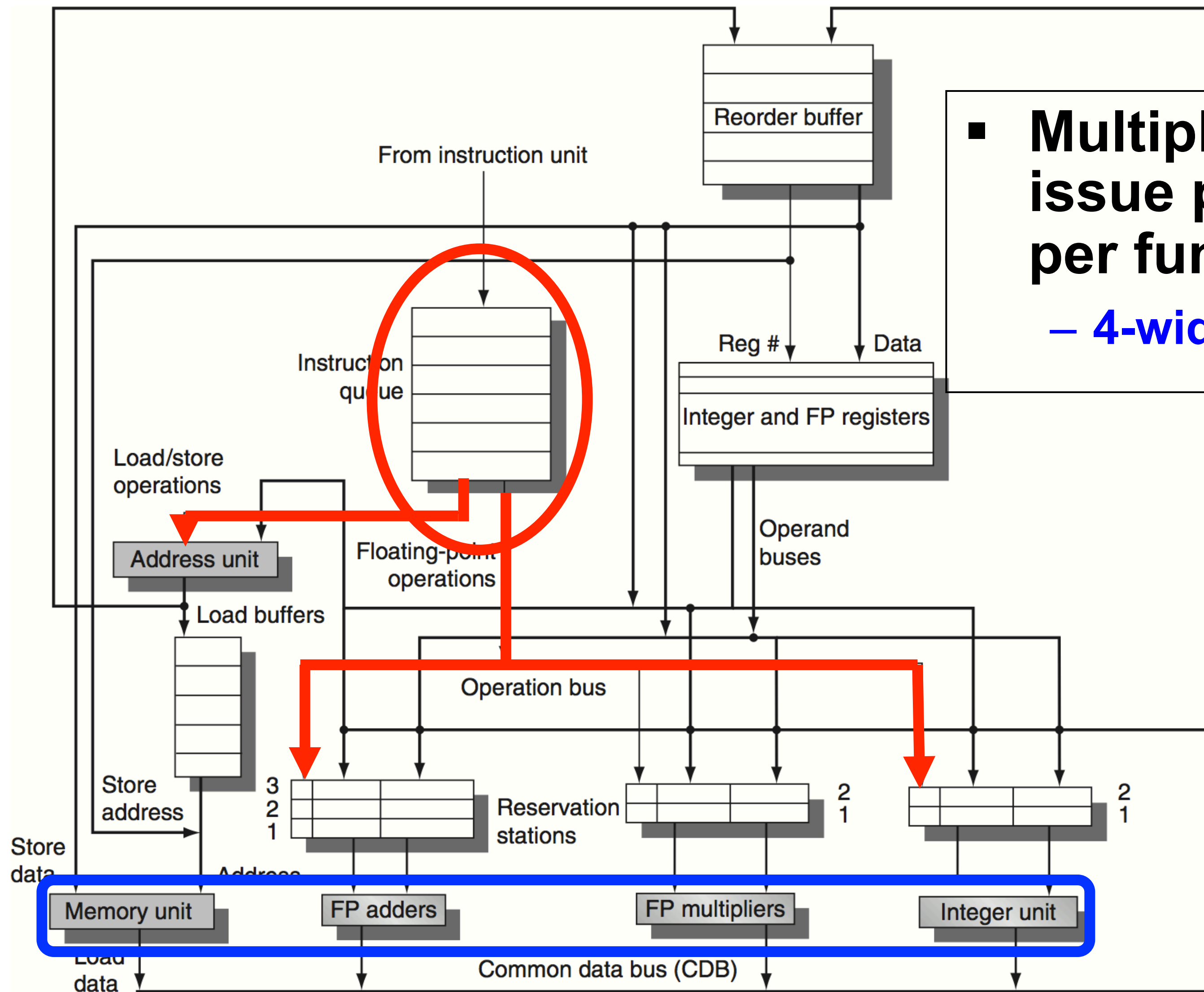
5-Stage In-order 2-Wide Pipeline



- what is involved in
  - fetching two instructions per cycle?
  - decoding two instructions per cycle?
  - executing two ALU operations per cycle?
  - accessing the data cache twice per cycle?
  - writing back two results per cycle?
- what about 4 or 8 instructions per cycle?

Implementation using Temasulo's Approach

- **Similar to Tomasulo with Speculation**



- **Multiple issue → one issue per clock cycle per functional unit**
  - 4-wide

Options and Challenges of Multiple Issue

- How to issue two instructions and keep in-order instruction issue for Tomasulo?
  - Assume 1 integer + 1 floating point
  - 1 Tomasulo control for integer, 1 for floating point

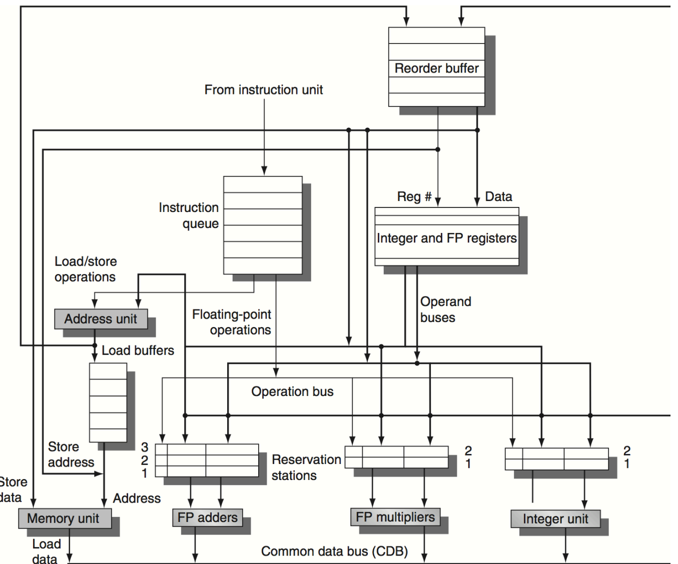
1. Issue two instrs pipelined in one cycle (half and half for each instr), so that issue remains in order → superpipelining
   - Hard to extend to 4 or more
2. Issue 2 instrs per cycle in parallel → true superscalar
   - Between FP and Integer operations: Only FP loads might cause dependency between integer and FP issue:
     - » Replace load reservation station with a load queue; operands must be read in the order they are fetched
     - » Load checks addresses in Store Queue to avoid RAW violation
     - » Store checks addresses in Load Queue to avoid WAR,WAW
     - » Called "decoupled architecture"
3. Mix of both
   - Superpipeling and superscalar

Multiple Issue Challenges

- While Integer/FP split is simple for the HW, get CPI of 0.5 only for programs with:
  – Exactly 50% FP operations
  – No hazards
- If more instructions issue at same time, greater difficulty of decode and issue:
  – Even 2-scalar => examine 2 opcodes, 6 register specifiers, & decide if 1 or 2 instructions can issue
  – Multiported rename logic: must be able to rename same register multiple times in one cycle!
  – Rename logic one of key complexities in the way of multiple issue!

Multiple Issue

- **Bundle multiple instrs in one issue unit**
  - ***N*-wide**

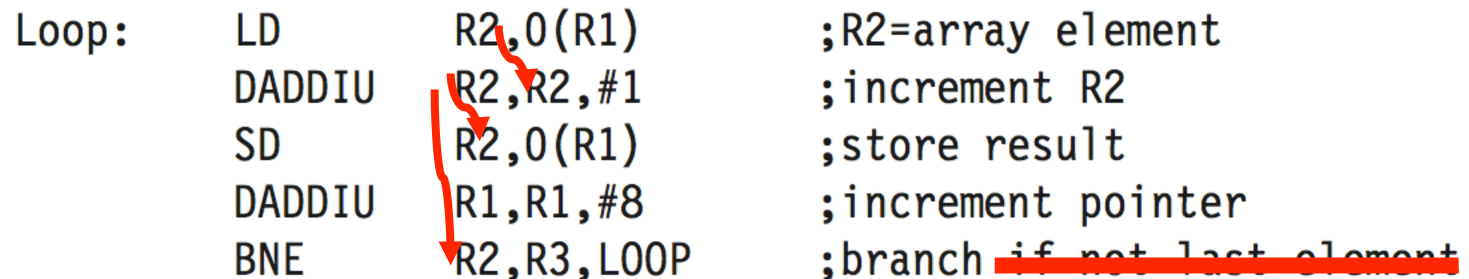


1. **Assign a reservation station and a reorder buffer for *every* instruction that *might* be issued in the next issue bundle**
   - **N entries in ROB**
   - **Ensure enough RS available for the bundle**
   - **If not enough RS/ROB, break the bundle**
2. **Analyze dependency in the issue bundle**
3. **Inter-dependency between instrs in a bundle**
   - **Update the reservation station table entries using the assigned ROB entries to link the dependency**
     - » **Register renaming happened**

- **In-order commit to make sure instrs commit in order**
- **Other techniques**
  - **Speculative multiple issue in Intel i7**

Example

Example Consider the execution of the following loop, ~~which increments each element of an integer array~~, on a two-issue processor, once without speculation and once with speculation:



```
Loop:    LD       R2,0(R1)       ;R2=array element
         DADDIU   R2,R2,#1       ;increment R2
         SD       R2,0(R1)       ;store result
         DADDIU   R1,R1,#8       ;increment pointer
         BNE      R2,R3,LOOP     ;branch if not last element
```

Assume that there are separate integer functional units for effective address calculation, for ALU operations, and for branch condition evaluation. Create a table for the first three iterations of this loop for both processors. Assume that up to two instructions of any type can commit per clock.

BNE has RAW dependence on DADDIU

Without Speculation

LD can be issued but CANNOT be executed before BNE completes

Iteration number	Instructions	Issues at clock cycle number	Executes at clock cycle number	access at clock cycle number	Write CDB at clock cycle number	Comment
1	LD R2,0(R1)	1	2	3	4	First issue
1	DADDIU R2,R2,#1	1	5		6	Wait for LW
1	SD R2,0(R1)	2	3	7		Wait for DADDIU
1	DADDIU R1,R1,#8	2	3		4	Execute directly
1	BNE R2,R3,LOOP	3	7			Wait for DADDIU
2	LD R2,0(R1)	4	8	9	10	Wait for BNE
2	DADDIU R2,R2,#1	4	11		12	Wait for LW
2	SD R2,0(R1)	5	9	13		Wait for DADDIU
2	DADDIU R1,R1,#8	5	8		9	Wait for BNE
2	BNE R2,R3,LOOP	6	13			Wait for DADDIU
3	LD R2,0(R1)	7	14	15	16	Wait for BNE
3	DADDIU R2,R2,#1	7	17		18	Wait for LW
3	SD R2,0(R1)	8	15	19		Wait for DADDIU
3	DADDIU R1,R1,#8	8	14		15	Wait for BNE
3	BNE R2,R3,LOOP	9	19			Wait for DADDIU

No Speculation

No Speculation

Figure 3.19 The time of issue, execution, and writing result for a dual-issue version of our pipeline *without* speculation. Note that the LD following the BNE cannot start execution earlier because it must wait until the branch

With Speculation

LD can be speculatively executed before BNE completes

Iteration number	Instructions	Issues at clock number	Execute at clock number	at clock number	clock number	at clock number	Comment
1	LD R2,0(R1)	1	2	3	4	5	First issue
1	DADDIU R2,R2,#1	1	5		6	7	Wait for LW
1	SD R2,0(R1)	2	3			7	Wait for DADDIU
1	DADDIU R1,R1,#8	2	3		4	8	Commit in order
1	BNE R2,R3,LOOP	3	7			8	Wait for DADDIU
2	LD R2,0(R1)	4	5	6	7	9	No execute delay
2	DADDIU R2,R2,#1	4	8		9	10	Wait for LW
2	SD R2,0(R1)	5	6			10	Wait for DADDIU
2	DADDIU R1,R1,#8	5	6		7	11	Commit in order
2	BNE R2,R3,LOOP	6	10			11	Wait for DADDIU
3	LD R2,0(R1)	7	8	9	10	12	Earliest possible
3	DADDIU R2,R2,#1	7	11		12	13	Wait for LW
3	SD R2,0(R1)	8	9			13	Wait for DADDIU
3	DADDIU R1,R1,#8	8	9		10	14	Executes earlier
3	BNE R2,R3,LOOP	9	13			14	Wait for DADDIU

With Speculation

Figure 3.20 The time of issue, execution, and writing result for a dual-issue version of our pipeline *with* speculation. Note that the LD following the BNE can start execution early because it is speculative.

ADVANCED TECHNIQUES FOR INSTRUCTION DELIVERY AND SPECULATION

1. Advanced Branch Prediction
2. Explicit Register Renaming
3. Others that are important but not covered: Load/store speculation, value predication, correlate branch prediction, tournament predictor, trace cache
4. Put all together on ARM Cortex-A8 and Intel Core i7

BRANCH PREDICTION BEFORE DECODING

Branch History Table for Dynamic Branch Prediction

- **Solution: 2-bit scheme where change prediction only if get misprediction *twice*.**

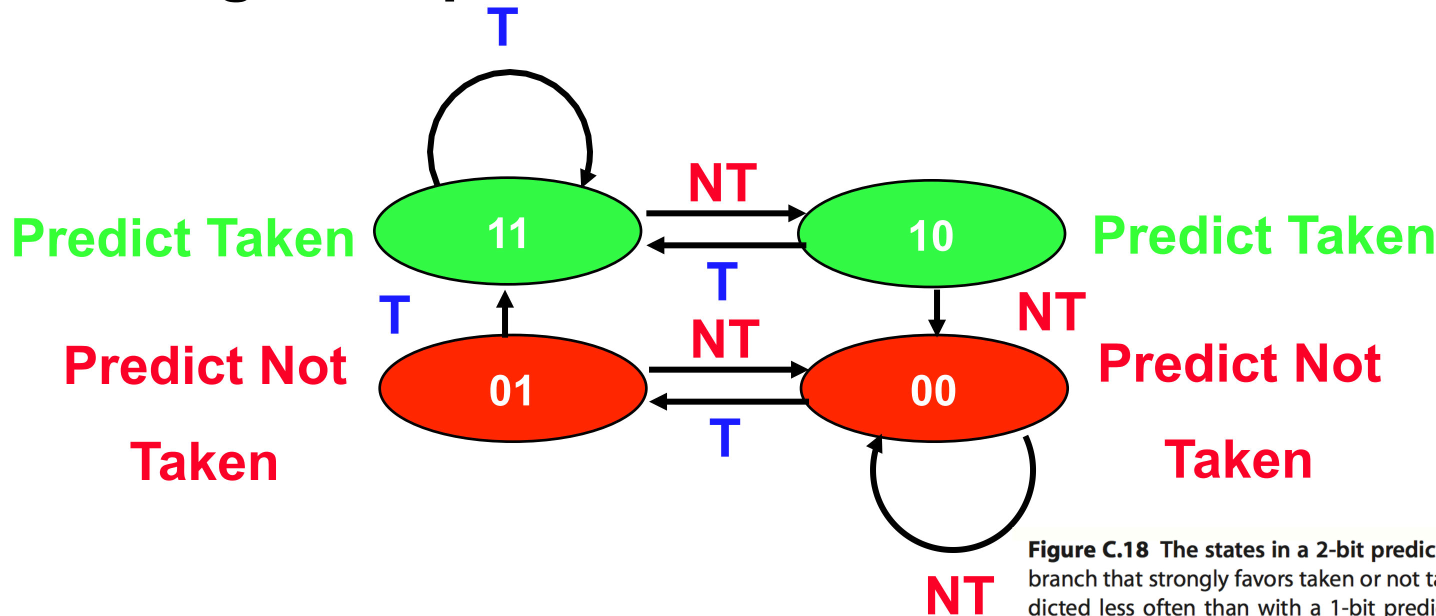
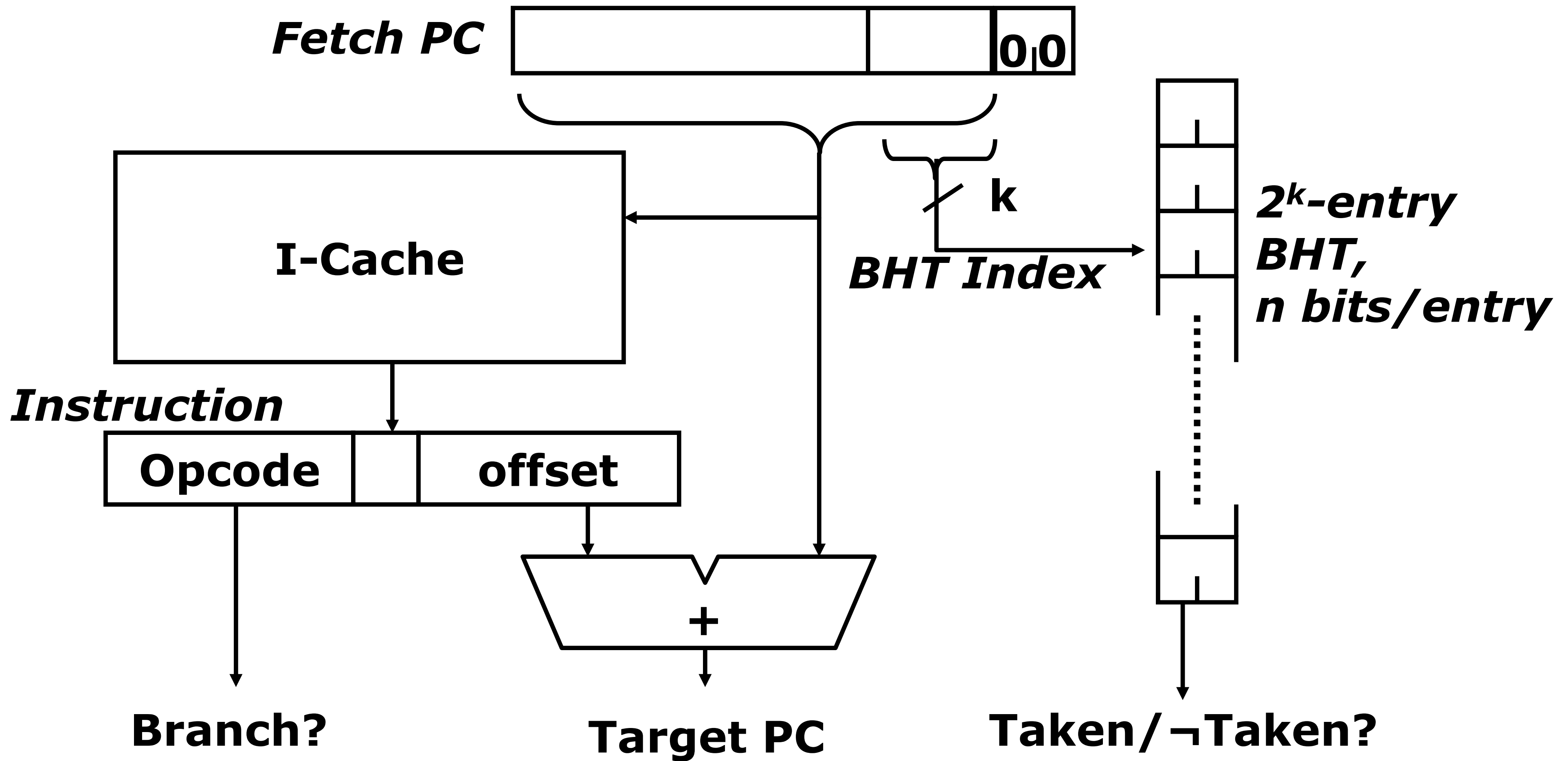


Figure C.18 The states in a 2-bit prediction scheme. By using 2 bits rather than 1, a branch that strongly favors taken or not taken—as many branches do—will be mispredicted less often than with a 1-bit predictor. The 2 bits are used to encode the four states in the system. The 2-bit scheme is actually a specialization of a more general scheme that has an *n*-bit saturating counter for each entry in the prediction buffer. With an *n*-bit counter, the counter can take on values between 0 and $2^n - 1$: When the counter is greater than or equal to one-half of its maximum value ($2^n - 1$), the branch is predicted as taken; otherwise, it is predicted as untaken. Studies of *n*-bit predictors have shown that the 2-bit predictors do almost as well, thus most systems rely on 2-bit branch predictors rather than the more general *n*-bit predictors.

- **Red: stop, not taken;**
- **Blue: go, taken;**
- **Adds *hysteresis* to decision making process.**

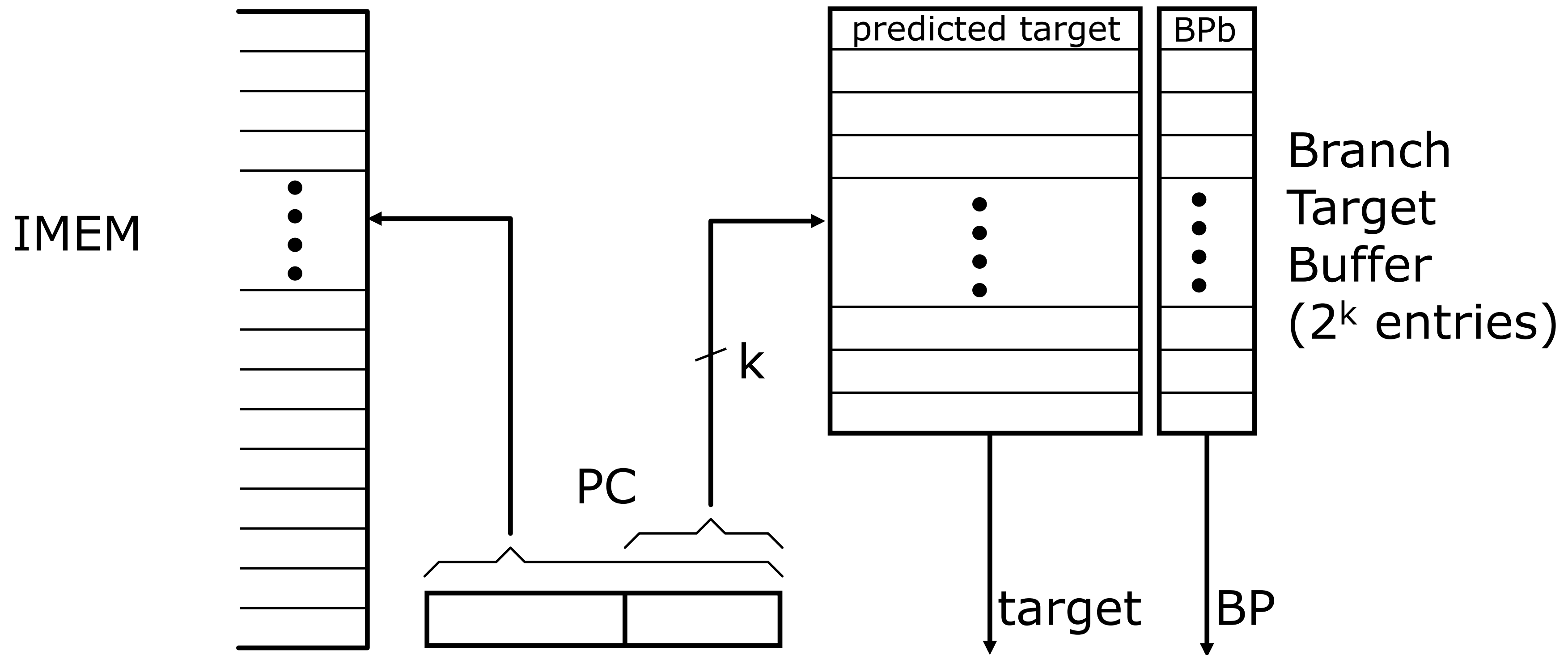
42

Typical Branch History Table



4K-entry BHT, 2 bits/entry, ~80-90% correct predictions

Branch Target Buffer



BP bits are stored with the predicted target address. (BHT)

IF stage: *If (BP=taken) then nPC=target else nPC=PC+4*
later: *check prediction, if wrong then kill the instruction and update BTB & BPb else update BPb*

Branch Target Buffer (BTB)

2^k-entry direct-mapped BTB
(can also be associative)

I-Cache

PC

k

Entry PC

Valid

Predicted target pc

=

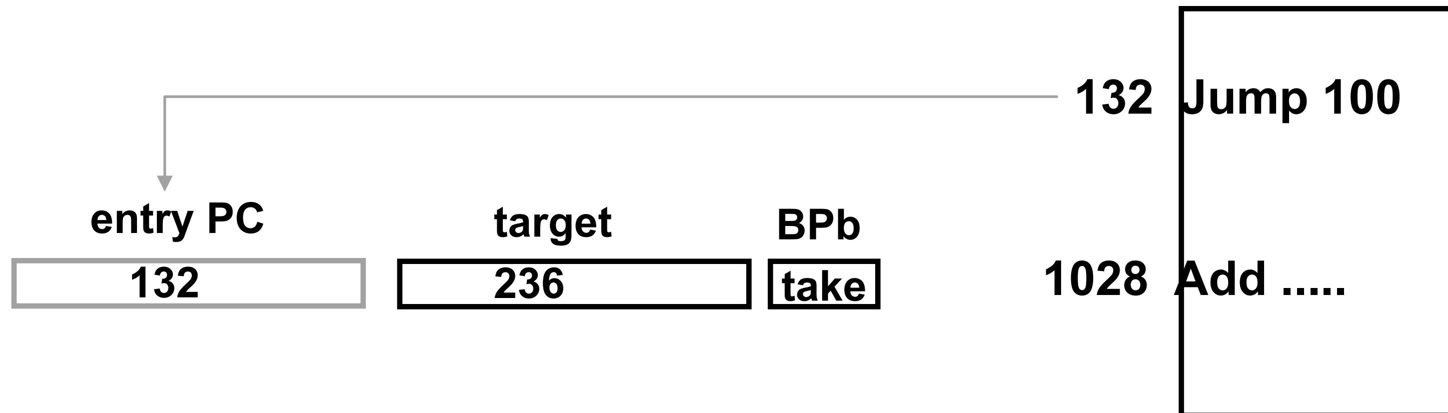
match

valid

target

- **Keep both the branch PC and target PC in the BTB**
- **PC+4 is fetched if match fails**
- **Only *predicted taken* branches and jumps held in BTB**
- **Next PC determined *before* branch fetched and decoded**

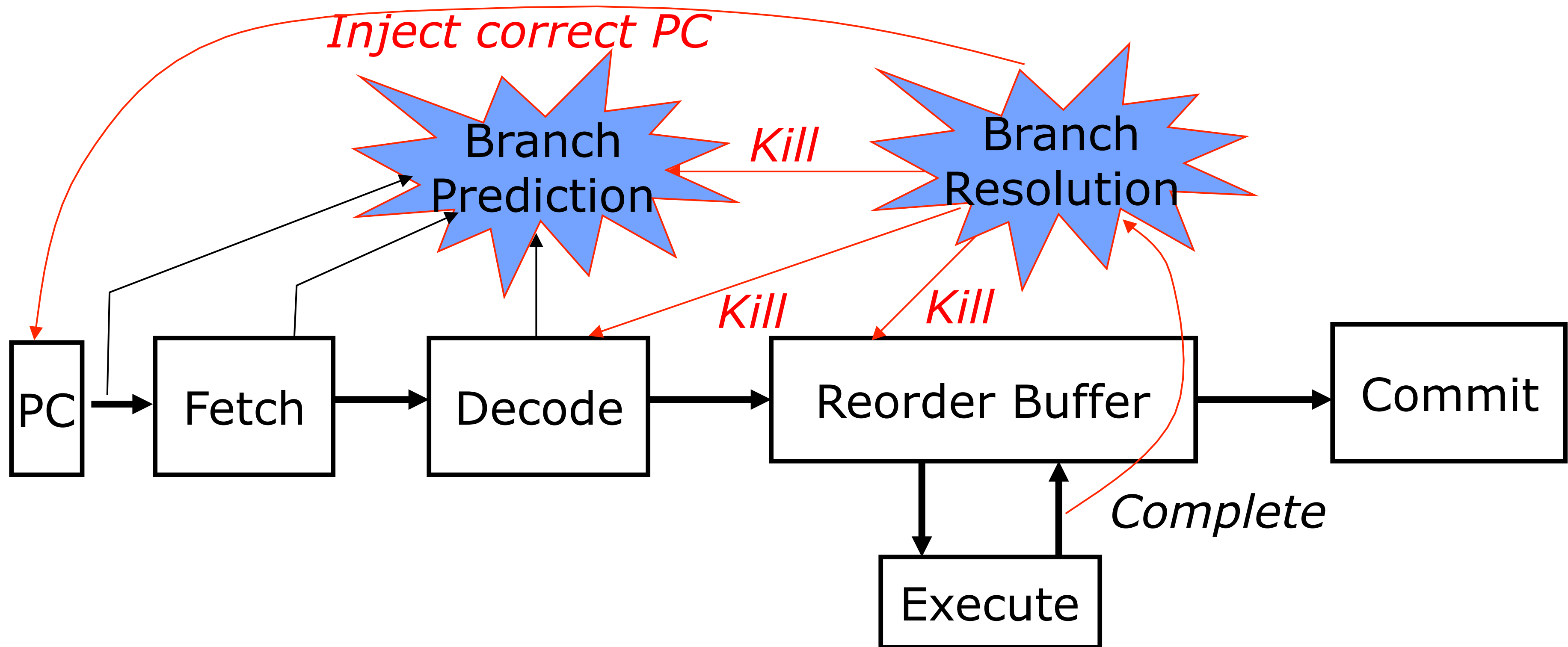
Consulting BTB Before Decoding



- **The match for PC=1028 fails and 1028+4 is fetched**
  ⇒ ***eliminates false predictions after ALU instructions***

- **BTB contains entries only for control transfer instructions**
  ⇒ ***more room to store branch targets***

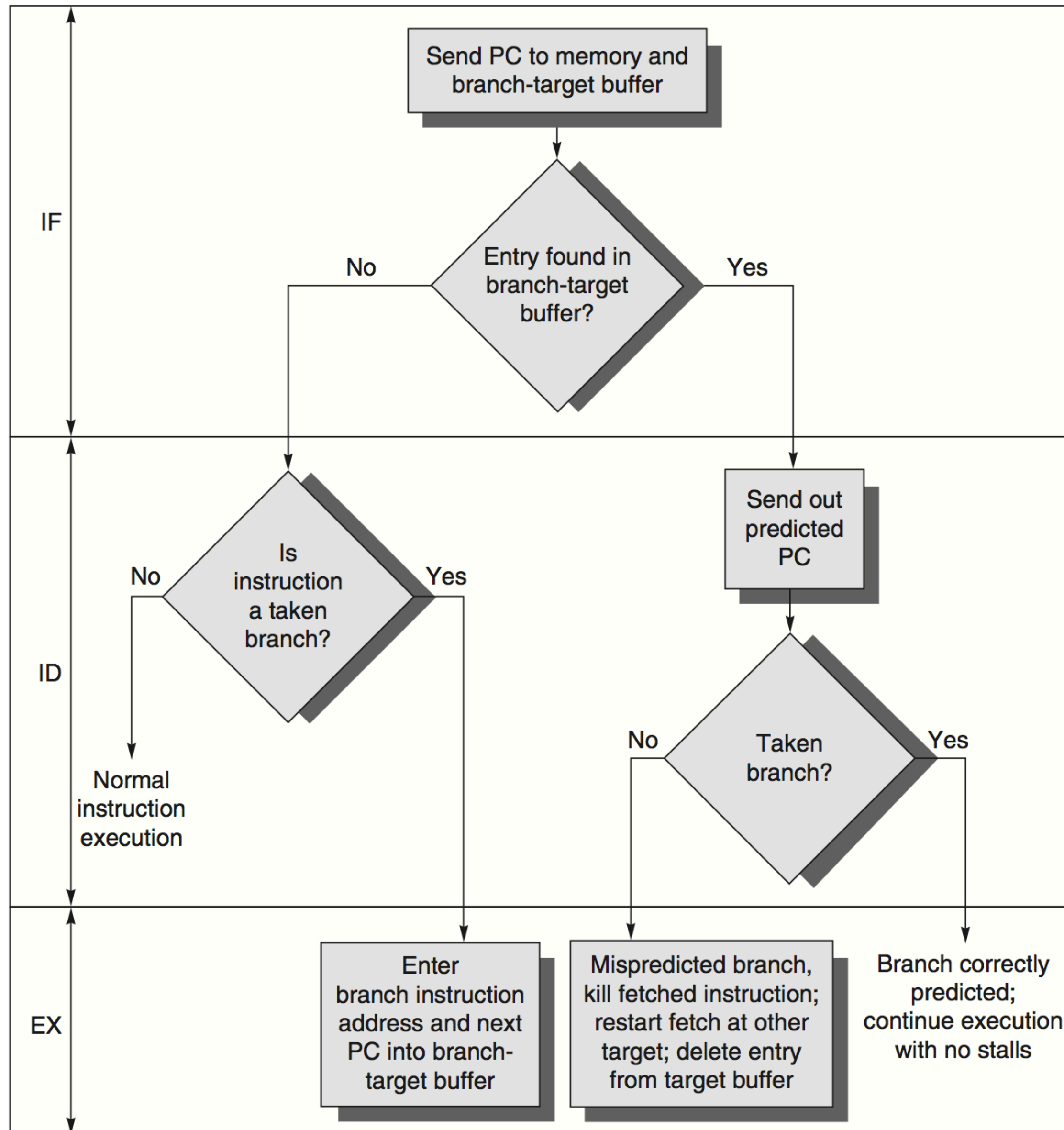
Branch Misprediction in Pipeline



- **Can have multiple unresolved branches in ROB**
- **Can resolve branches out-of-order by killing all the instructions in ROB that follow a mispredicted branch**

Branch With a Target Buffer

- **Steps**



Subroutine Return Stack

Small structure to accelerate JR for subroutine returns, typically much more accurate than BTBs.

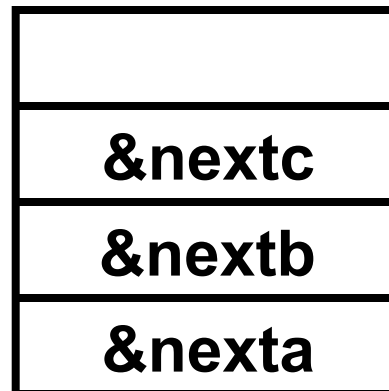
fa() { fb(); nexta: }

fb() { fc(); nextb: }

fc() { fd(); nextc: }

Push return address when function call executed

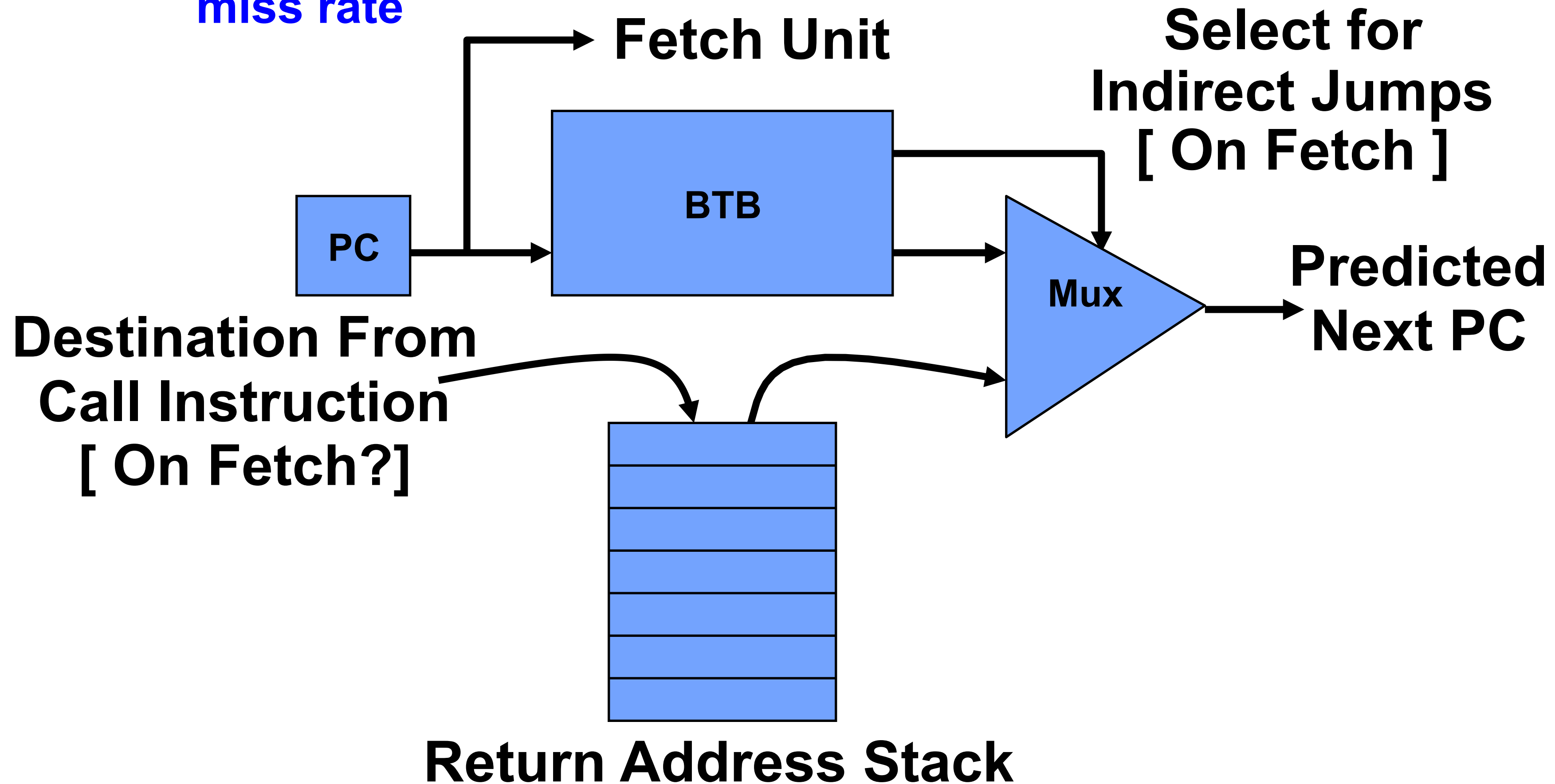
Pop return address when subroutine return decoded



k entries (typically k=8-16)

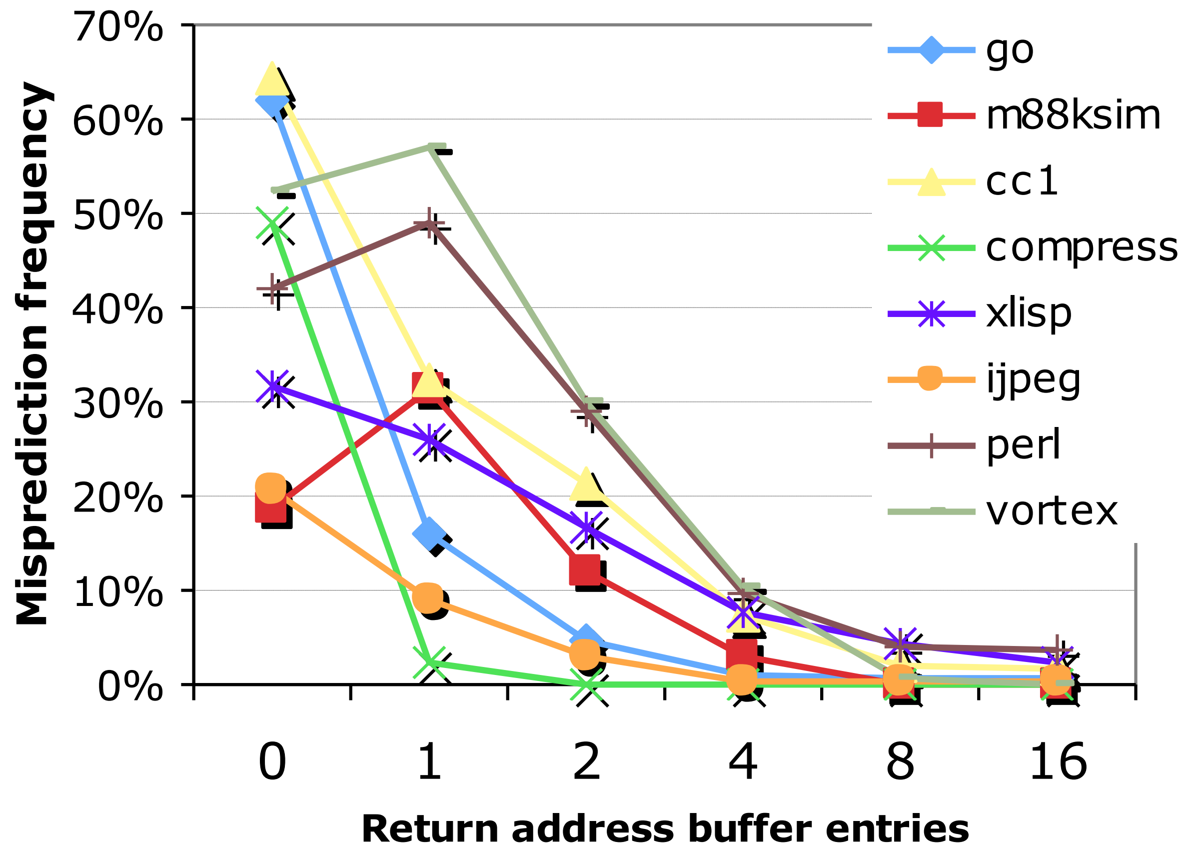
Special Case Return Addresses

- Register Indirect branch hard to predict address
  - SPEC89 85% such branches for procedure return
  - Since stack discipline for procedures, save return address in small buffer that acts like a stack: 8 to 16 entries has small miss rate



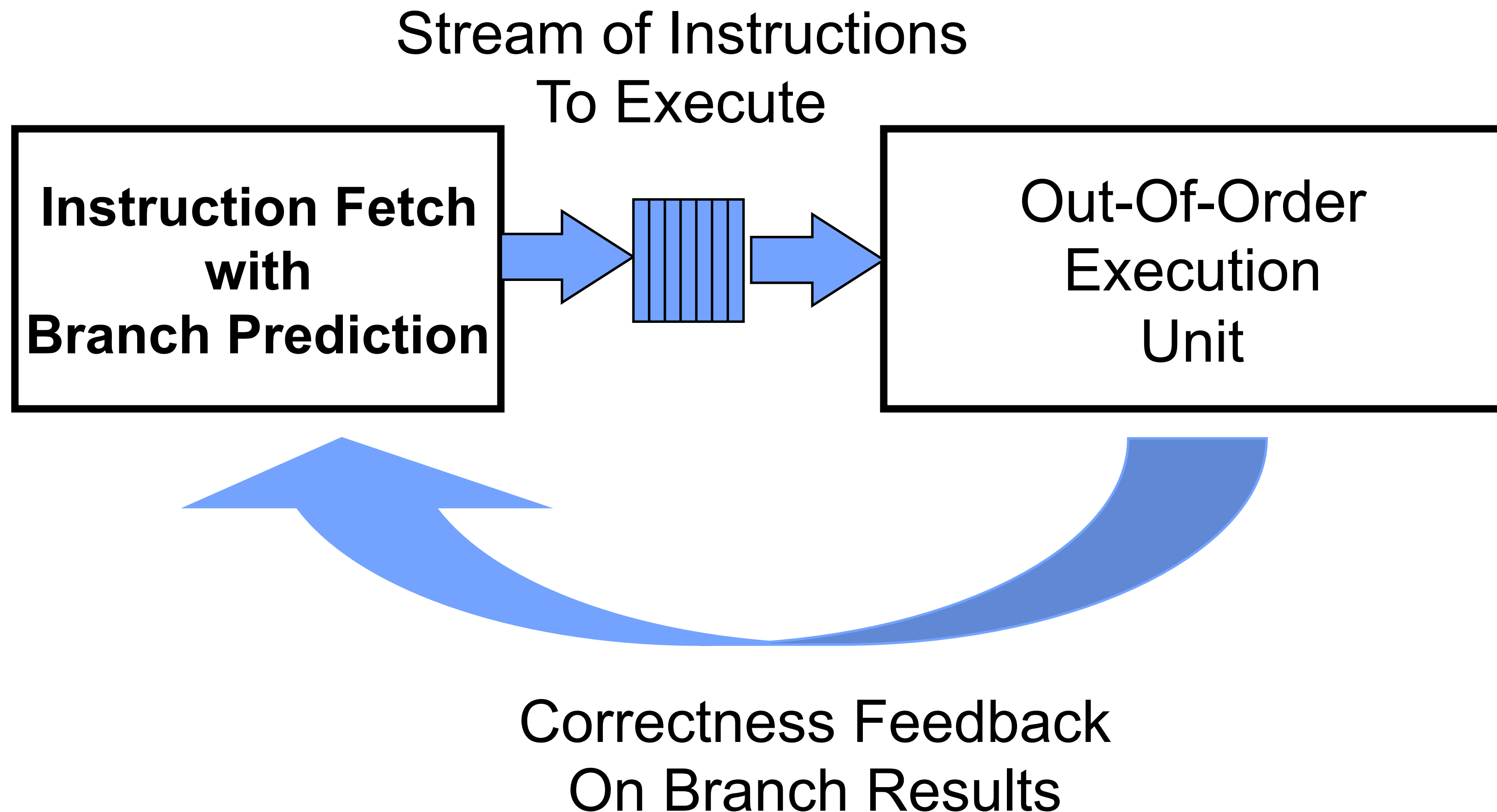
Performance: Return Address Predictor

- **Cache most recent return addresses:**
  - **Call ⇒ Push a return address on stack**
  - **Return ⇒ Pop an address off stack & predict as new PC**



Independent "Fetch" unit

- **Instruction fetch decoupled from execution**
  - **Instruction Buffer in-between**
- **Often issue logic (+ rename) included with Fetch**



EXPLICIT REGISTER RENAMING

Explicit Register Renaming

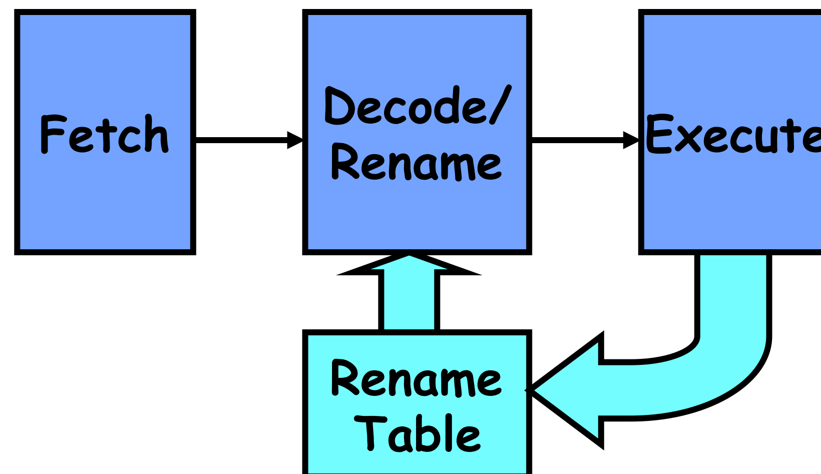
- **Tomasulo provides *Implicit Register Renaming***
  - **User registers renamed to reservation station tags**
- **Explicit Register Renaming:**
  - **Use *physical* register file that is larger than number of registers specified by ISA**
- **Keep a translation table:**
  - **ISA register => physical register mapping**
  - **When register is written, replace table entry with new register from freelist.**
  - **Physical register becomes free when not being used by any instructions in progress.**
- **Pipeline can be exactly like "standard" DLX pipeline**
  - **IF, ID, EX, etc….**
- **Advantages:**
  - **Removes all WAR and WAW hazards**
  - **Like Tomasulo, good for allowing full out-of-order completion**
  - **Allows data to be fetched from a single register file**
  - **Makes speculative execution/precise interrupts easier:**
    - » **All that needs to be "undone" for precise break point is to undo the table mappings**

Explicit Renaming Support Includes:

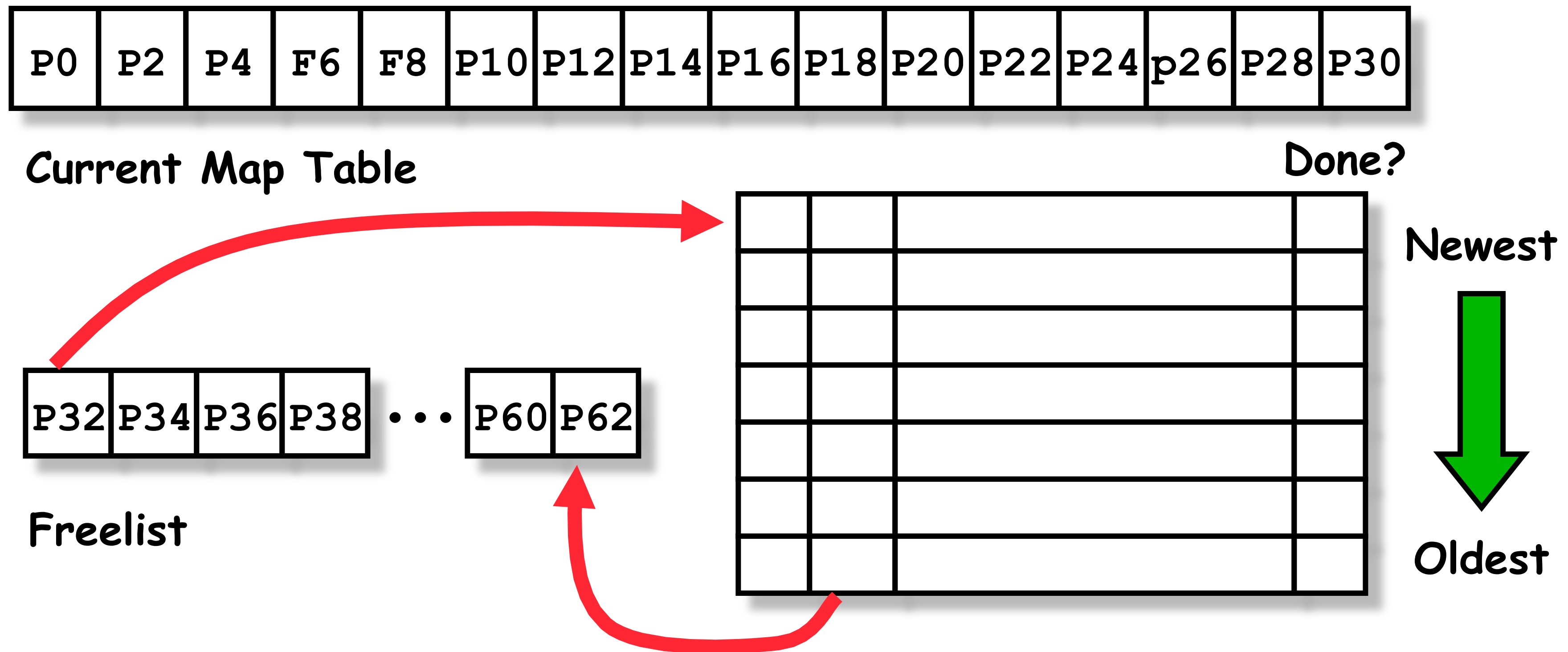
- Rapid access to a table of translations
- A physical register file that has more registers than specified by the ISA
- Ability to figure out which physical registers are free.
  - No free registers ⇒ stall on issue
- Thus, register renaming doesn't require reservation stations. However:
  - Many modern architectures use explicit register renaming + Tomasulo-like reservation stations to control execution.

Explicit Register Renaming

- Make use of a *physical* register file that is larger than number of registers specified by ISA
- Keep a translation table:
  - ISA register => physical register mapping
  - When register is written, replace table entry with new register from freelist.
  - Physical register becomes free when not being used by any instructions in progress.



Explicit register renaming:
R10000 Freelist Management



- Physical register file larger than ISA register file
- On issue, each instruction that modifies a register is allocated new physical register from freelist
- Used on: R10000, Alpha 21264, HP PA8000

Explicit register renaming:
R10000 Freelist Management

P32	P2	P4	F6	F8	P10	P12	P14	P16	P18	P20	P22	P24	p26	P28	P30

Current Map Table

P34	P36	P38	P40	•••	P60	P62

Freelist

Done?

Newest

F0	P0	LD P32,10(R2)	N

Oldest

- Note that physical register P0 is "dead" (or not "live") past the point of this load.
  - When we go to commit the load, we free up

Explicit register renaming:
R10000 Freelist Management

P32	P2	P4	P6	P8	P34	P12	P14	P16	P18	P20	P22	P24	P26	P28	P30

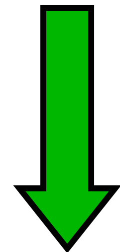
Current Map Table

P36	P38	P40	P42	•••	P60	P62

Freelist

			Done?
F10	P10	ADDD P34,P4,P32	N
F0	P0	LD P32,10(R2)	N

Newest



Oldest

Explicit register renaming:
R10000 Freelist Management

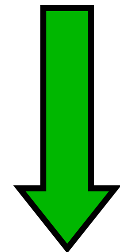
P32	P36	P4	F6	F8	P34	P12	P14	P16	P18	P20	P22	P24	p26	P28	P30

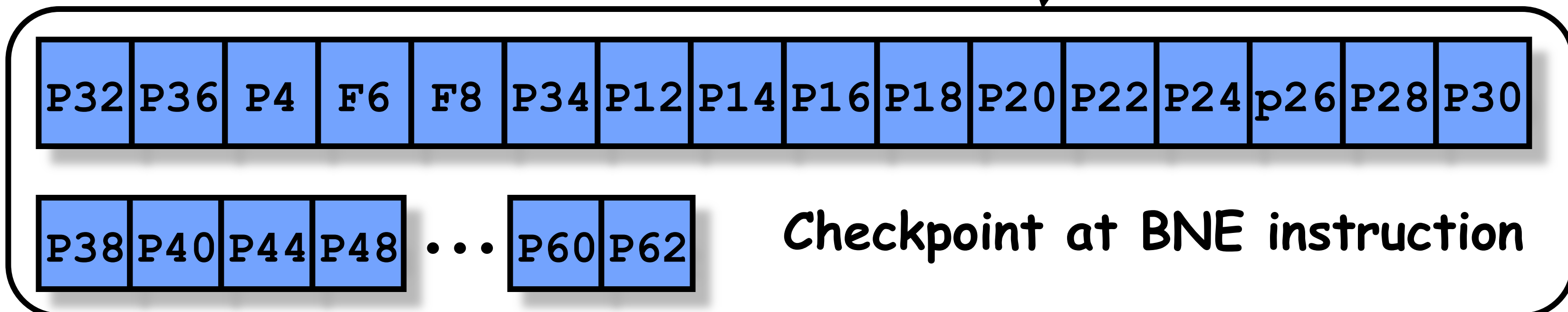
Current Map Table

P38	P40	P44	P48	•••	P60	P62

Freelist

			Done?
--			
--		BNE P36,<...>	N
F2	P2	DIVD P36,P34,P6	N
F10	P10	ADDD P34,P4,P32	N
F0	P0	LD P32,10(R2)	N





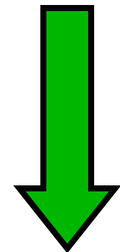
Explicit register renaming:
R10000 Freelist Management

P40	P36	P38	F6	F8	P34	P12	P14	P16	P18	P20	P22	P24	p26	P28	P30

Current Map Table

			Done?
--		ST 0(R3),P40	Y
F0	P32	ADDD P40,P38,P6	Y
F4	P4	LD P38,0(R3)	Y
--		BNE P36,<…>	N
F2	P2	DIVD P36,P34,P6	N
F10	P10	ADDD P34,P4,P32	y
F0	P0	LD P32,10(R2)	y

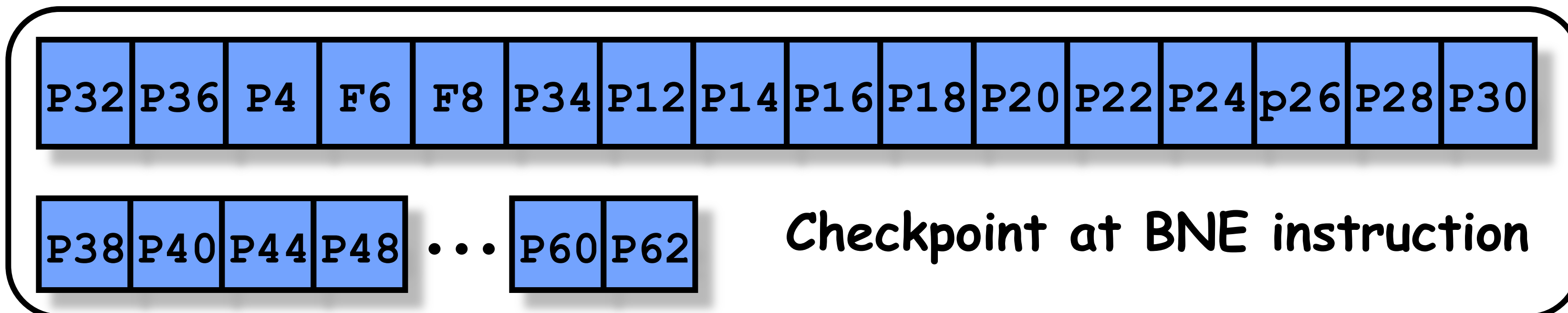
Newest



Oldest

P42	P44	P48	P50	•••	P0	P10

Freelist



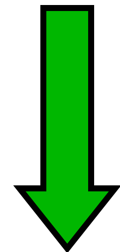
Explicit register renaming:
R10000 Freelist Management

P32	P36	P4	F6	F8	P34	P12	P14	P16	P18	P20	P22	P24	p26	P28	P30

Current Map Table

			Done?
F2	P2	DIVD P36,P34,P6	N
F10	P10	ADDD P34,P4,P32	y
F0	P0	LD P32,10(R2)	y

Newest

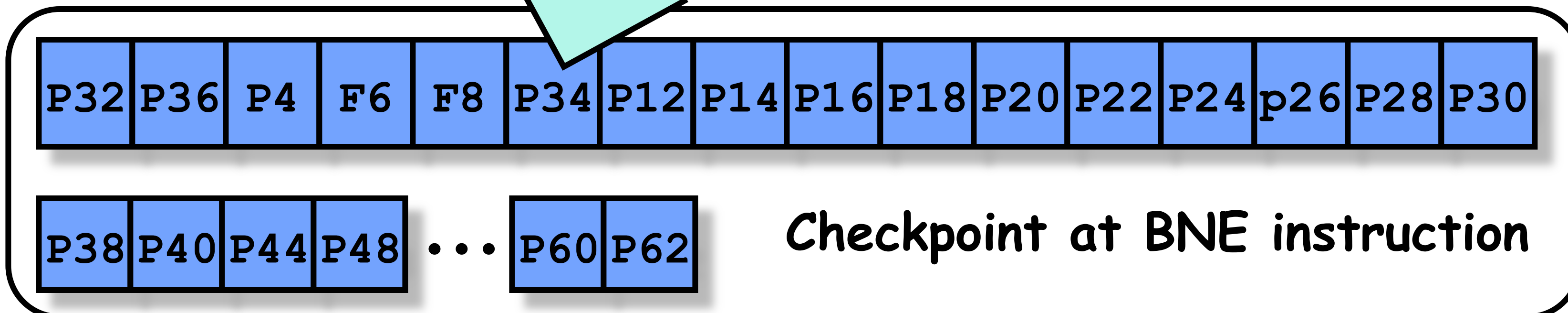


Oldest

P38	P40	P4			P0	P10

Freelist

Error fixed by restoring map table and *merging* freelist

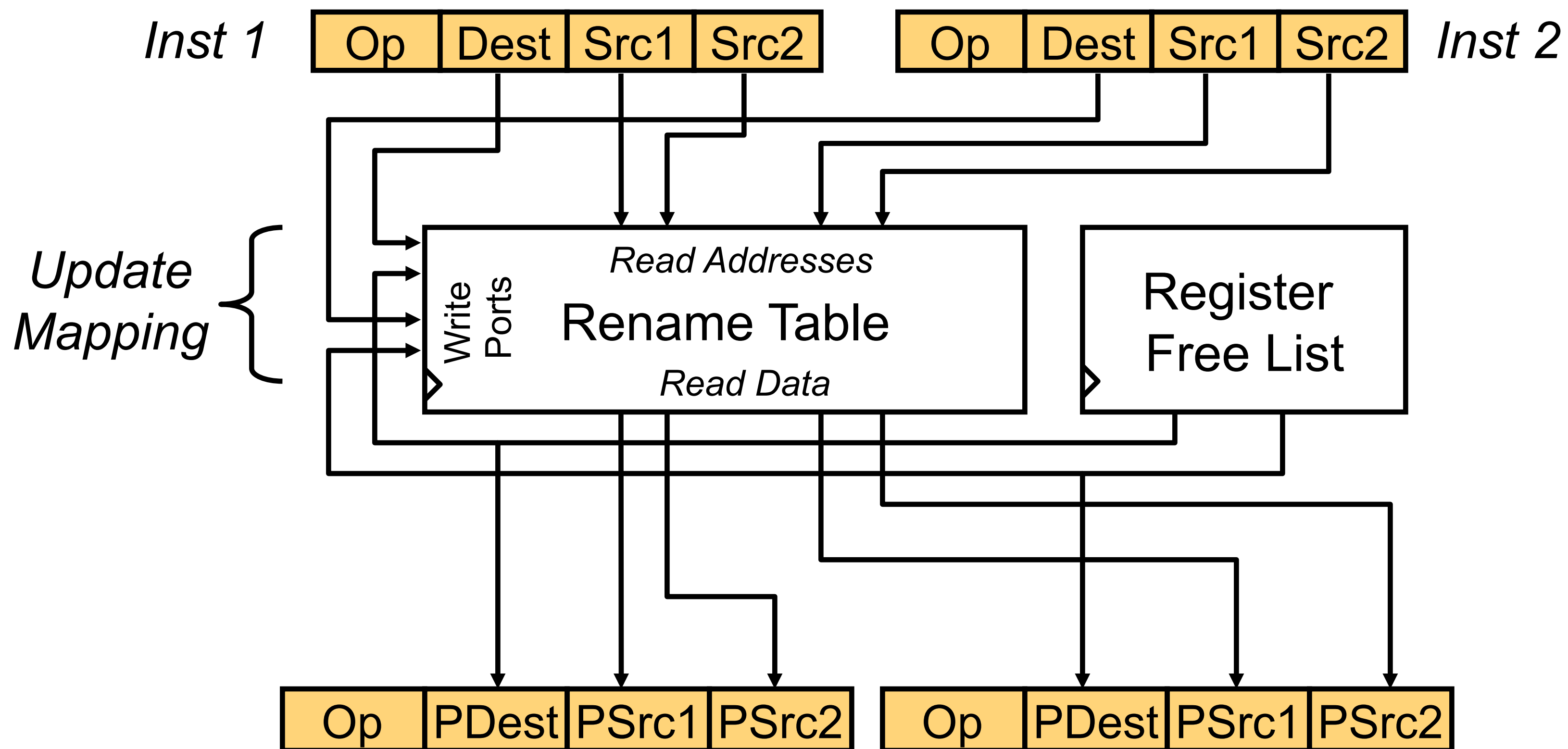


Advantages of Explicit Renaming

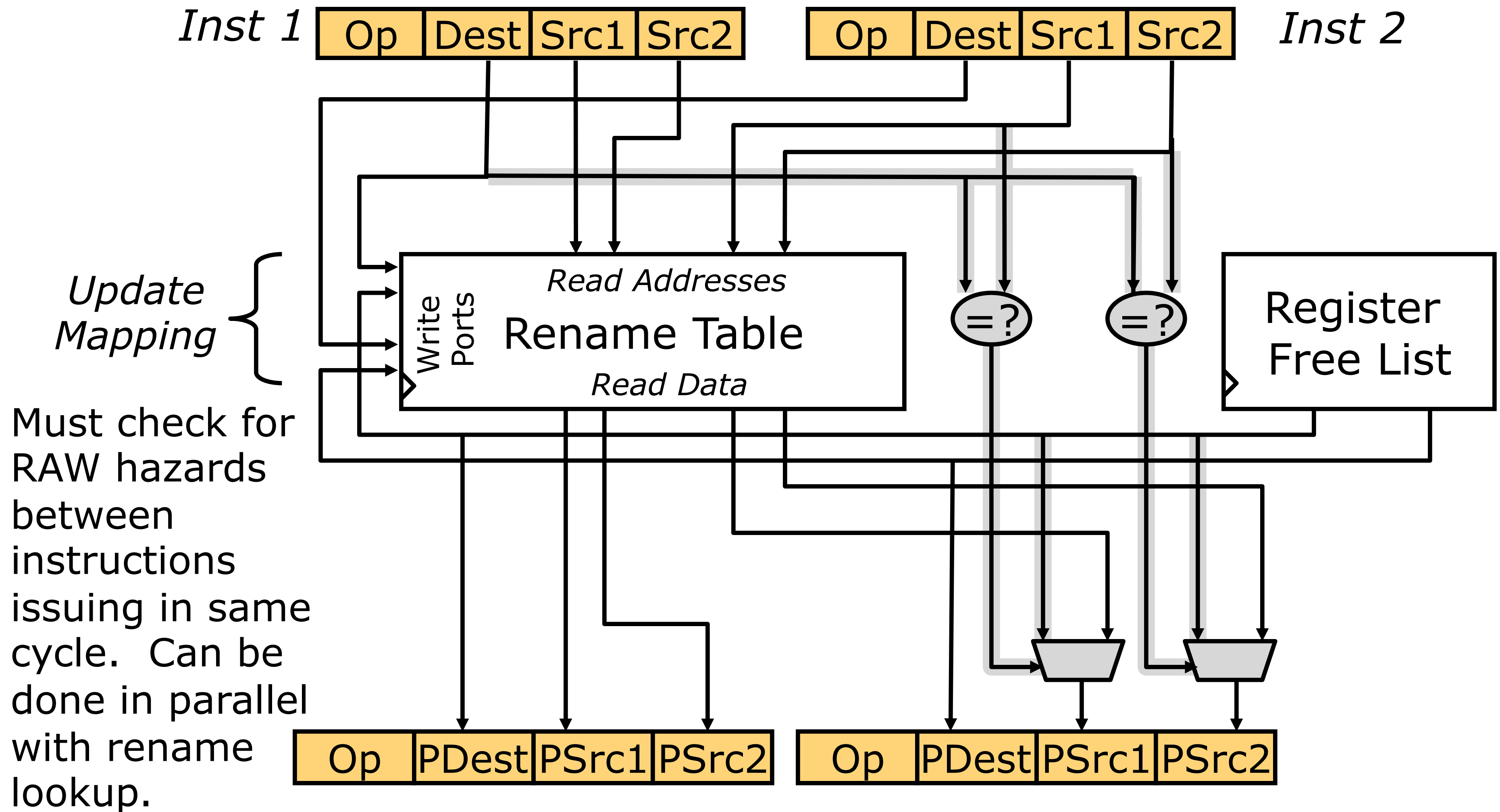
- Decouples *renaming* from *scheduling:*
  - Pipeline can be exactly like "standard" DLX pipeline (perhaps with multiple operations issued per cycle)
  - Or, pipeline could be tomasulo-like or a scoreboard, etc.
  - Standard forwarding or bypassing could be used
- Allows data to be fetched from single register file
  - No need to bypass values from reorder buffer
  - This can be important for balancing pipeline
- Many processors use a variant of this technique:
  - R10000, Alpha 21264, HP PA8000
- Another way to get precise interrupt points:
  - All that needs to be "undone" for precise break point is to undo the table mappings
  - Provides an interesting mix between reorder buffer and future file
    - » Results are written immediately back to register file
    - » Registers *names* are "freed" in program order (by ROB)

Superscalar Register Renaming

- During decode, instructions allocated new physical destination register
- Source operands renamed to physical register with newest value
- Execution unit only sees physical register numbers



Superscalar Register Renaming (Try #2)



Must check for RAW hazards between instructions issuing in same cycle. Can be done in parallel with rename lookup.

MIPS R10K renames 4 serially-RAW-dependent insts/cycle

Reality and References

- Modern processors uses the advanced technologies we talked about in this class and others not covered
  - Principles are the same mostly
- Historically and more depth
  - Lots of ideas have been evaluated and developed
  - Appendix L.5 for history and references
  - VLIW/EPIC and software pipelining: Appendix H
- More and Latest Info (Conference)
  - MICRO: Annual IEEE/ACM International Symposium on Microarchitecture
    - » https://www.microarch.org
  - IEEE Symposium on High Performance Computer Architecture (HPCA)
    - » http://hpca2017.org/
  - International Symposium on Computer Architecture (ISCA)
  - ACM International Conference on Architectural Support for Programming Languages and Operating Systems
    - » http://www.ece.cmu.edu/calcm/asplos2016
  - SIGARCH – The ACM Special Interest Group on Computer Architecture
    - » https://www.sigarch.org/

Put It All Together Examples: ARM Cortex-A8 and Intel Core i7

- **ARM Cortex-A8 core, the basis for the Apple A9 processor in the iPad, iPhones 3GS and 4**
  - **Dual-issue, statically scheduled superscalar with dynamic issue detection → 0.5 CPI ideally**
  - **The basic pipeline structure of the 13-stage pipeline.**
  - **set history C.**
  - **esses.**

e

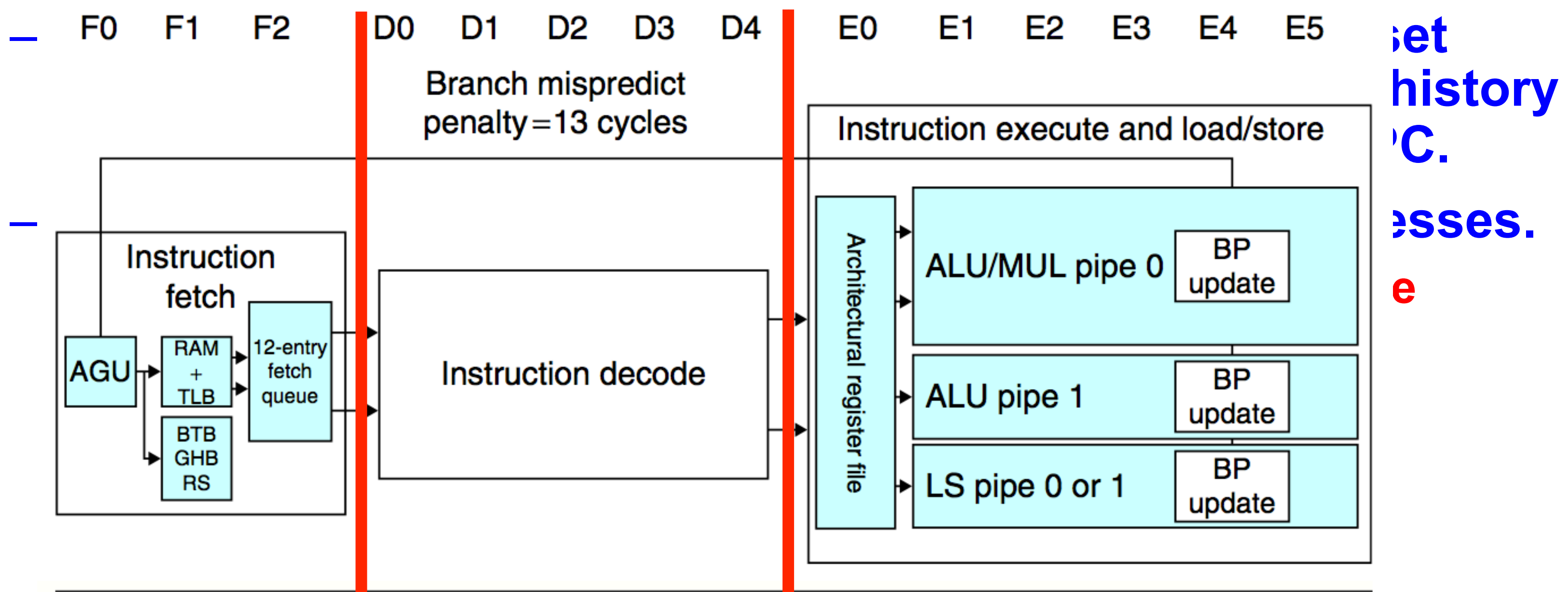
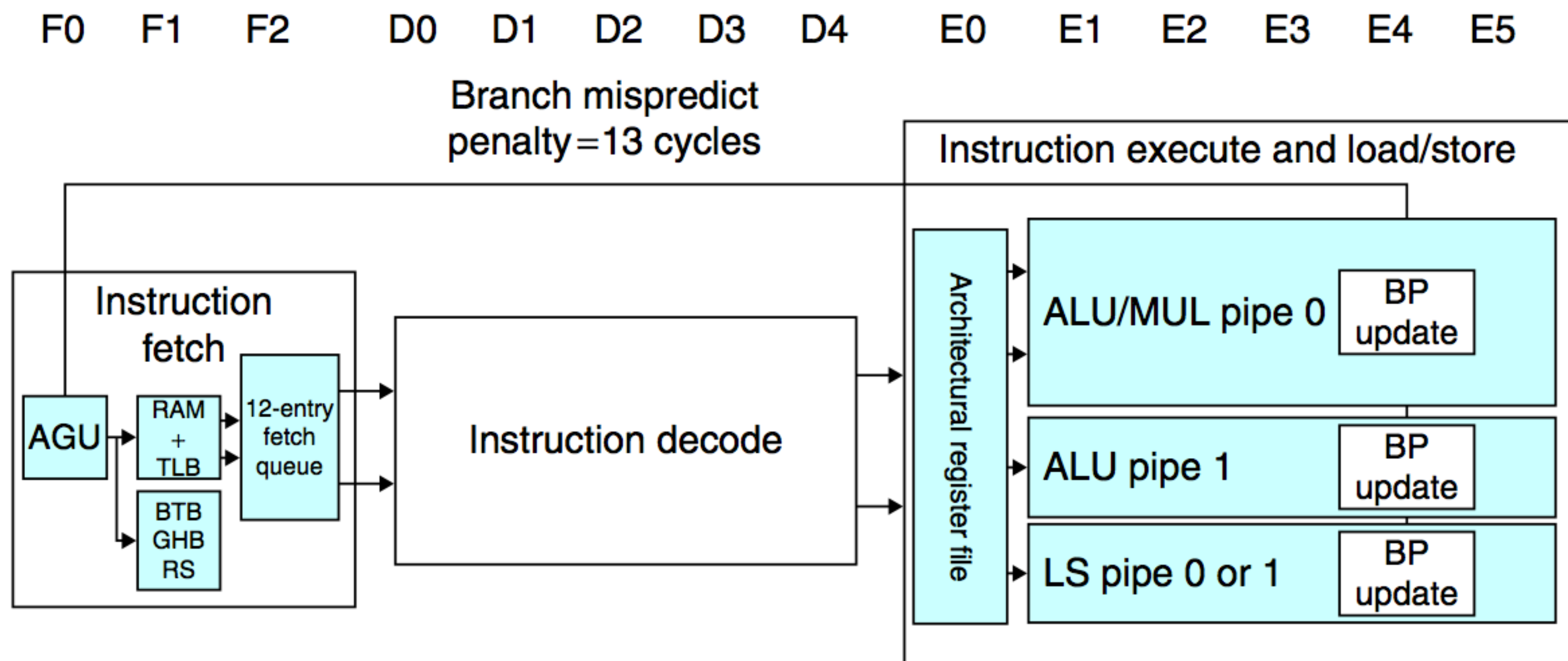


Figure 3.36 The basic structure of the A8 pipeline is 13 stages. Three cycles are used for instruction fetch and four for instruction decode, in addition to a five-cycle integer pipeline. This yields a 13-cycle branch misprediction penalty. The instruction fetch unit tries to keep the 12-entry instruction queue filled.

Put It All Together Examples: ARM Cortex-A8 and Intel Core i7

- **ARM Cortex-A8 core, the basis for the Apple A9 processor in the iPad, iPhones 3GS and 4**
  – **A dynamic branch predictor with a 512-entry two-way set associative branch target buffer and a 4K-entry global history buffer, indexed by the branch history and the current PC.**
  – **An eight-entry return stack is kept to track return addresses.**
    » **Misprediction results in a 13- cycle penalty of the pipeline flushed.**



Cortex-A8 Decode

- **5-stage decoding**

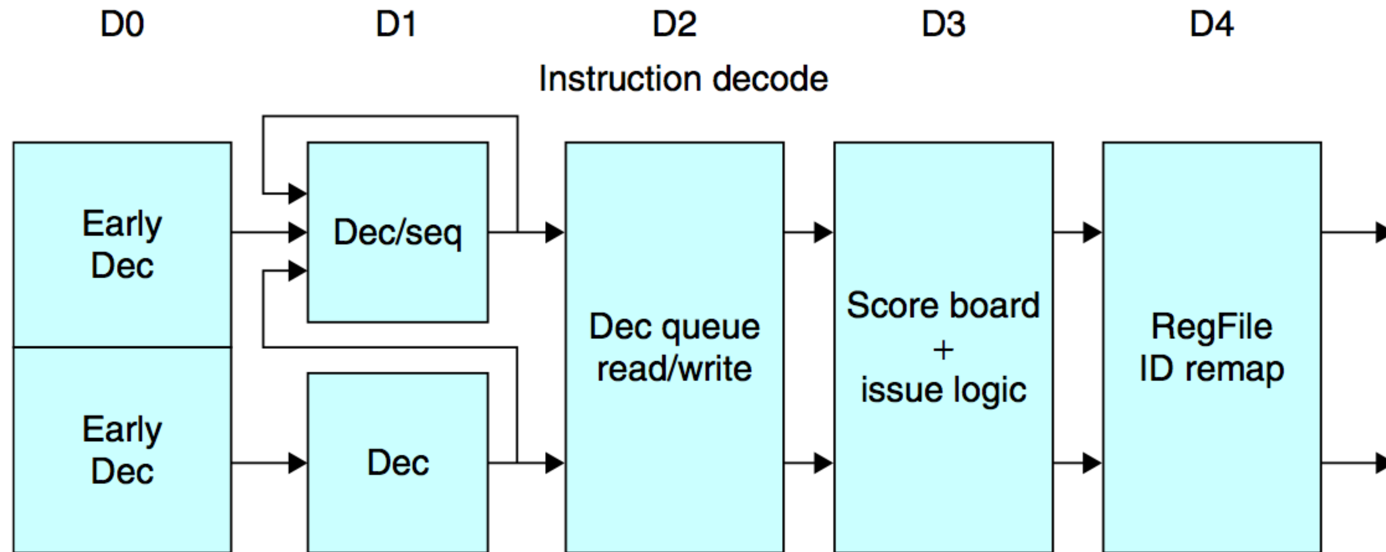
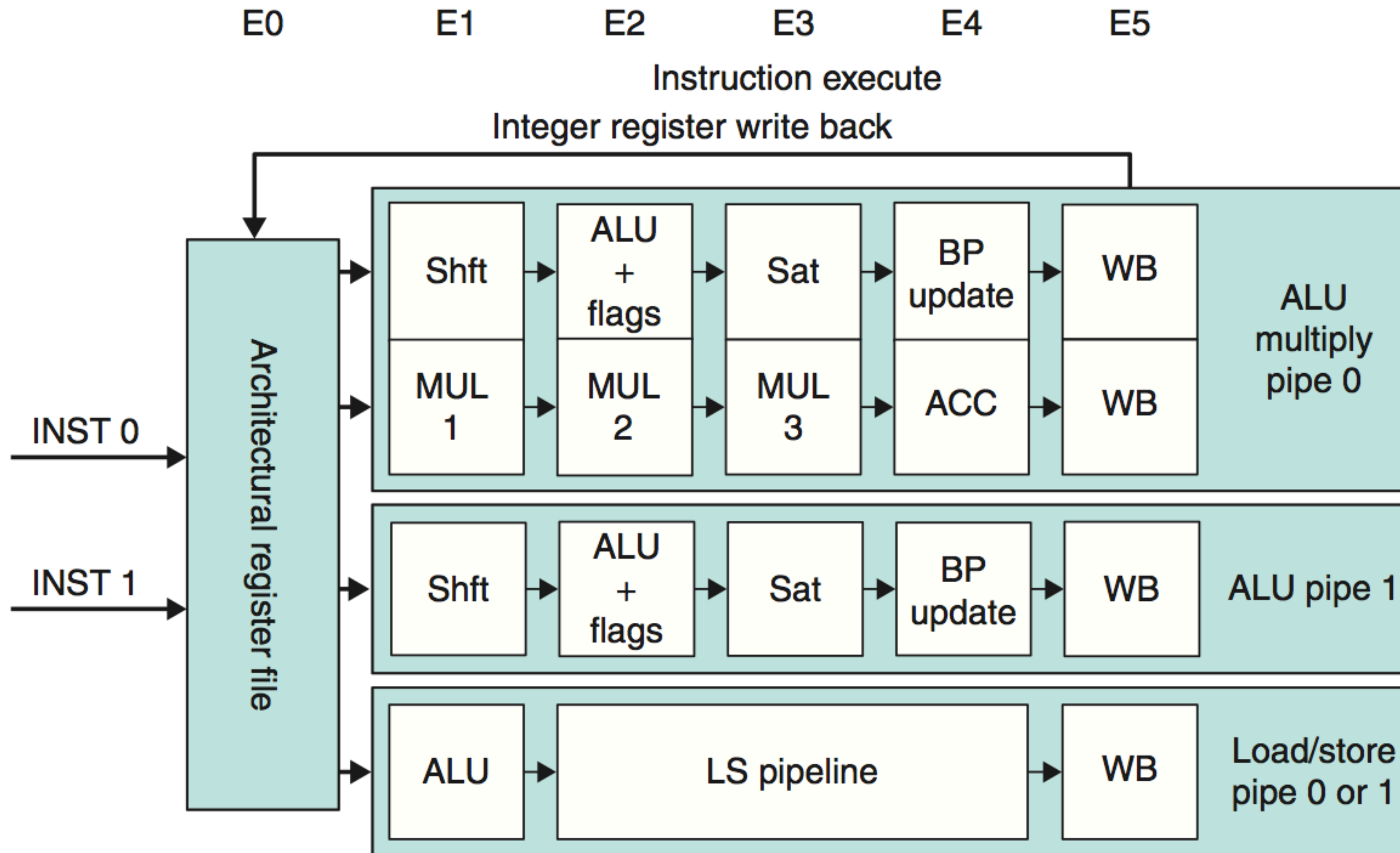


Figure 3.37 The five-stage instruction decode of the A8. In the first stage, a PC produced by the fetch unit (either from the branch target buffer or the PC incrementer) is used to retrieve an 8-byte block from the cache. Up to two instructions are decoded and placed into the decode queue; if neither instruction is a branch, the PC is incremented for the next fetch. Once in the decode queue, the scoreboard logic decides when the instructions can issue. In the issue, the register operands are read; recall that in a simple scoreboard, the operands always come from the registers. The register operands and opcode are sent to the instruction execution portion of the pipeline.

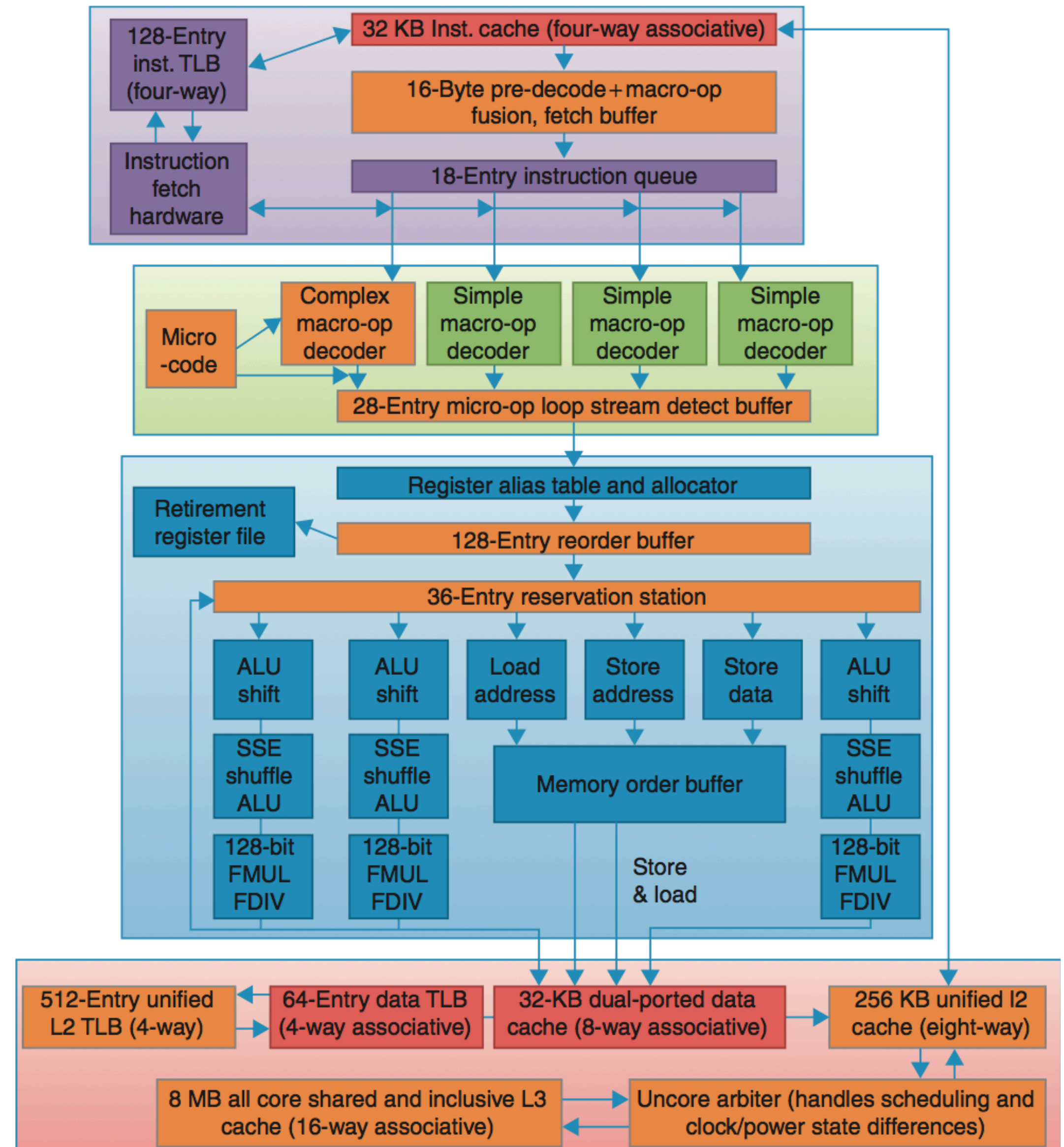
Cortex-A8 EXE Stage

- **5-stage execution (E5 is actually WB)**

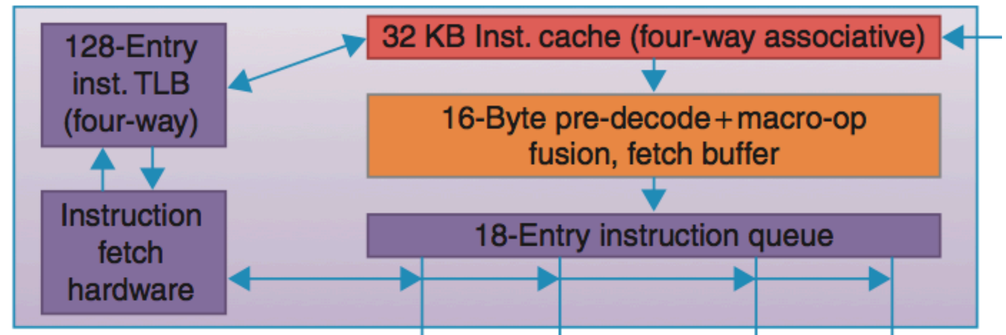


Intel Core i7

- Aggressive out-of-order speculative
- 14 stages pipeline,
- Branch mispredictions costing 17 cycles.
- 48 load and 32 store buffers.
- Six independent functional units
  - 6-wide superscalar

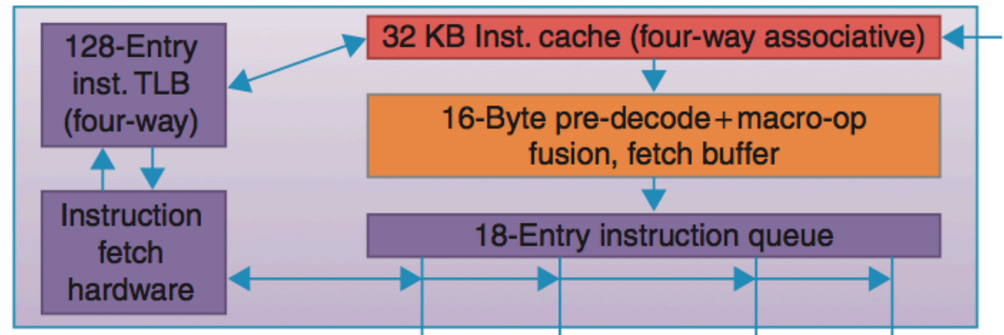


Core i7 Pipeline: IF



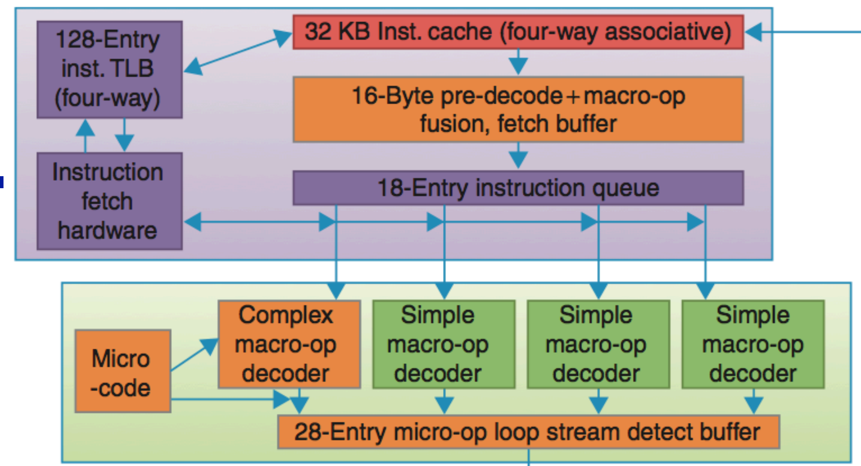
- Instruction fetch – Fetch 16 bytes from the I cache
  - A multilevel branch target buffer to achieve a balance between speed and prediction accuracy.
  - A return address stack to speed up function return.
  - Mispredictions cause a penalty of about 15 cycles.

Core i7 Pipeline: Predecode



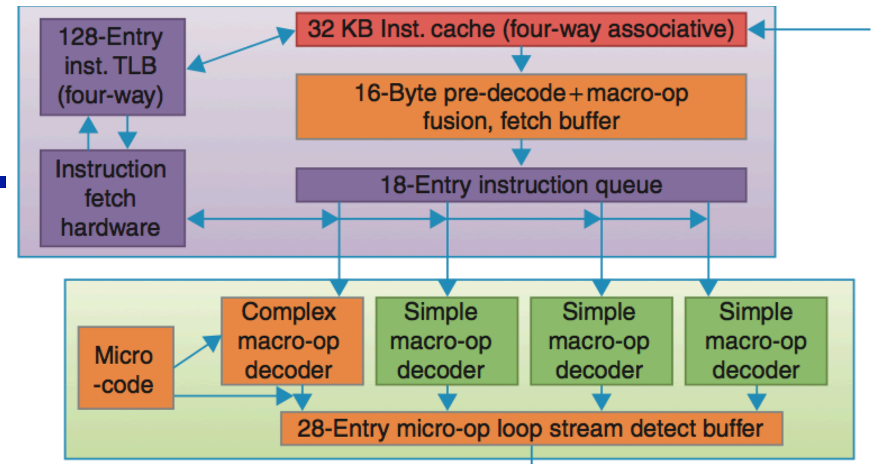
- Predecode –16 bytes instr in the predecode I buffer
  - *Macro-op fusion:* Fuse instr combinations such as compare followed by a branch into a single operation.
  - Instr break down: breaks the 16 bytes into individual x86 instructions.
    - » nontrivial since the length of an x86 instruction can be from 1 to 17 bytes and the predecoder must look through a number of bytes before it knows the instruction length.
  - Individual x86 instructions (including some fused instructions) are placed into the 18-entry instruction queue.

Core i7 Pipeline: Micro-op decode



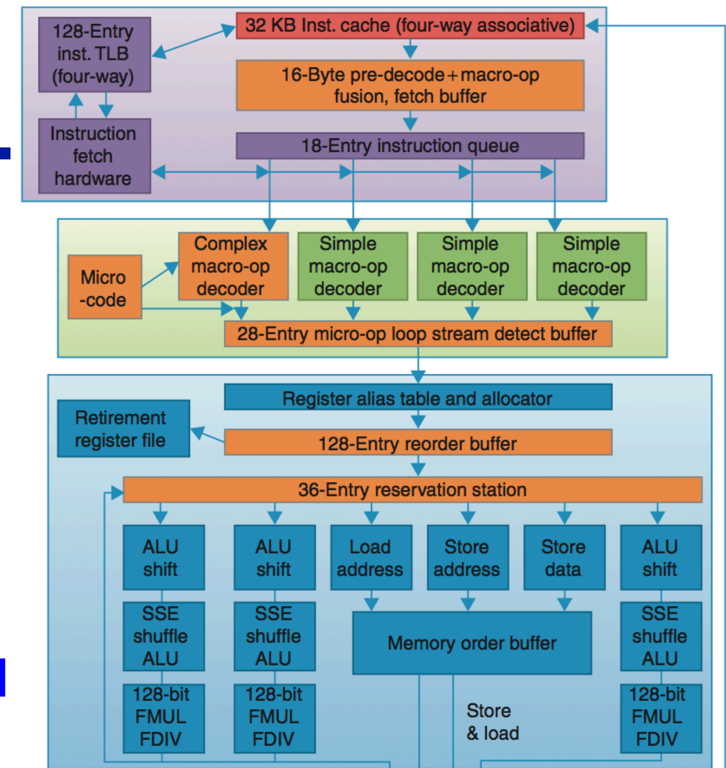
- **Micro-op decode – Translate Individual x86 instructions into micro-ops.**
  - **Micro-ops are simple MIPS-like instructions that can be executed directly by the pipeline (RISC style)**
    - » **introduced in the Pentium Pro in 1997 and has been used since.**
  - **Three simple micro-op decoders handle x86 instructions that translate directly into one micro-op.**
  - **One complex micro-op decoder produce the micro-op sequence of complex x86 instr;**
    - » **produce up to four micro-ops every cycle**
  - **The micro-ops are placed according to the order of the x86 instructions in the 28- entry micro-op buffer.**

Core i7 Pipeline: loop stream detection and microfusion



- *loop stream detection* and *microfusion by* the micro-op buffer preforms
  - If there is a sequence of instructions (less than 28 instrs or 256 bytes in length) that comprises a loop, the loop stream detector will find the loop and directly issue the micro-ops from the buffer
    - » eliminating the need for the instruction fetch and instruction decode stages to be activated.
  - Microfusion combines instr pairs such as load/ALU operation and ALU operation/store and issues them to a single reservation station, thus increasing the usage of the buffer.
    - » Study comparing the microfusion and macrofusion by Bird et al. [2007] discovered that microfusion had little impact on performance, while macrofusion appears to have a modest positive impact on integer performance and little impact on FP.

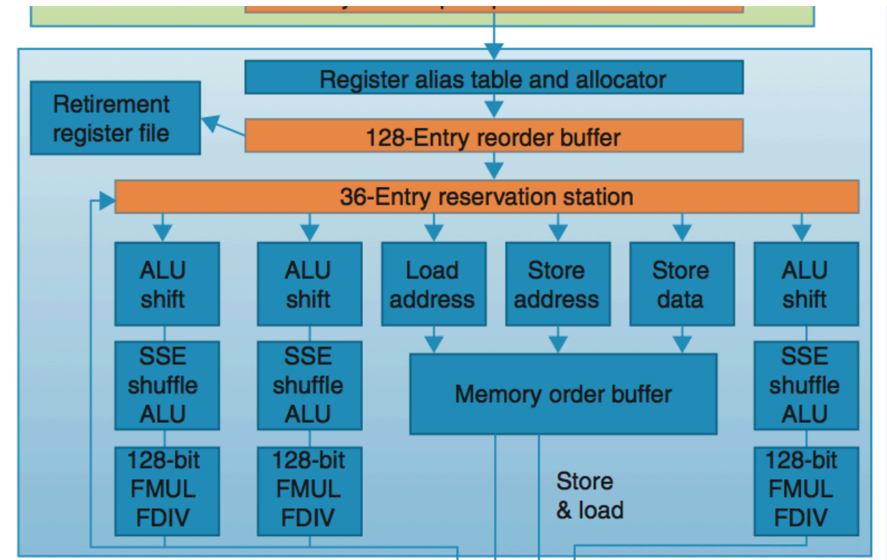
Core i7 Pipeline: Issue



- Basic instruction issue
  - Looking up the register location in the register tables
  - renaming the registers
  - allocating a reorder buffer entry
  - fetching any results from the registers or reorder buffer before send reservation stations.
- 36-entry centralized reservation station shared by six functional units

  Up to six micro-ops may be dispatched to the functional units every clock cycle.

Core i7 Pipeline: EXE and Retirement



- **Micro-ops are executed by the individual function units**
  - **results are sent back to any waiting reservation station as well as to the register retirement unit, where they will update the register state. The entry corresponding to the instruction in the reorder buffer is marked as complete.**
- **Retirement**
  - **When one or more instructions at the head of the reorder buffer have been marked as complete, the pending writes in the register retirement unit are executed, and the instructions are removed from the reorder buffer.**

Core i7 Performance

- The integer CPI values range from 0.44 to 2.66 with a standard deviation of 0.77
- The FP CPU is from 0.62 to 1.38 with a standard deviation of 0.25.
- Cache behavior is major contribution to the stall CPI