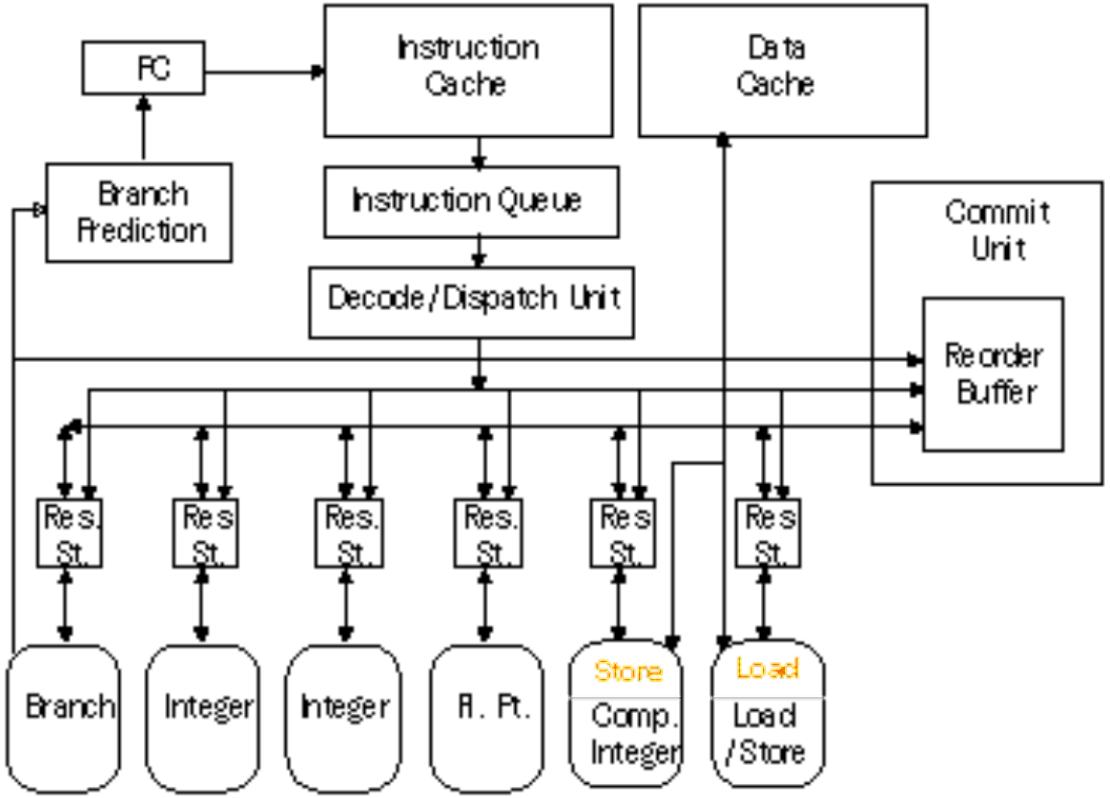
CS425
Computer Systems Architecture

Fall 2019

Multiple Issue:
Superscalar and VLIW

Example: Dynamic Scheduling in PowerPC 604 and Pentium Pro

- In-order Issue, Out-of-order execution, In-order Commit



Multiple Issue

$CPI = CPI_{IDEAL} + Stalls_{STRUC} + Stalls_{RAW} + Stalls_{WAR} + Stalls_{WAW} + Stalls_{CONTROL}$

- Have to maintain:
  - **Data Flow**
  - **Exception Behavior**

Dynamic instruction scheduling (HW)	Static instruction scheduling (SW/compiler)
Scoreboard (reduce RAW stalls)	Loop Unrolling
Register Renaming (reduce WAR & WAW stalls) •**Tomasulo** • **Reorder buffer**	SW pipelining
Branch Prediction (reduce control stalls)	Trace Scheduling
Multiple Issue (CPI < 1) **Multithreading** (CPI < 1)	

Beyond CPI = 1

- Initial goal to achieve CPI = 1
- Can we improve beyond this?
- <u>Superscalar:</u>
  - varying no. instructions/cycle (1 to 8), i.e. 1-way, 2-way, …, 8-way superscalar
  - scheduled by compiler (statically scheduled) or by HW (dynamically scheduled)
  - e.g. IBM PowerPC, Sun UltraSparc, DEC Alpha, HP 8000
  - the successful approach (to date) for general purpose computing
- Lead to use of Instructions Per Cycle (IPC) vs. CPI

Beyond CPI = 1

- Alternative approach:
- Very Long Instruction Words (VLIW):
  - fixed number of instructions (4-16)
  - scheduled by the compiler; put ops into wide templates
  - Currently found more success in DSP, Multimedia applications
  - Joint HP/Intel agreement in 1999/2000
  - Intel Architecture-64 (Merced/A-64) 64-bit address
  - Style: “Explicitly Parallel Instruction Computer (EPIC)”

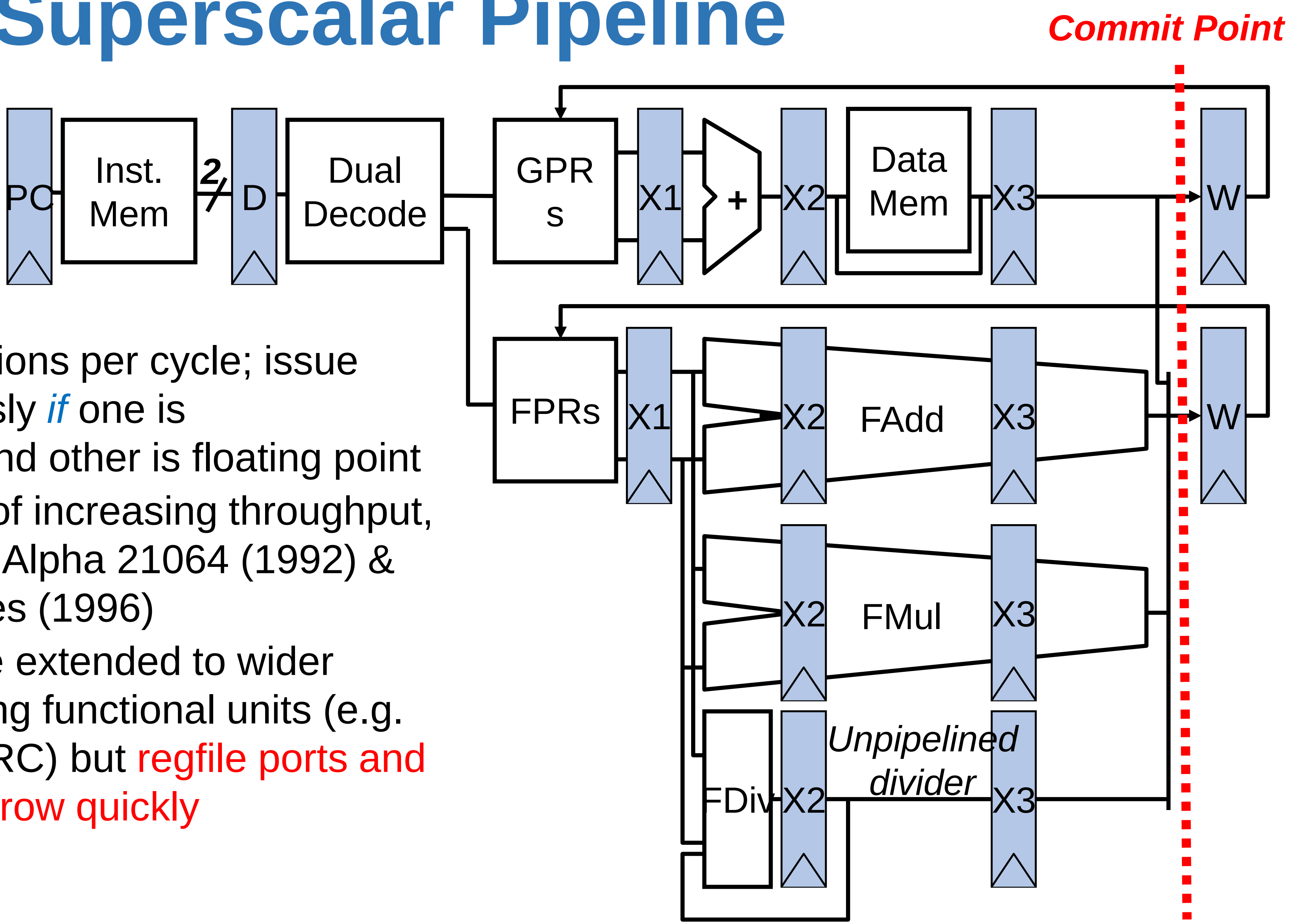
Getting CPI < 1: Issuing Multiple Instr/Cycle

- Superscalar DLX: 2 instructions, 1 FP & 1 anything else
  - Fetch 64-bits/clock cycle; Integer on left, FP on right
  - Can only issue 2^{nd} instruction if 1^{st} instruction issues
  - More ports for FP registers to do FP load & FP op in a pair

Type	*Pipe Stages*						
Int. instruction	IF	ID	EX	MEM	WB		
FP instruction	IF	ID	EX	MEM	WB		
Int. instruction		IF	ID	EX	MEM	WB	
FP instruction		IF	ID	EX	MEM	WB	
Int. instruction			IF	ID	EX	MEM	WB
FP instruction			IF	ID	EX	MEM	WB

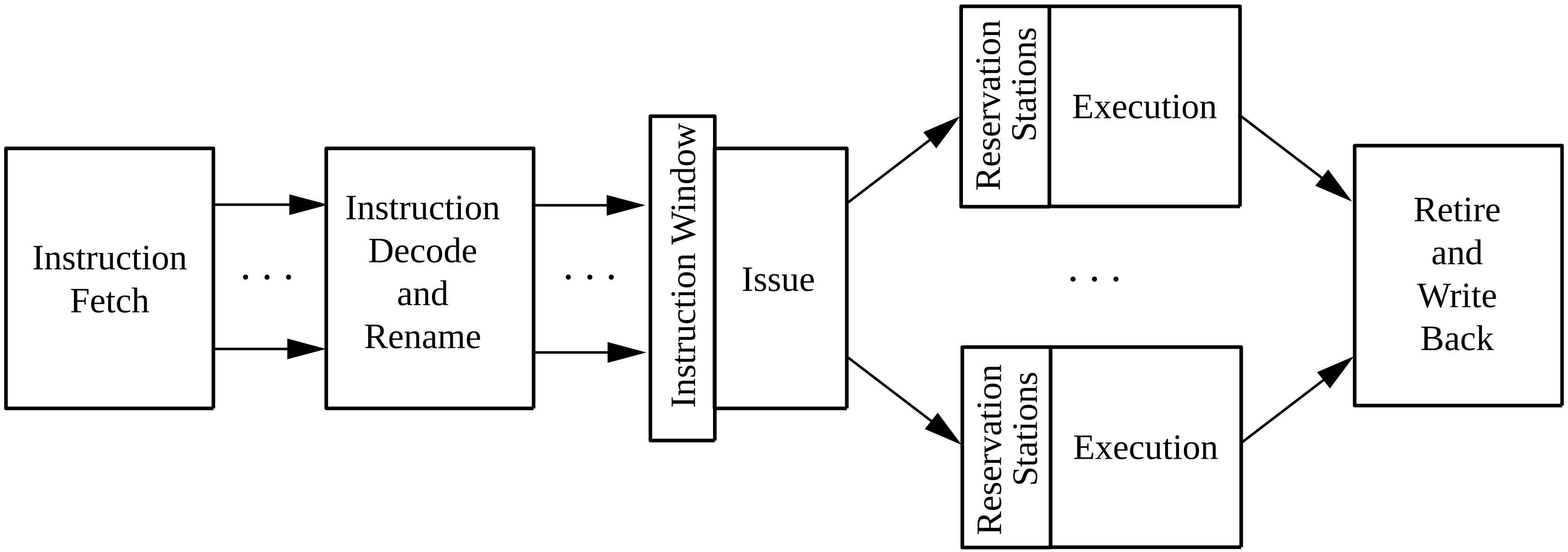
- 1 cycle load delay expands to 3 instructions in SS
  - instruction in right half can't use it, nor instructions in next slot

In-Order Superscalar Pipeline



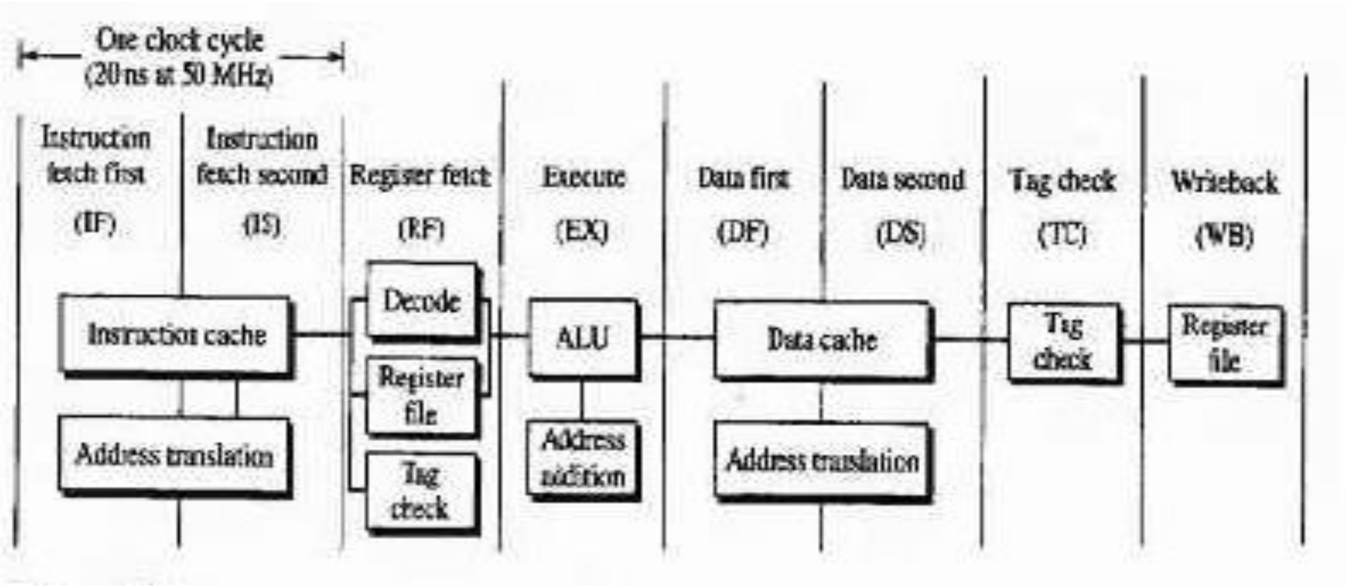
- Fetch two instructions per cycle; issue both simultaneously *if* one is integer/memory and other is floating point
- Inexpensive way of increasing throughput, examples include Alpha 21064 (1992) & MIPS R5000 series (1996)
- Same idea can be extended to wider issue by duplicating functional units (e.g. 4-issue UltraSPARC) but regfile ports and bypassing costs grow quickly

Superscalar Pipeline (PowerPC- and enhanced Tomasulo-Scheme)

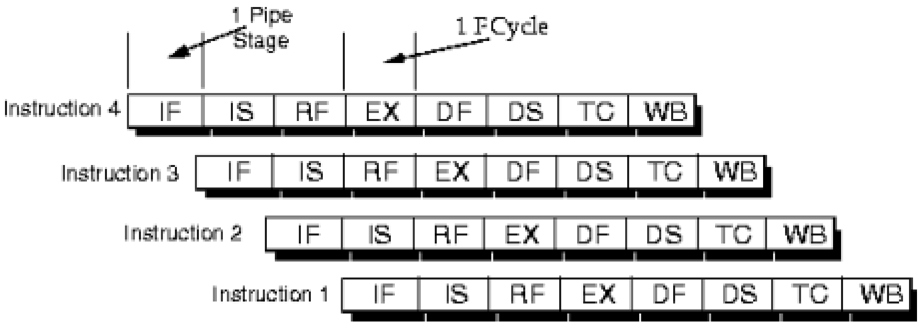


- Instructions in the instruction window are free from control dependencies due to branch prediction, and free from name dependences due to register renaming.
- Only (true) data dependences and structural conflicts remain to be solved.

Superpipelined Machines

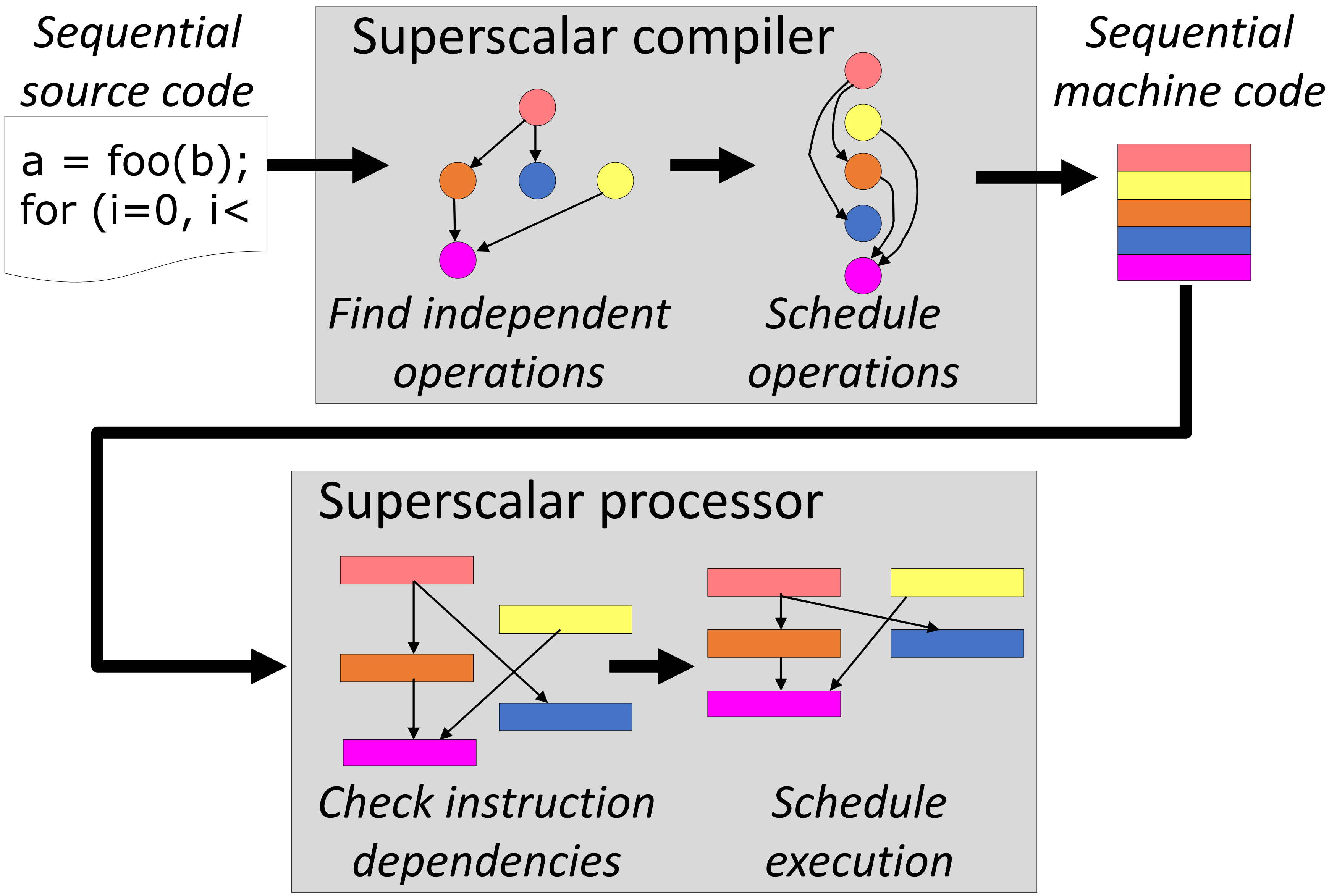


MIPS R4000



- Machine issues instructions faster than they are executed
- Advantage: increase in the number of instructions which can be in the pipeline at one time and hence the level of parallelism.
- Disadvantage: The larger number of instructions "in flight" (i.e. in some part of the pipeline) at any time, increases the potential for data and control dependencies to introduce stalls. Clock frequency is high.

Sequential ISA Bottleneck



Review: Unrolled Looping in Scalar

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

LD to ADDD: 1 Cycle
ADDD to SD: 2 Cycles

14 clock cycles, or 3.5 per iteration

Loop Unrolling in Superscalar

	Integer instruction	*FP instruction*	*Clock cycle*
Loop:	LD F0,0(R1)		1
	LD F6,-8(R1)		2
	LD F10,-16(R1)	ADDD F4,F0,F2	3
	LD F14,-24(R1)	ADDD F8,F6,F2	4
	LD F18,-32(R1)	ADDD F12,F10,F2	5
	SD 0(R1),F4	ADDD F16,F14,F2	6
	SD -8(R1),F8	ADDD F20,F18,F2	7
	SD -16(R1),F12		8
	SD -24(R1),F16		9
	SUBI R1,R1,#40		10
	BNEZ R1,LOOP		11
	SD -32(R1),F20		12

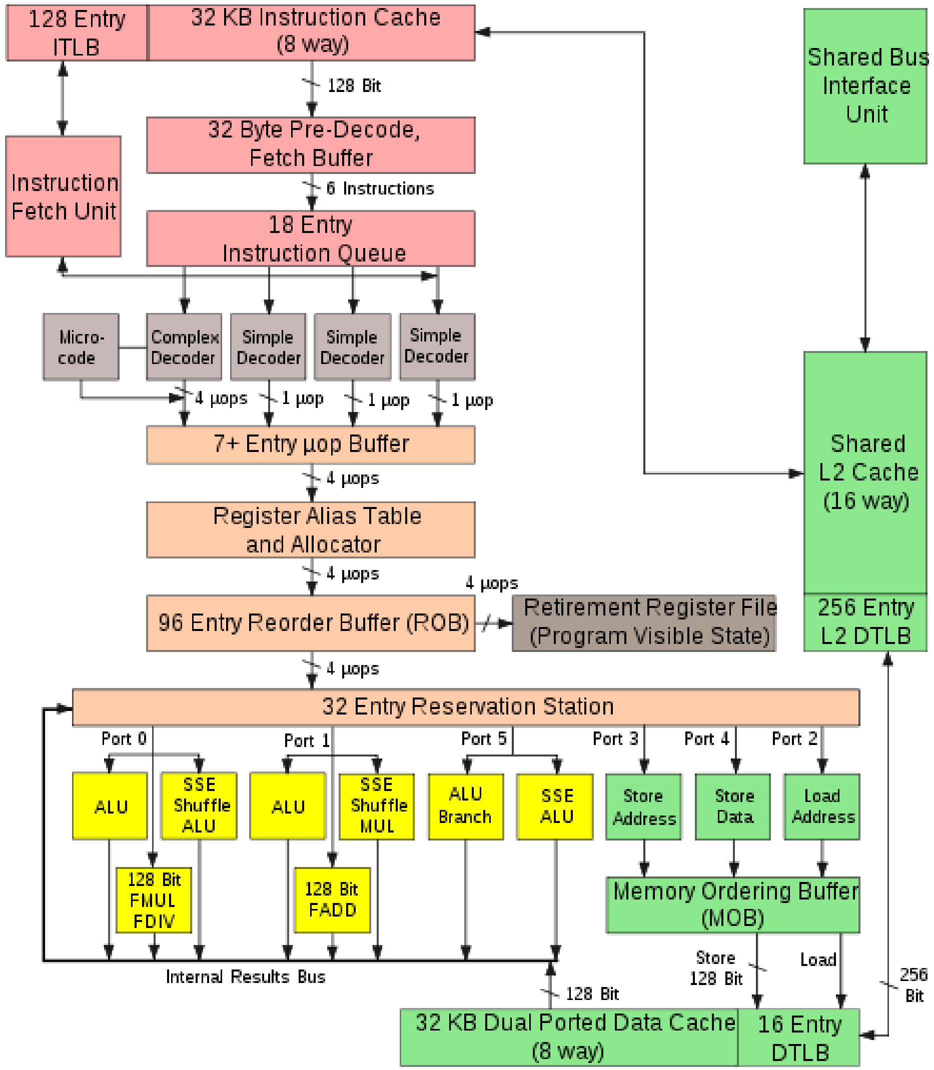
Unrolled 5 times to avoid delays

12 clocks, or 2.4 clocks per iteration (1.5X)

SS Advantages and Challenges

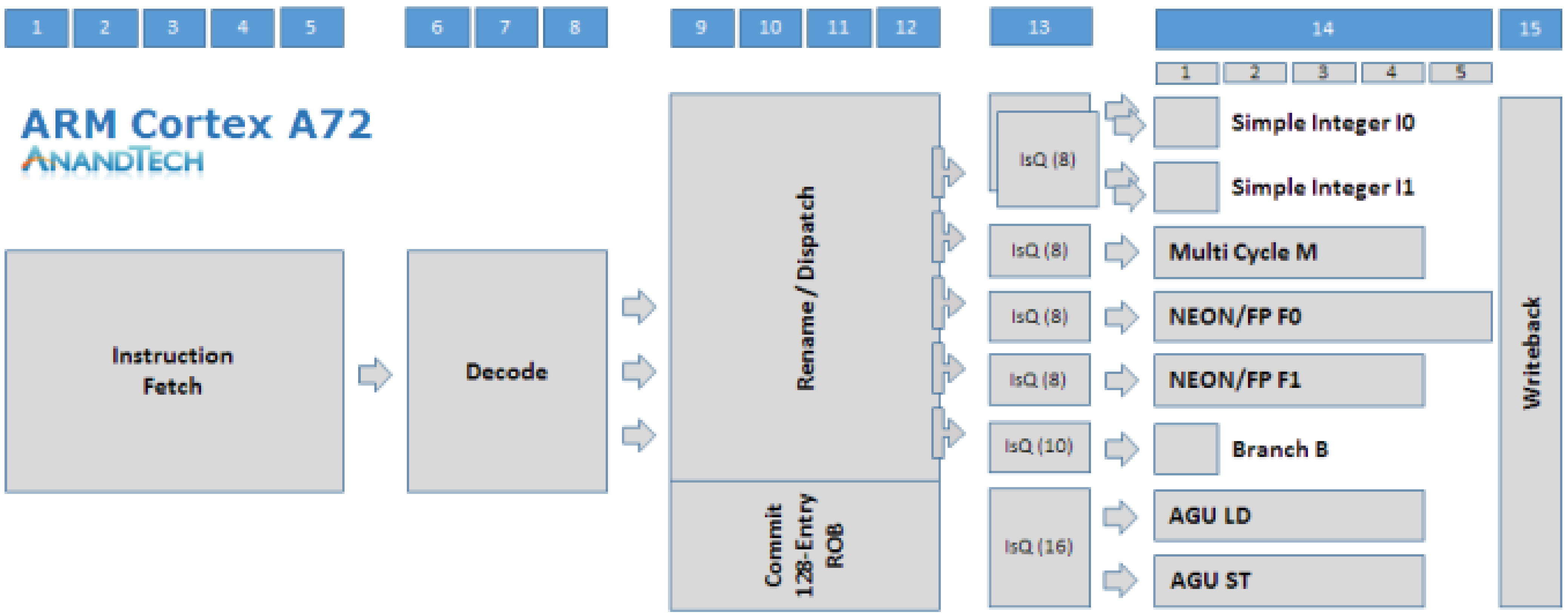
- The potential advantages of a SS processor versus a vector or VLIW processor are their ability to extract some parallelism from less structured code (i.e. not only loops) and their ability to easily cache all forms of data.
- While Integer/FP split is simple for the HW, get CPI of 0.5 only for programs with:
  - Exactly 50% FP operations
  - No hazards
- If more instructions issue at same time, greater difficulty of decode and issue
  - Even 2 way-scalar => examine 2 opcodes, 6 register specifiers, & decide if 1 or 2 instructions can issue

Example Desktop Processor: Intel Core 2

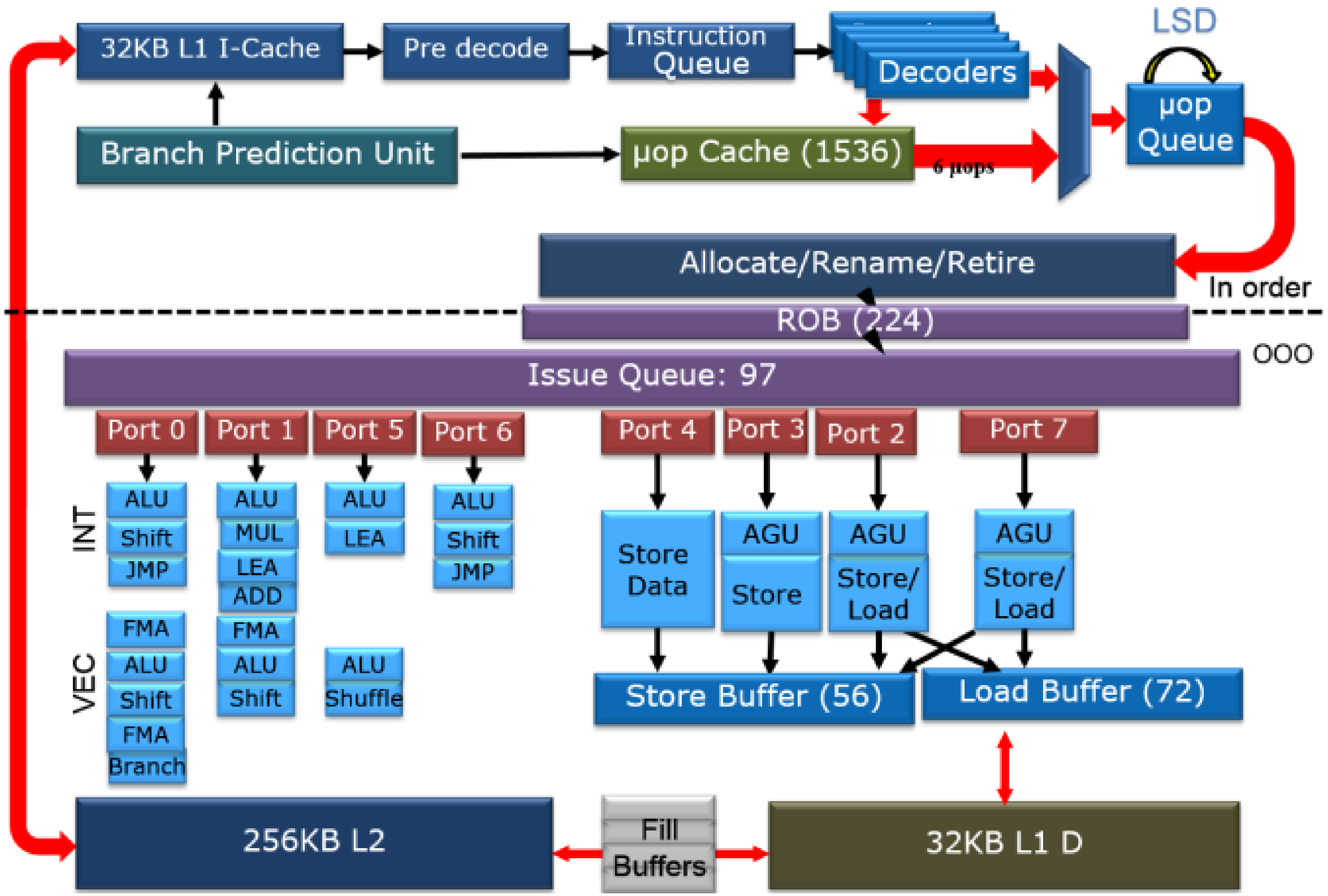


Superpipelined & Superscalar (4-way)

Example Mobile Processor: ARM A72



Example Server Processor: IBM POWER8



All in one: 2-way SS + OoO + Branch Prediction + Reorder Buffer (Speculation)

Show read cycle for clarification

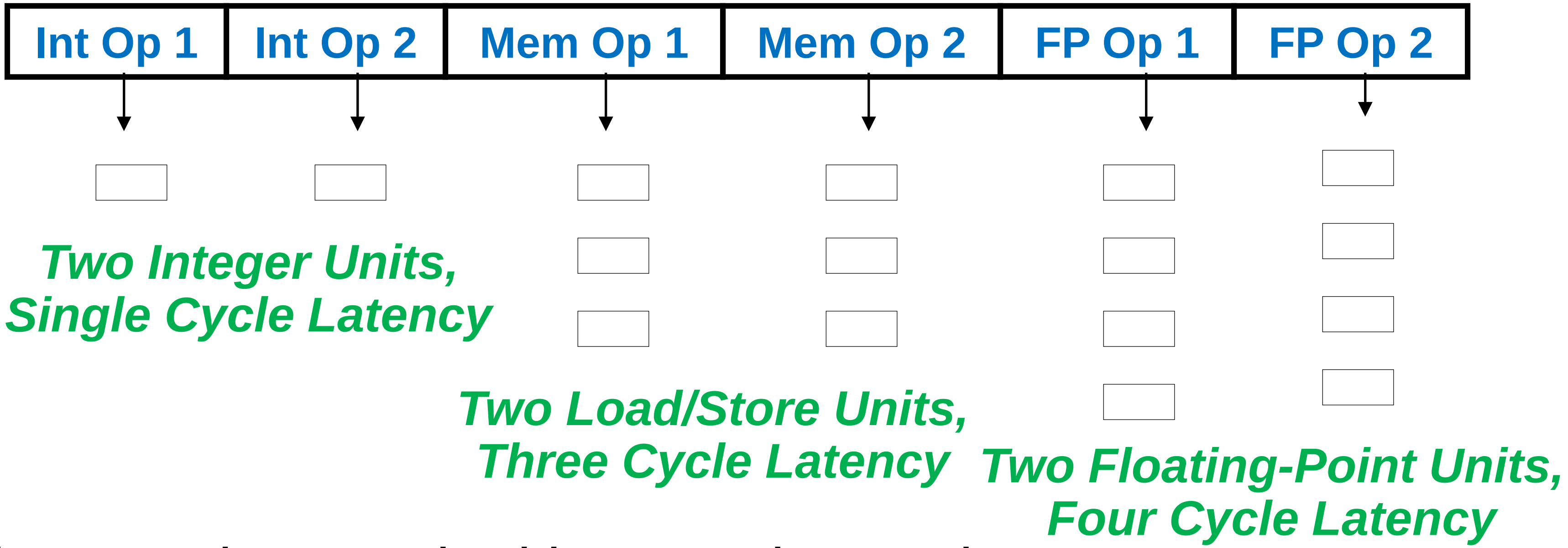
Iteration number	Instructions		Issues at clock number	Executes at clock number	Read access at clock number	Write CDB at clock number	Commits at clock number	Comment
1	LD	R2,0(R1)	1	2	3	4	5	First issue
1	DADDIU	R2,R2,#1	1	5		6	7	Wait for LW
1	SD	R2,0(R1)	2	3			7	Wait for DADDIU
1	DADDIU	R1,R1,#8	2	3		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

2 issues/cycle & 2 commits/cycle otherwise reorder buffer will overflow

Alternative Solutions

- Very Long Instruction Word (VLIW)
- Explicitly Parallel Instruction Computing (EPIC)
- Simultaneous Multithreading (SMT), next lecture
- Multi-core processors, ~last lecture
- VLIW: 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., 2 integer operations, 2 FP ops, 2 Memory refs, 1 branch
    - 16 to 24 bits per field → 7x16 or 112 bits to 7x24 or 168 bits wide
    - Intel Itanium 1 and 2 contain 6 operations per instruction packet
  - Need compiling technique that schedules across several branches

VLIW: Very Long Instruction Word



- Multiple operations packed into one instruction
- Each operation slot is for a fixed function
- Constant operation latencies are specified
- Architecture requires guarantee of:
  - Parallelism within an instruction → no cross-operation RAW check
  - No data use before data ready → no data interlocks

VLIW Compiler Responsibilities

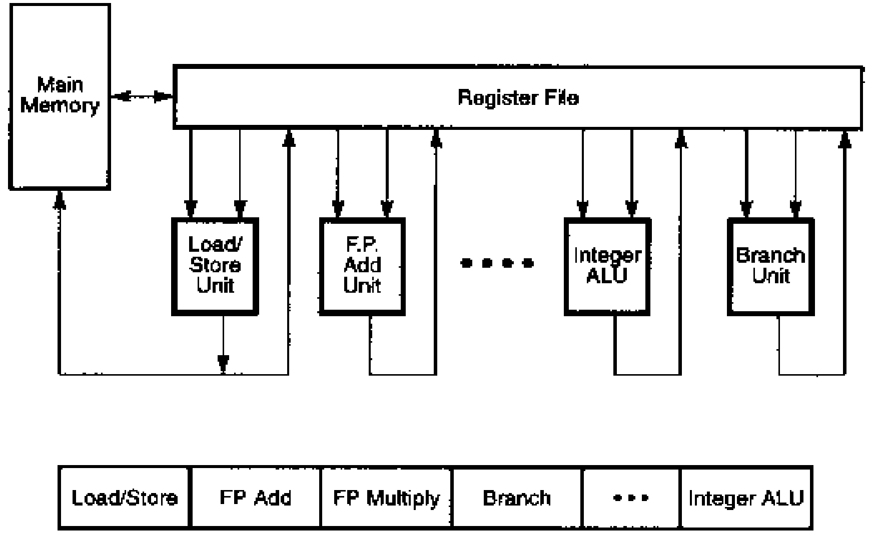
- Schedule operations to maximize parallel execution
- Guarantees intra-instruction parallelism
- Schedule to avoid data hazards (no interlocks)
  - Typically separates operations with explicit NOPs

- In a VLIW (also called Very Large Instruction Word) processor, several operations that can be executed in parallel are placed an a single instruction word.

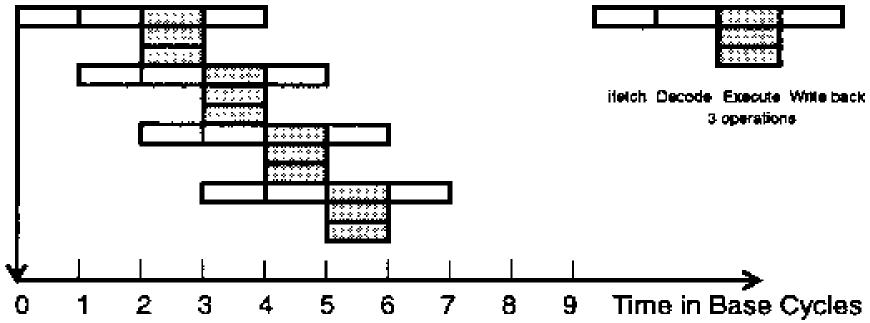
Instruction 1	op_1	op_2	op_3	op_4
Instruction 2	op_1	Ø	op_3	op_4
Instruction 3	Ø	op_2	op_3	Ø

- VLIW architectures rely on compile-time detection of parallelism.
  - The compiler analyzes the program and detects operations to be executed in parallel.
- After one instruction has been fetched all the corresponding operations are issued in parallel.
  - No hardware is needed for run-time detection of parallelism.
- The instruction window problem disappears: the compiler can potentially analyze the whole program to detect parallel operations.

Typical VLIW processor



(a) A typical VLIW processor and instruction format



(b) VLIW execution with degree $m = 3$

Figure 4.14 The architecture of a very long instruction word (VLIW) processor and its pipeline operations. (Courtesy of Multiflow Computer, Inc., 1987)

Loop Unrolling in VLIW

Memory reference 1	*Memory reference 2*	*FP operation 1*	*FP operation 2*	*Int. op/ branch*	*Clock*
LD F0,0(R1)	LD F6,-8(R1)				1
LD F10,-16(R1)	LD F14,-24(R1)				2
LD F18,-32(R1)	LD F22,-40(R1)	ADDD F4,F0,F2	ADDD F8,F6,F2		3
LD F26,-48(R1)		ADDD F12,F10,F2	ADDD F16,F14,F2		4
		ADDD F20,F18,F2	ADDD F24,F22,F2		5
SD 0(R1),F4	SD -8(R1),F8	ADDD F28,F26,F2			6
SD -16(R1),F12	SD -24(R1),F16				7
SD -32(R1),F20	SD -40(R1),F24			SUBI R1,R1,#56	8
SD 8(R1),F28				BNEZ R1,LOOP	9

- Unrolled 7 times to avoid delays
- Unrolling 7 times results in 9 clocks, or 1.3 clocks per iteration (1.8x vs SS)
- Average: 2.5 ops per clock, 50% efficiency
- Note: Need more registers in VLIW (15 vs. 6 in SS)

Advantages of VLIW

- Compiler prepares fixed packets of multiple operations that give the full "plan of execution"
  - dependencies are determined by compiler and used to schedule according to function unit latencies
  - function units are assigned by compiler and correspond to the position within the instruction packet ("slotting")
  - compiler produces fully-scheduled, hazard-free code → hardware doesn't have to "rediscover" dependencies or schedule

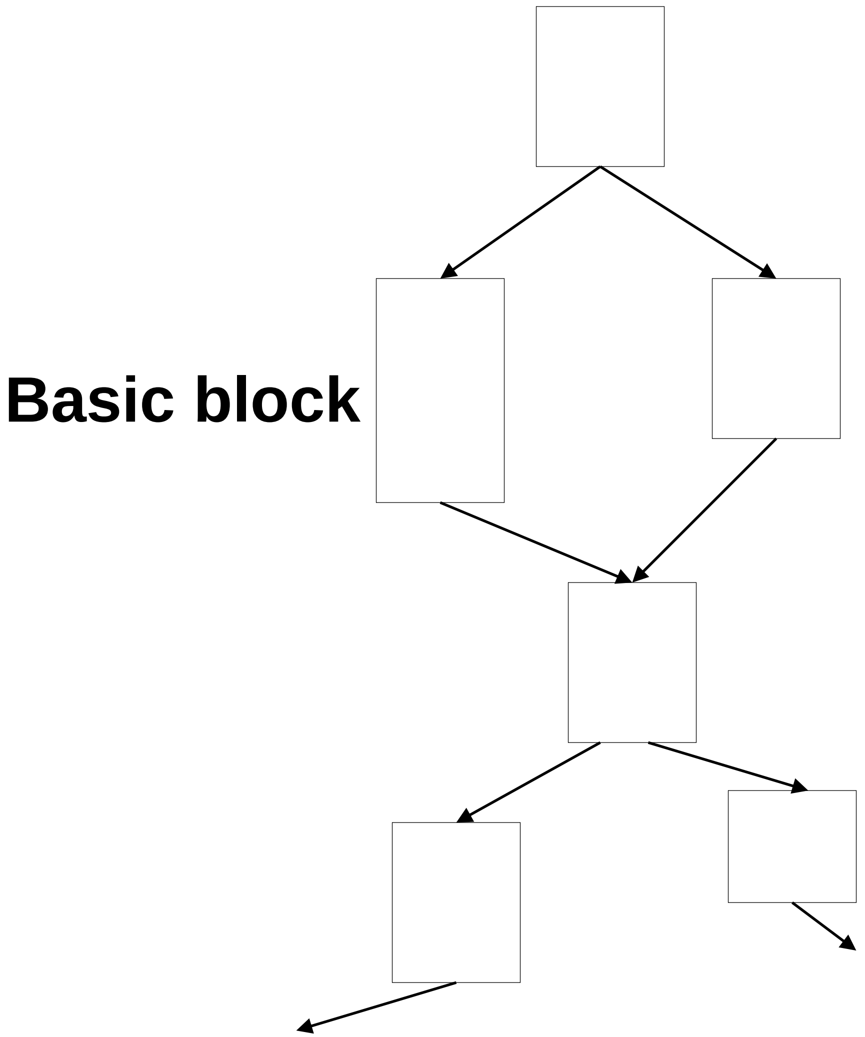
Disadvantages of VLIW

- Object-code compatibility
  - recompile all code for every machine, even for two machines in same generation
- Object code size
  - instruction padding wastes instruction memory/cache
  - loop unrolling/software pipelining replicates code
- Scheduling variable latency memory operations
  - caches and/or memory bank conflicts impose statically unpredictable variability
  - as the issue rate and number of memory references becomes large, this synchronization restriction becomes unacceptable
- Knowing branch probabilities
  - Profiling requires a significant extra step in build process
- Scheduling for statically unpredictable branches
  - optimal schedule varies with branch path

What if there are not many loops?

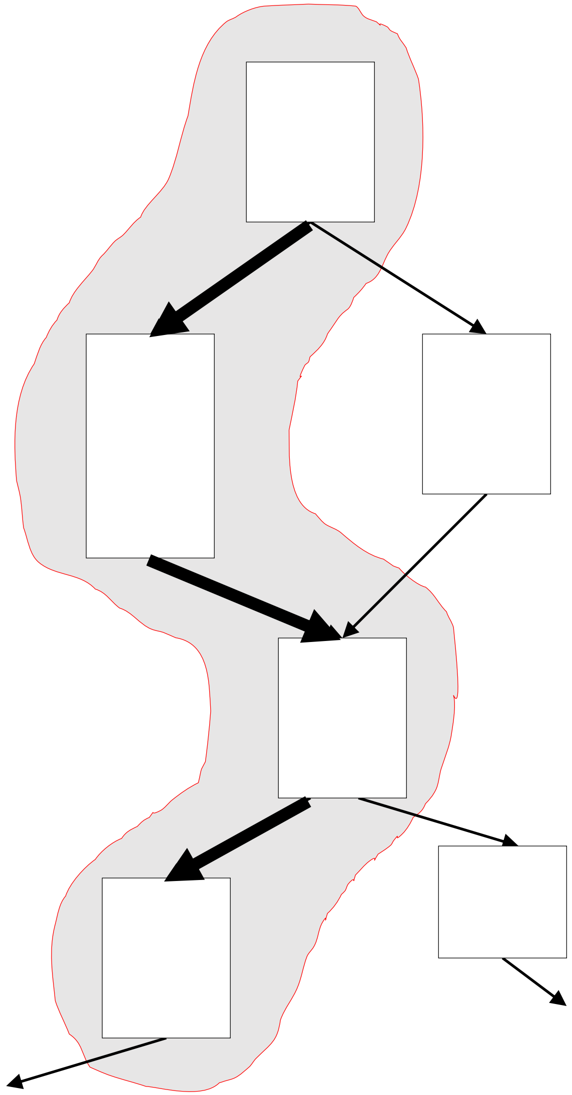
- Branches limit basic block size in control-flow intensive irregular code

- Difficult to find ILP in individual basic blocks



Trace Scheduling [Fisher, Ellis]

- **Trace selection:** Pick string of basic blocks, a trace, that represents most frequent branch path
- Use profiling feedback or compiler heuristics to find common branch paths
- **Trace Compaction:** Schedule whole “trace” at once. Packing operations to few wide instructions
- Add fixup code to cope with branches jumping out of trace
- Effective to certain classes of programs
- Key assumption is that the trace is much more probable than the alternatives



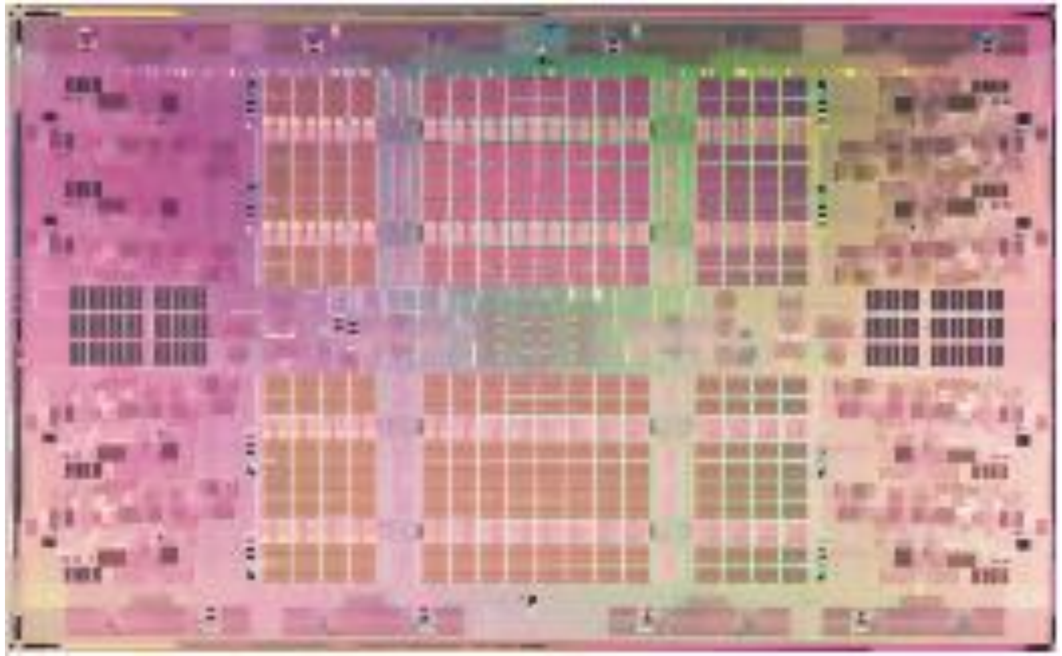
Intel Itanium, EPIC IA-64

- EPIC is the style of architecture (cf. CISC, RISC)
  - Explicitly Parallel Instruction Computing (really just VLIW)
- IA-64 (Intel Itanium architecture) is Intel's chosen ISA (cf. x86, MIPS)
  - IA-64 = Intel Architecture 64-bit
  - An object-code-compatible VLIW
- Merced was first Itanium implementation (cf. 8086)
  - First customer shipment expected 1997 (actually 2001)
  - McKinley, second implementation shipped in 2002
  - Recent version, Poulson, eight cores, 32nm, 2012
- Different instruction format than VLIW architectures using with indicators
- Support for SW speculation

Eight Core Itanium “Poulson” [Intel 2012]

- 8 cores
- 1-cycle 16KB L1 I&D caches
- 9-cycle 512KB L2 I-cache
- 8-cycle 256KB L2 D-cache
- 32 MB shared L3 cache
- 544mm^2 in 32nm CMOS
- Over 3 billion transistors
- Cores are 2-way multithreaded
- 6 instruction/cycle fetch
  - Two 128-bit bundles (3 instrs/bundle)

– Up to 12 instrs/cycle execute



IA-64 Registers

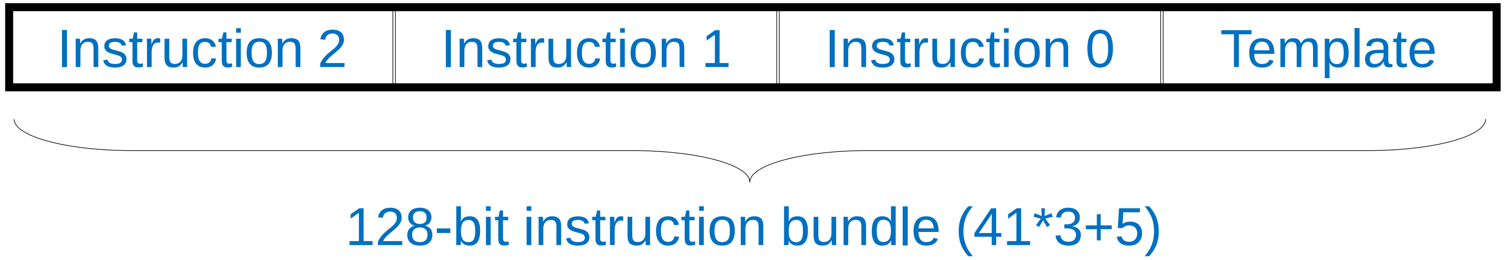
- 128 General Purpose 64-bit Integer Registers
- 128 General Purpose 82-bit Floating Point Registers
- 64 x 1-bit Predicate Registers
- 8 x 64-bit Branch Registers

- Register stack mechanism: GPRs “rotate” to reduce code size for software pipelined loops
  - Rotation is a simple form of register renaming allowing one instruction to address different physical registers on each procedure call

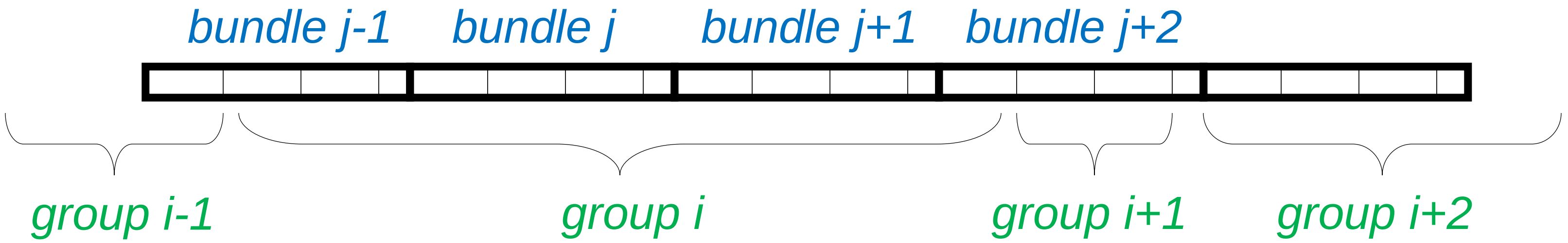
Execution Units

Execution unit slot	Instruction type	Instruction description	Example instructions
I-unit	A	Integer ALU	add, subtract, and, or, compare
	I	Non-ALU integer	integer and multimedia shifts, bit tests, moves
M-unit	A	Integer ALU	add, subtract, and, or, compare
	M	Memory access	Loads and stores for integer/FP registers
F-unit	F	Floating point	Floating-point instructions
B-unit	B	Branches	Conditional branches, calls, loop branches
L + X	L + X	Extended	Extended immediates, stops and no-ops

IA-64 Instruction Format



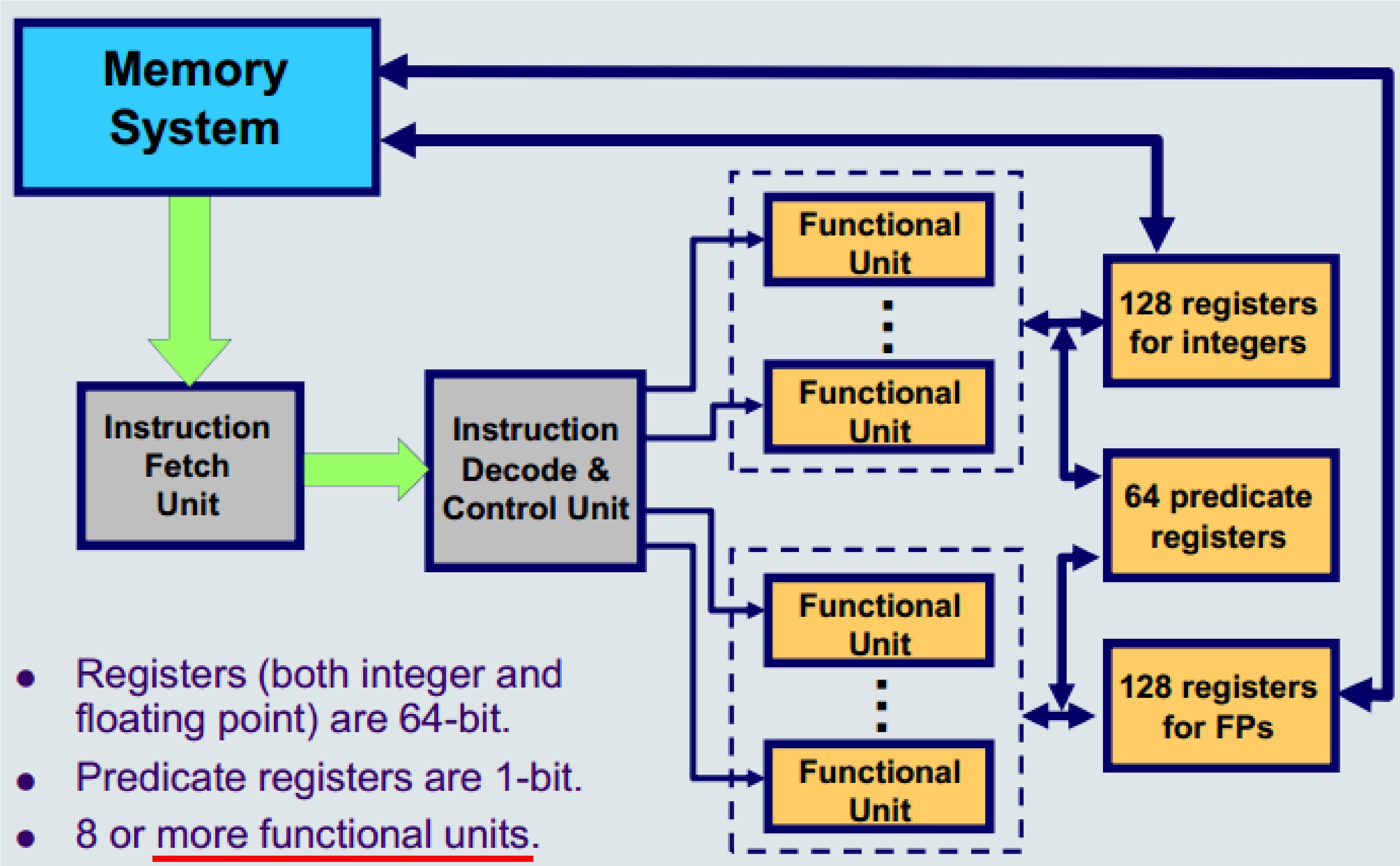
- Template bits describe the types of instructions and the grouping of these instructions with others in adjacent bundles
- Each group contains instructions that can execute in parallel and the boundary of a group (stop) is indicated by the template



IA-64 Template

Template	Slot 0	Slot 1	Slot 2
0	M	I	I
1	M	I	I
2	M	I	I
3	M	I	I
4	M	L	X
5	M	L	X
8	M	M	I
9	M	M	I
10	M	M	I
11	M	M	I
12	M	F	I
13	M	F	I
14	M	M	F
15	M	M	F
16	M	I	B
17	M	I	B
18	M	B	B
19	M	B	B
22	B	B	B
23	B	B	B
24	M	M	B
25	M	M	B
28	M	F	B
29	M	F	B

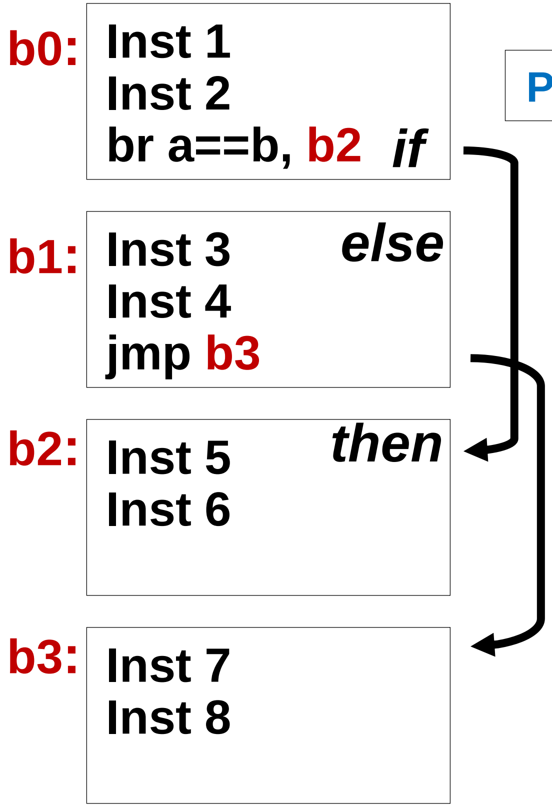
IA-64 Basic Architecture



- Registers (both integer and floating point) are 64-bit.
- Predicate registers are 1-bit.
- 8 or more functional units.

IA-64 Predicated Execution

- **Problem:** Mispredicted branches limit ILP
- **Solution:** Eliminate hard to predict branches with predicated execution
  - Almost all IA-64 instructions can be executed conditionally under predicate
  - Instruction becomes NOP if predicate register false



Four basic blocks

Predication

Inst 1
Inst 2
p1= (a!=b), p2 = (a==b)
(p1) Inst 3 || (p2) Inst 5
(p1) Inst 4 || (p2) Inst 6
Inst 7
Inst 8

One basic block

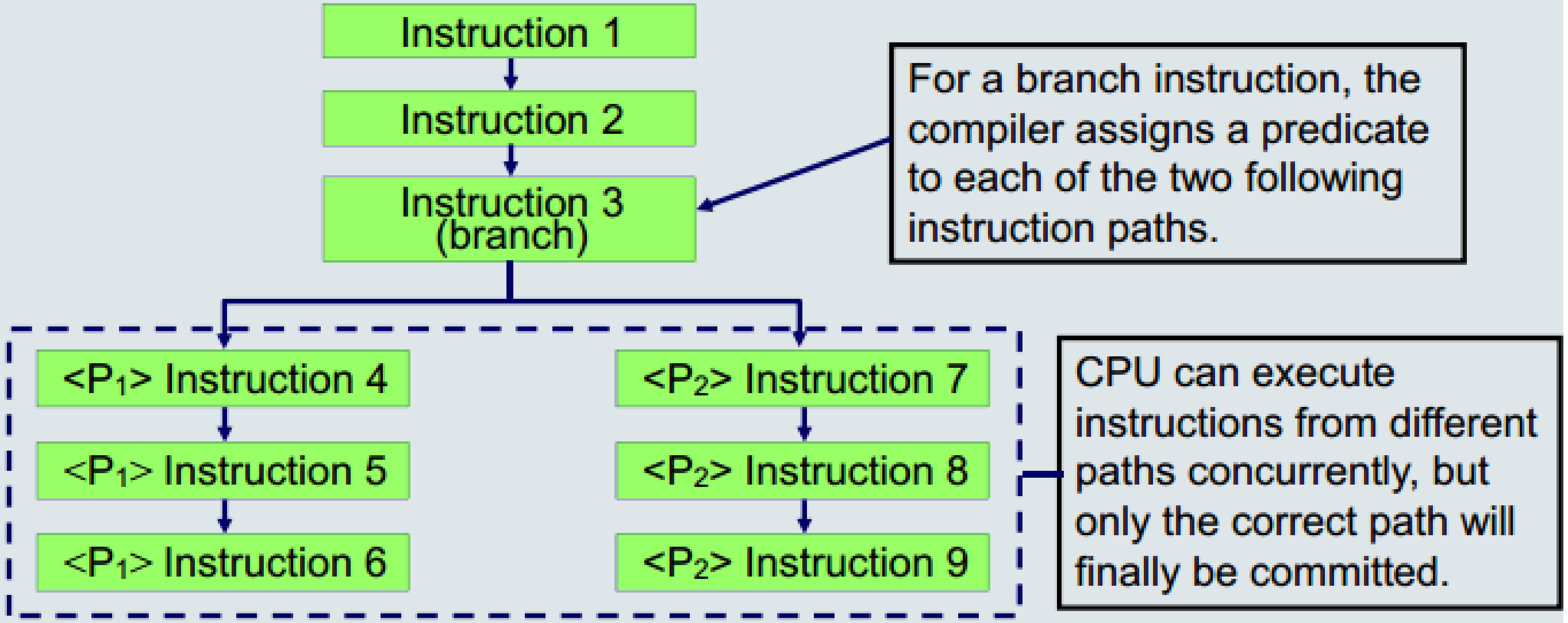
Mahlke et al, ISCA95:
On average >50% branches removed

Branch Predication

- **Branch predication** is an aggressive compilation technique to generate code with higher degree of instruction level parallelism.
- It lets operations from both branches of a conditional branch to be executed in parallel, to increase the amount of parallel operations.
- In this way, branches are eliminated and replaced by conditional execution.
  - Hardware support is needed, as implemented in the IA-64 architecture.

<u>The idea is:</u> let instructions from both branches go on in parallel, before the branch condition has been evaluated. The hardware takes care that only those corresponding to the right branch will be finally committed.

Branch Predication Example



For a VLIW machine, the instructions may be arranged as follows:

Instruction 1	Instruction 2	Instruction 3
<P_1> Instruction 4	<P_2> Instruction 7	<P_1> Instruction 5
<P_2> Instruction 8	<P_1> Instruction 6	<P_2> Instruction 9