Multiple Instruction Issue

Multiple instructions issued each cycle

- a processor that can execute more than one instruction per cycle
- **issue width** = the number of **issue slots**, 1 slot/instruction
- not all types of instructions can be issued together
  - an example: 2 ALUs, 1 load/store unit, 1 FPU

    1 ALU does shifts & integer multiplies; the other executes branches

Motivation:

⇒ better performance

- increase instruction throughput
- decrease in CPI (below 1)

Cost:

⇒ greater hardware complexity, potentially longer wire lengths

⇒ harder code scheduling job for the compiler

Multiple Instruction Issue on Superscalars

Require:

- instruction fetch
  - fetching of multiple instructions at once
  - dynamic branch prediction & fetching speculatively beyond conditional branches
- instruction issue
  - methods for determining which instructions can be issued next
  - the ability to issue multiple instructions in parallel
- instruction commit
  - methods for committing several instructions in fetch order
- duplicate & more complex hardware

Multiple Instruction Issue

Scheduling instructions: which instructions are sent to the functional units for execution & when

Superscalar processors

- instructions are scheduled for execution ***by the hardware***
- ***different*** numbers of instructions may be issued simultaneously

VLIW (“very long instruction word”) **processors**

- instructions are scheduled for execution ***by the compiler***
- a ***fixed*** number of operations are formatted as one big instruction
- usually **LIW** (3 operations) today

In-order vs. Out-of-order Execution

In-order instruction execution

- instructions are fetched, executed & committed in compiler-generated order
    - if one instruction stalls, all instructions behind it stall
- instructions are **statically scheduled** by the hardware
    - scheduled in compiler-generated order
    - how many of the next *n* instructions can be issued, where *n* is the superscalar issue width
        - superscalars can have structural & data hazards within the *n* instructions
- advantage of in-order instruction scheduling: simpler implementation
    - ➤ faster clock cycle
    - ➤ fewer transistors
    - ➤ faster design/development/debug time

In-order vs. Out-of-order Execution

Out-of-order instruction execution

- instructions are fetched in compiler-generated order
- instruction completion may be in-order (today) or out-of-order (older computers)
- in between they may be executed in some other order
- instructions are **dynamically scheduled** by the hardware
  - hardware decides in what order instructions can be executed
  - instructions behind a stalled instruction can pass it if not dependent upon it
- advantages: higher performance
  - better at hiding latencies, less processor stalling
  - higher utilization of functional units

In-order instruction issue: Alpha 21164

2 styles of static instruction scheduling

- dispatch buffer & instruction slotting (Alpha 21164)
- shift register model (UltraSPARC-1)

In-order instruction issue: Alpha 21164

Instruction slotting

- can issue up to 4 instructions
  - completely empty the instruction buffer before filling it again
  - compiler can pad with `nops` so a conflicting instruction is issued with the following instructions, not alone
- no data dependences in same issue cycle (some exceptions)
  - hardware to:
    - detect data hazards
    - control bypass logic

21164 Instruction Unit Pipeline

Fetch & issue

S0: instruction fetch
branch prediction bits read

S1: opcode decode
target address calculation
if predict taken, redirect the fetch

S2: **instruction slotting**: decide which of the next 4 instructions can be issued

- intra-cycle structural hazard check
- intra-cycle data hazard check

S3: **instruction dispatch**

- inter-cycle load-use hazard check
- register read

21164 Integer Pipeline

Execute (2 integer pipelines)

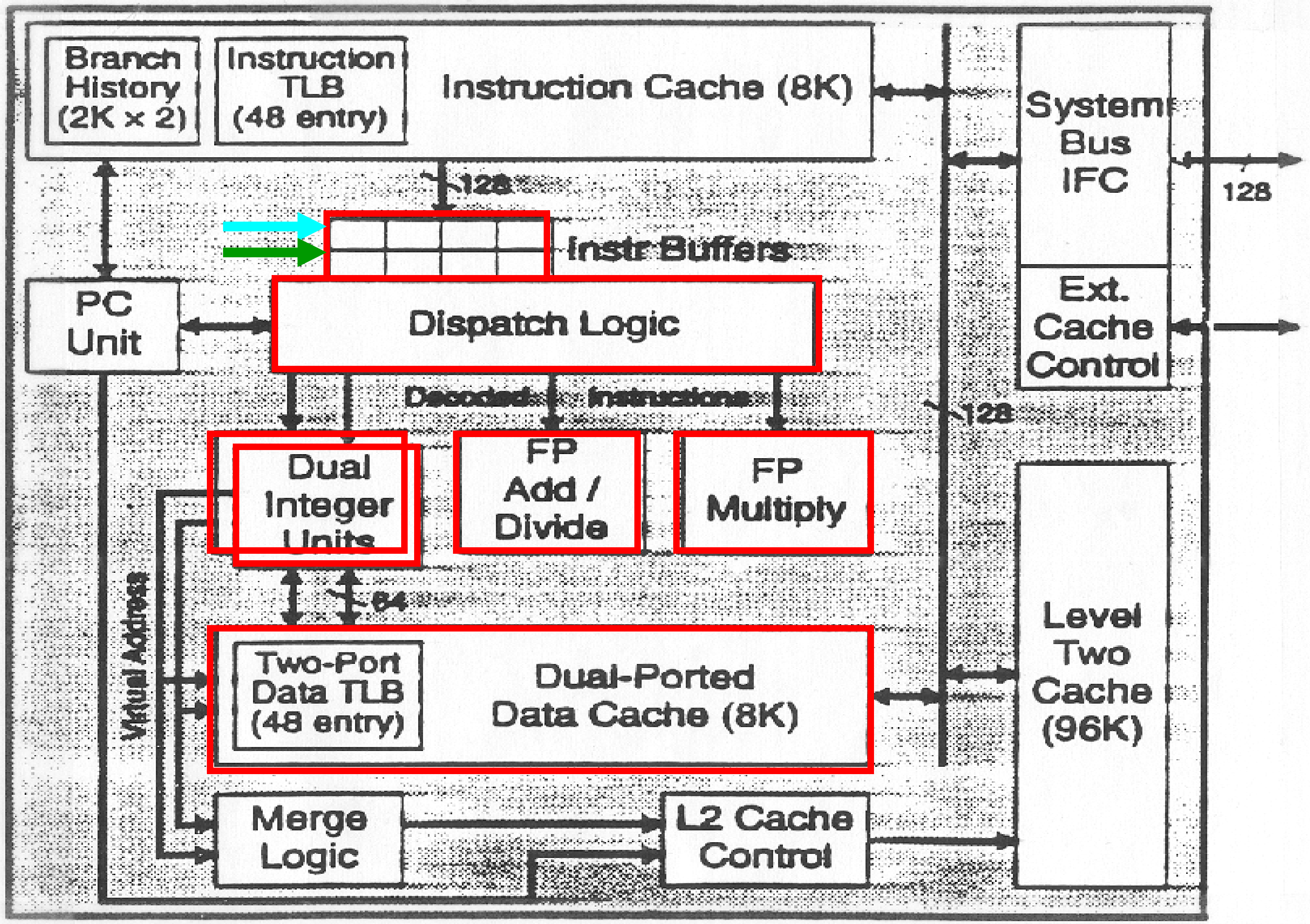
S4: integer execution
effective address calculation

S5: conditional move & branch execution
data cache access

S6: register write

also a 9-stage FP pipeline

ALPHA 21164

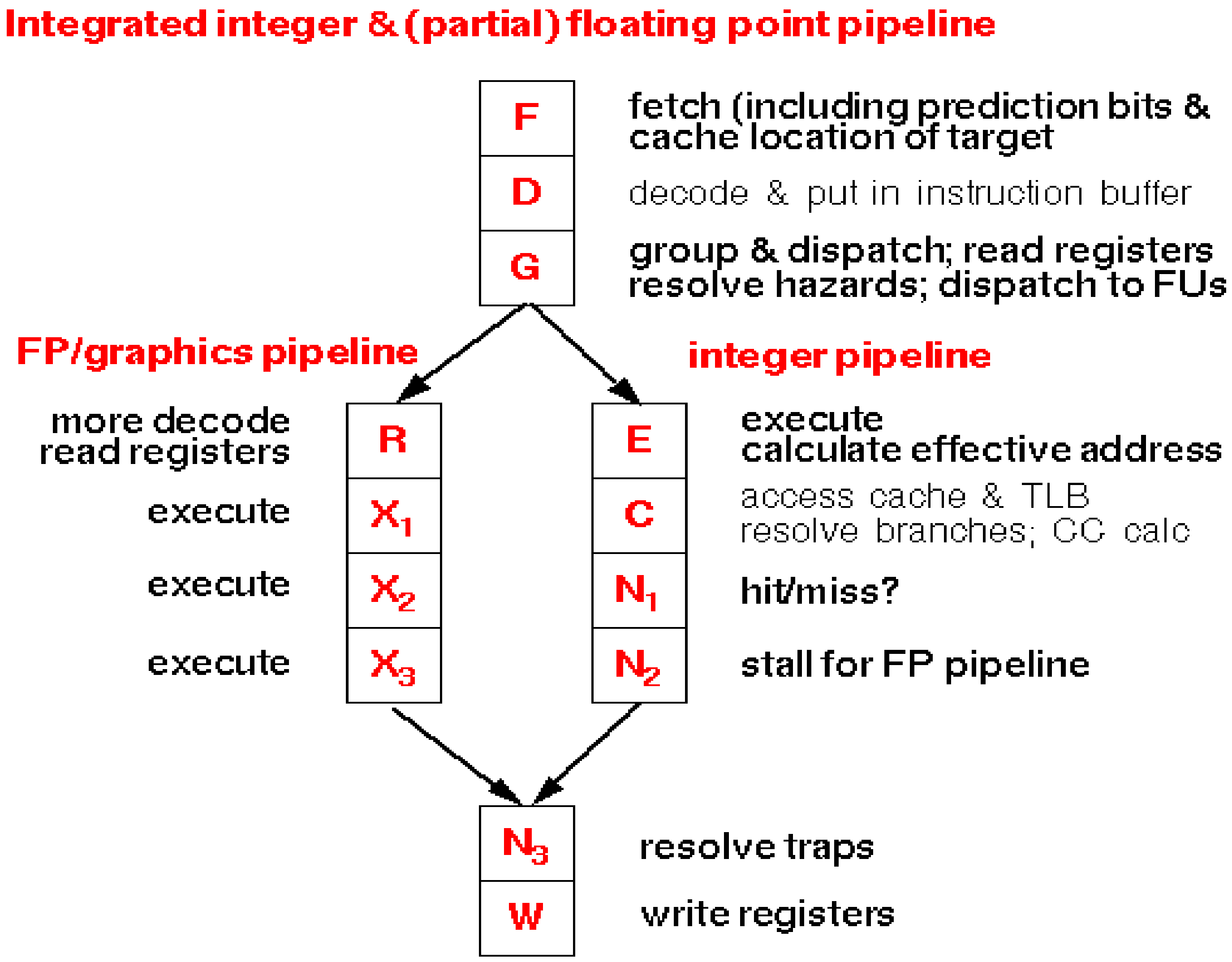

Branch History (2K x 2)
Instruction TLB (48 entry)
Instruction Cache (8K)
128
Instr Buffers
System Bus IFC
128
PC Unit
Dispatch Logic
Ext. Cache Control
Decoded Instructions
128
Dual Integer Units
FP Add / Divide
FP Multiply
64
Virtual Address
Two-Port Data TLB (48 entry)
Dual-Ported Data Cache (8K)
Level Two Cache (96K)
Merge Logic
L2 Cache Control

In-order instruction issue: UltraSparc 1

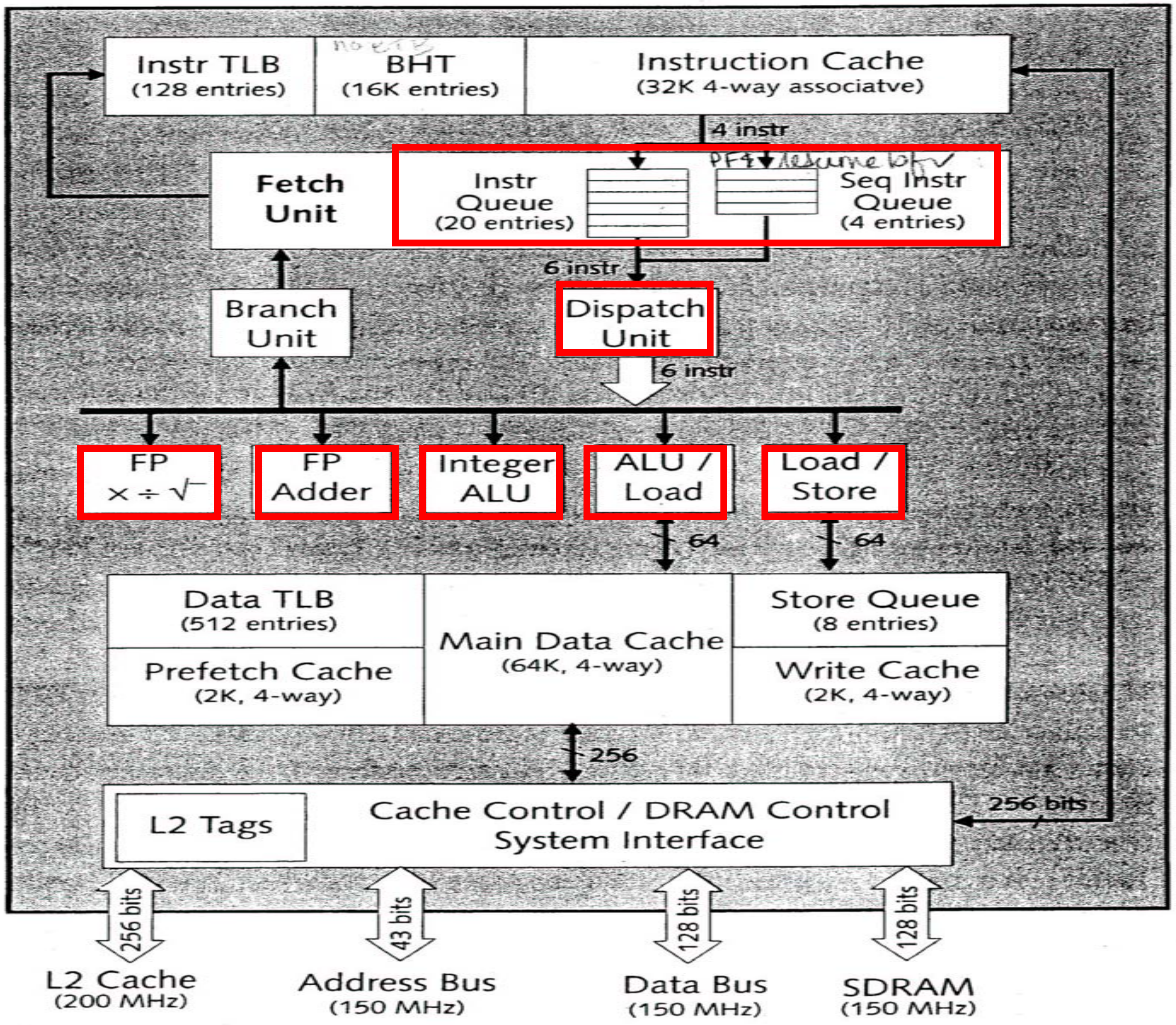
Shift register model

- can issue up to 4 instructions per cycle
- shift in new instructions after every group of instructions is issued
- some data dependent instructions can issue in same cycle

UltraSPARC 1


Integrated integer & (partial) floating point pipeline
F
fetch (including prediction bits & cache location of target
D
decode & put in instruction buffer
G
group & dispatch; read registers resolve hazards; dispatch to FUs
FP/graphics pipeline
integer pipeline
more decode read registers
R
execute
X_1
execute
X_2
execute
X_3
E
execute calculate effective address
C
access cache & TLB resolve branches; CC calc
N_1
hit/miss?
N_2
stall for FP pipeline
N_3
resolve traps
W
write registers

ultraSPArc - 3



Superscalars

Performance impact:

- increase performance because execute multiple instructions in parallel, not just overlapped
- CPI potentially < 1 (.5 if 2-wide)
- IPC (instructions/cycle) potentially > 1 (2 if 2-wide)
- better functional unit utilization

but

- need to fetch more instructions – how many?
- need independent instructions – why?
- need a good local mix of instructions – why?
- need more instructions to hide load delays – why?
- need to make better branch predictions – why?

Code Scheduling on Superscalars

Original code

```
Loop:   lw R1, 0(R5)
        addu R1, R1, R6
        sw R1, 0(R5)
        addi R5, R5, -4
        bne R5, R0, Loop
```

Code Scheduling on Superscalars

Original code

```
Loop:   lw R1, 0(R5)
        addu R1, R1, R6
        sw R1, 0(R5)
        addi R5, R5, -4
        bne R5, R0, Loop
```

With latency-hiding code scheduling

```
Loop:   lw R1, 0(s1)
        addi R5, R5, -4
        addu R1, R1, R6
        sw R1, 4(R5)
        bne R5, $0, Loop
```

	ALU/branch instructions	memory instructions	clock cycle
Loop:			1
			2
			3
			4

Code Scheduling on Superscalars: Loop Unrolling

	ALU/branch instruction	Data transfer instruction	clock cycle
`Loop:`	`addi R5, R5, -16`	`lw R1, 0(R5)`	1
		`lw R2, 12(R5)`	2
	`addu R1, R1, R6`	`lw R3, 8(R5)`	3
	`addu R2, R2, R6`	`lw R4, 4(R5)`	4
	`addu R3, R3, R6`	`sw R1, 16(R5)`	5
	`addu R4, R4, R6`	`sw R2, 12(R5)`	6
		`sw R3, 8(R5)`	7
	`bne R5, R0, Loop`	`sw R4, 4(R5)`	8

What is the cycles per iteration?

What is the IPC?

Loop unrolling provides:

+ fewer instructions that cause hazards (I.e., branches)
+ more independent instructions (from different iterations) & therefore increased instruction throughput
- increases **register pressure**
- must change **offsets**

Superscalars

Hardware impact:

- more & pipelined functional units
- multi-ported registers for multiple register access
- more buses from the register file to the additional functional units
- multiple decoders
- more hazard detection logic
- more bypass logic
- wider instruction fetch
- multi-banked L1 data cache

or else the processor has structural hazards (due to an unbalanced design) and stalling

There are restrictions on instruction types that can be issued together to reduce the amount of hardware.

Static (compiler) scheduling helps.