Lecture 4
Instruction Level Parallelism (2)

EEC 171 Parallel Architectures

John Owens

UC Davis

Credits

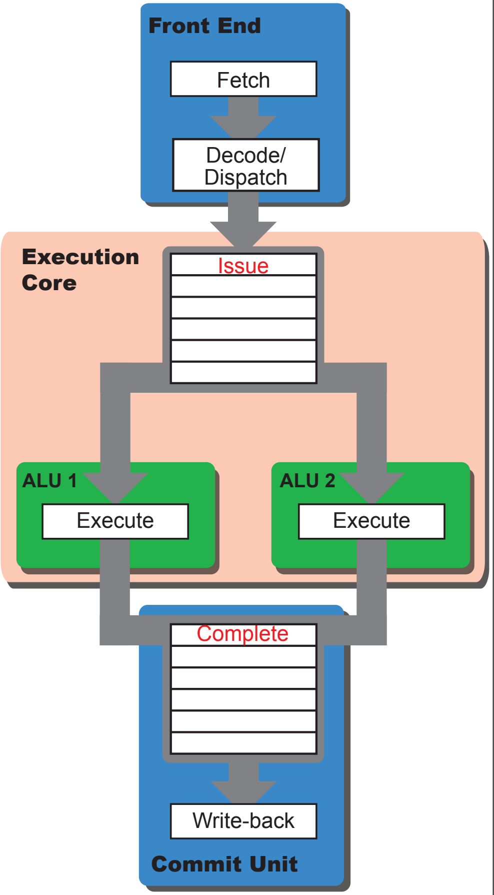
- © John Owens / UC Davis 2007–9.
- Thanks to many sources for slide material: Computer Organization and Design (Patterson & Hennessy) © 2005, Computer Architecture (Hennessy & Patterson) © 2007, Inside the Machine (Jon Stokes) © 2007, © Dan Connors / University of Colorado 2007, © Wen-Mei Hwu/David Kirk, University of Illinois 2007, © David Patterson / UCB 2003–6, © John Lazzaro / UCB 2006, © Mary Jane Irwin / Penn State 2005, © John Kubiatowicz / UCB 2002, © Krste Asinovic/Arvind / MIT 2002, © Morgan Kaufmann Publishers 1998.

Today’s Goals

- Out-of-order execution
- An alternate approach to machine parallelism: software scheduling & VLIW
- How do we ensure we have ample instruction-level parallelism?
  - Branch prediction

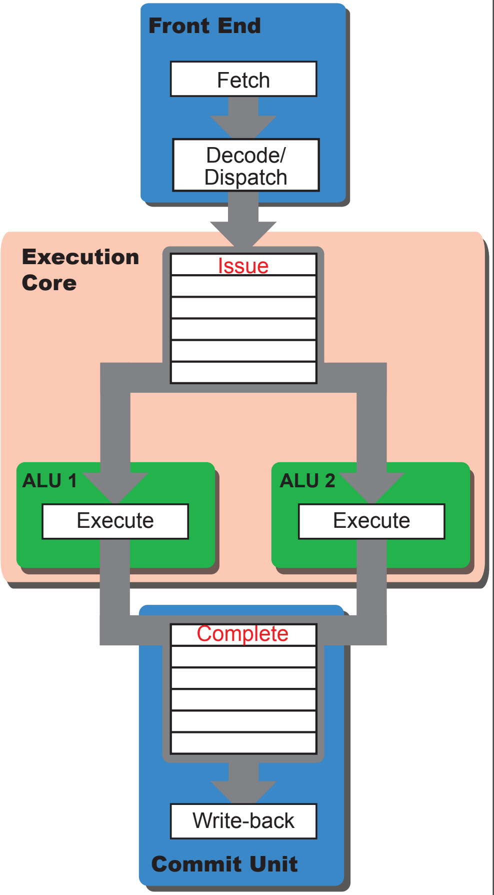
Pentium Retrospective

- Limited in performance by “front end”
  - Has to support variable-length instrs and segments
- Supporting all x86 features tough!
  - 30% of transistors are for legacy support
    - Up to 40% in Pentium Pro!
    - Down to 10% in P4
  - Microcode ROM is huge



Pentium Retrospective

- Pentium is in-order issue, in-order complete
- "Static scheduling" by the dispatch logic:
  - Fetch/dispatch/execute/retire: all in order
- Drawbacks:
  - Adapts poorly to dynamic code stream
  - Adapts poorly to future hardware
    - What if we had 3 pipes not 2?



Multiple-Issue Datapath Responsibilities

- Must handle, with a combination of hardware and software fixes, the fundamental limitations of
  - Storage (data) dependencies—aka data hazards
    - Most instruction streams do not have huge ILP so ...
    - ... this limits performance in a superscalar processor

Multiple-Issue Datapath Responsibilities

- Must handle, with a combination of hardware and software fixes, the fundamental limitations of
  - Procedural dependencies—aka control hazards
    - Ditto, but even more severe
    - Use dynamic branch prediction to help resolve the ILP issue
      - Future lecture

Multiple-Issue Datapath Responsibilities

- Must handle, with a combination of hardware and software fixes, the fundamental limitations of
  - Resource conflicts—aka structural hazards
    - A SS/VLIW processor has a much larger number of potential resource conflicts
    - Functional units may have to arbitrate for result buses and register-file write ports
    - Resource conflicts can be eliminated by duplicating the resource or by pipelining the resource

Instruction Issue and Completion Policies

- Instruction-issue—initiate execution
  - Instruction lookahead capability—fetch, decode and issue instructions beyond the current instruction
- Instruction-completion—complete execution
  - Processor lookahead capability—complete issued instructions beyond the current instruction
- Instruction-commit—write back results to the RegFile or D$ (i.e., change the machine state)

In-order issue with in-order completion
In-order issue with out-of-order completion
Out-of-order issue with out-of-order completion and in-order commit
Out-of-order issue with out-of-order completion

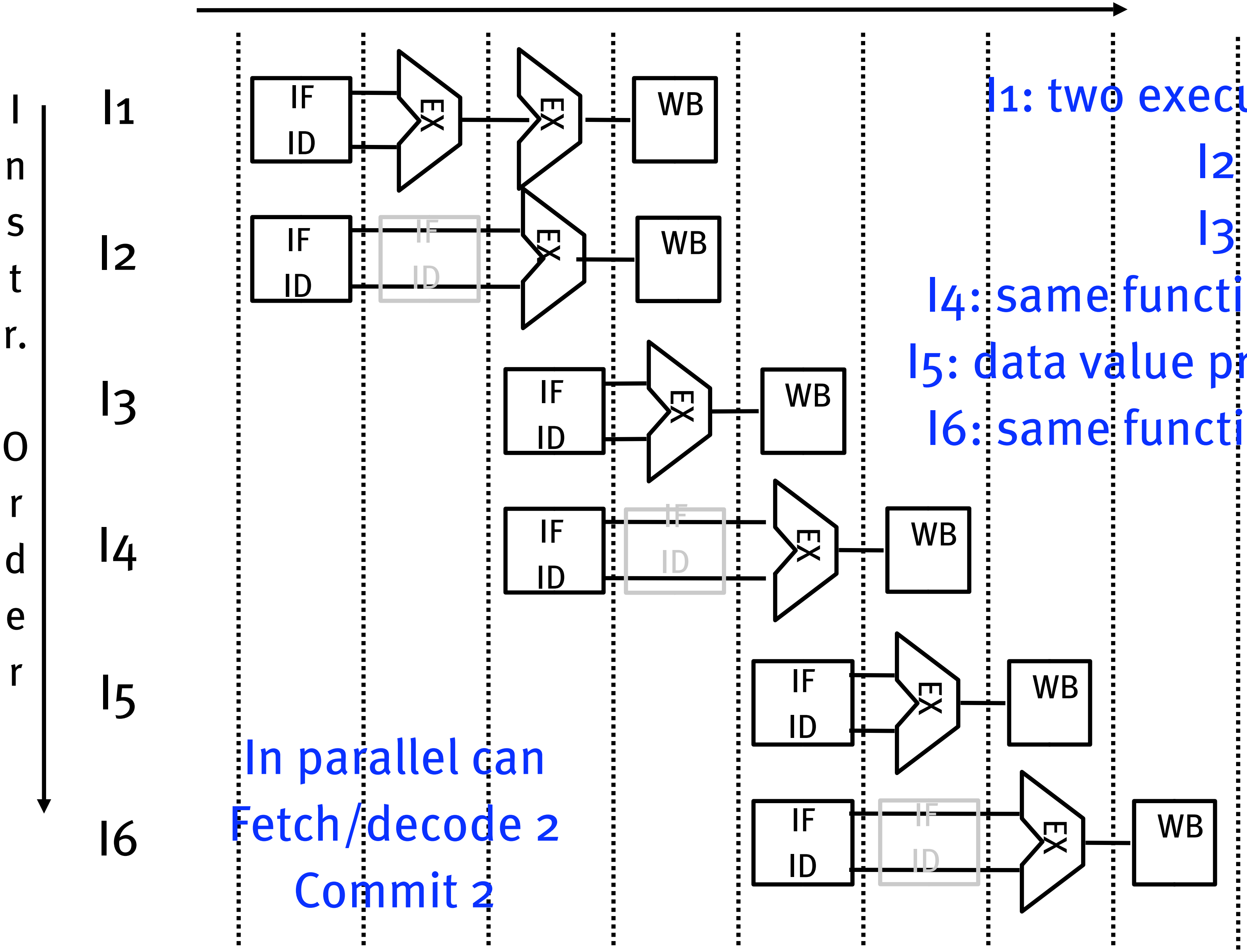
In-Order Issue with In-Order Completion

- Simplest policy is to issue instructions in exact program order and to complete them in the same order they were fetched (i.e., in program order)

In-Order Issue with In-Order Completion (Ex.)

- Assume a pipelined processor that can fetch and decode two instructions per cycle, that has three functional units (a single cycle adder, a single cycle shifter, and a two cycle multiplier), and that can complete (and write back) two results per cycle
- Instruction sequence:
  I1: needs two execute cycles (a multiply)
  I2
  I3
  I4: needs the same function unit as I3
  I5: needs data value produced by I4
  I6: needs the same function unit as I5

In-Order Issue, In-Order Completion Example



I1: two execute cycles

I2

I3

I4: same function unit as I3

I5: data value produced by I4

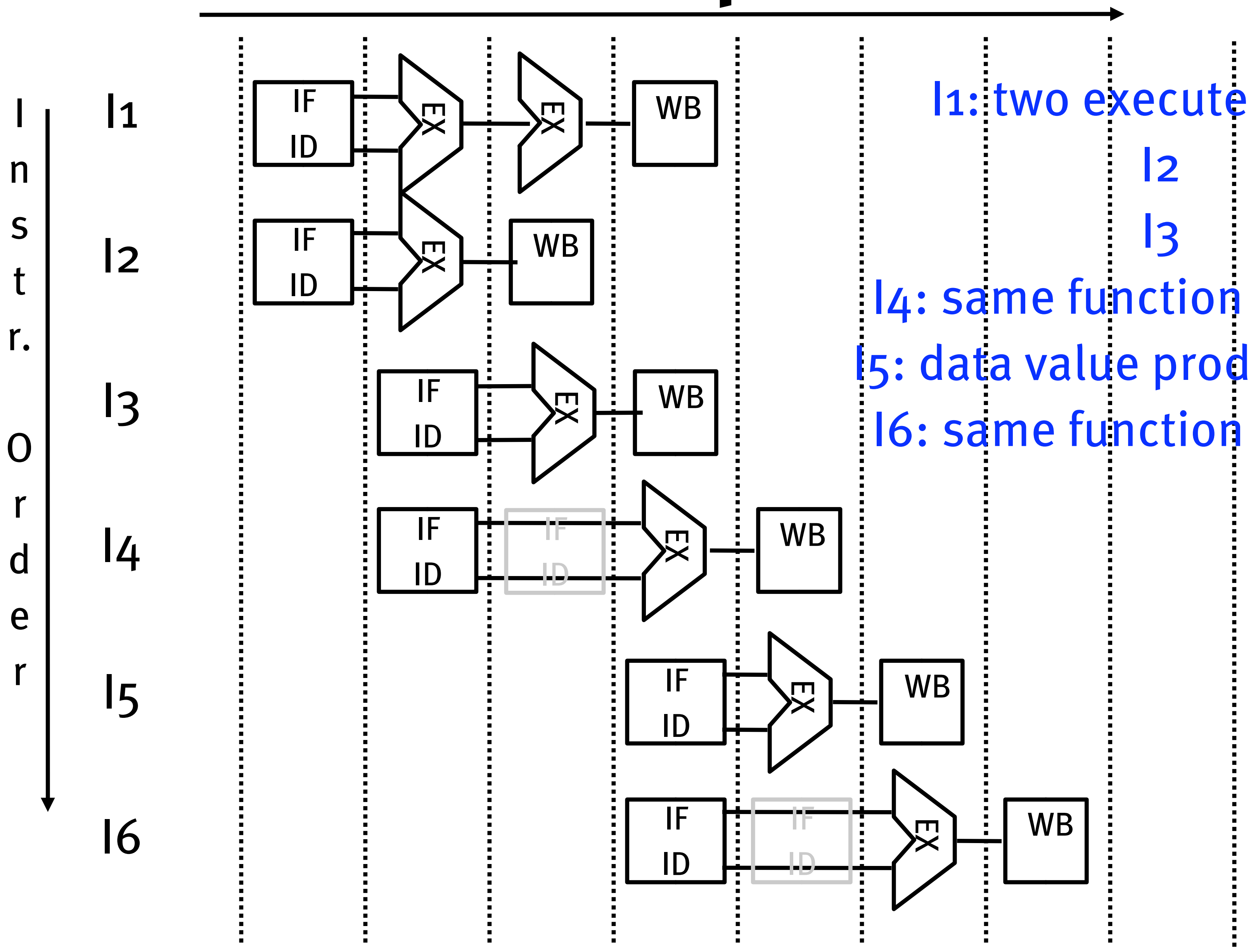
I6: same function unit as I5

In parallel can
Fetch/decode 2
Commit 2

In-Order Issue with Out-of-Order Completion

- With out-of-order completion, a later instruction may complete before a previous instruction
  - Out-of-order completion is used in single-issue pipelined processors to improve the performance of long-latency operations such as divide
- When using out-of-order completion instruction issue is stalled when there is a resource conflict (e.g., for a functional unit) or when the instructions ready to issue need a result that has not yet been computed

IOI-OOC Example



I1: two execute cycles

I2

I3

I4: same function unit as I3

I5: data value produced by I4

I6: same function unit as I5

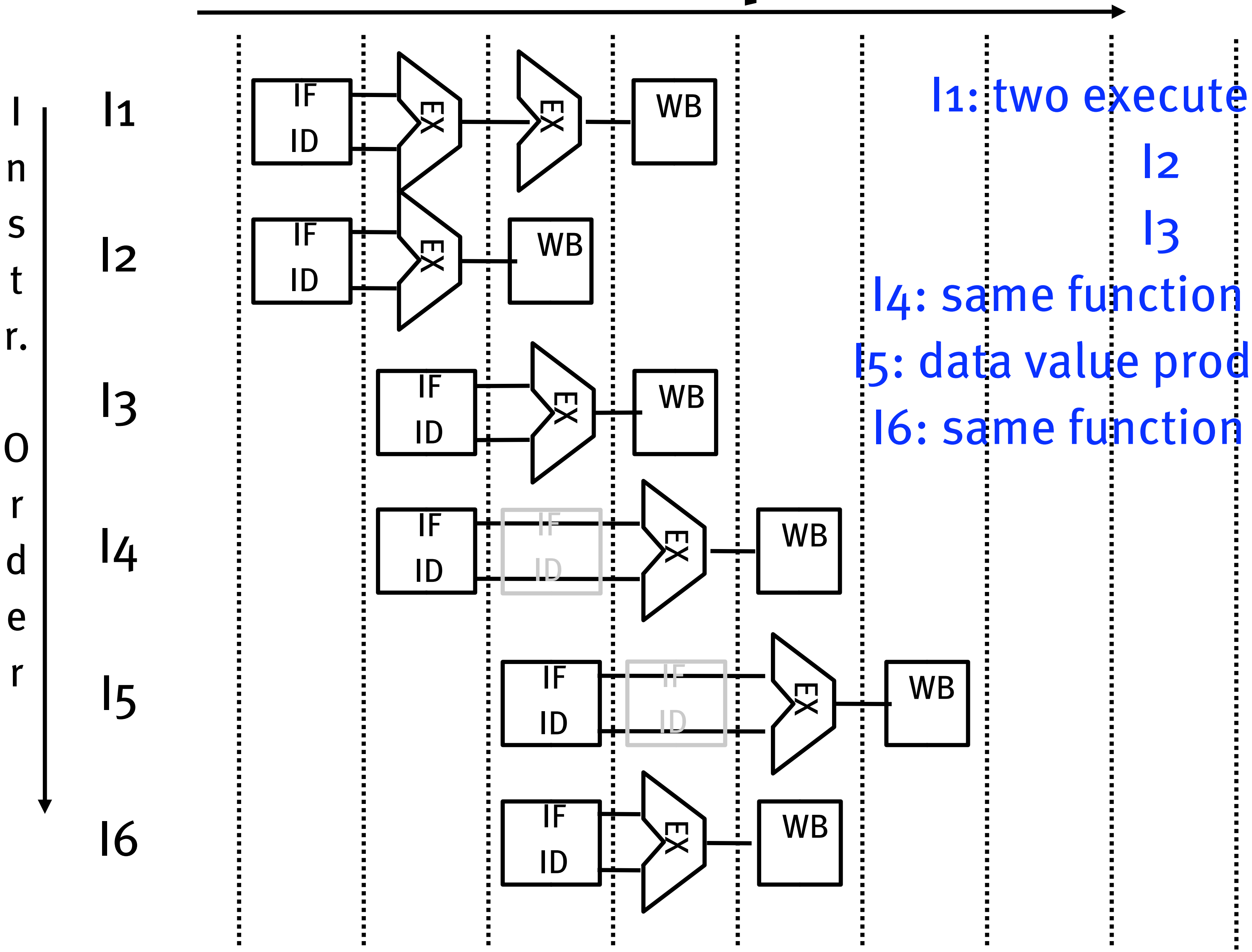
Handling Output Dependencies

- There is one more situation that stalls instruction issuing with IOI-OOC, assume
  - I1 – writes to R3
    I2 – writes to R3
    I5 – reads R3
- If the I1 write occurs after the I2 write, then I5 reads an incorrect value for R3
- I2 has an output dependency on I1—write before write
- The issuing of I2 would have to be stalled if its result might later be overwritten by an previous instruction (i.e., I1) that takes longer to complete—the stall happens before instruction issue
- While IOI-OOC yields higher performance, it requires more dependency checking hardware (both read-before-write and write-before-write)

Out-of-Order Issue with Out-of-Order Completion

- With in-order issue the processor stops decoding instructions whenever a decoded instruction has a resource conflict or a data dependency on an issued, but uncompleted instruction
    - The processor is not able to look beyond the conflicted instruction even though more downstream instructions might have no conflicts and thus be issueable
- Fetch and decode instructions beyond the conflicted one ("instruction window": Tetris), store them in an instruction buffer (as long as there's room), and flag those instructions in the buffer that don't have resource conflicts or data dependencies
- Flagged instructions are then issued from the buffer without regard to their program order

OOI-OOC Example



I1: two execute cycles

I2

I3

I4: same function unit as I3

I5: data value produced by I4

I6: same function unit as I5

Dependency Examples

- R3 := R3 * R5
  R4 := R3 + 1
  R3 := R5 + 1

True data dependency (RAW)

Output dependency (WAW)

Antidependency (WAR)

Antidependencies

- With OOI also have to deal with data antidependencies – when a later instruction (that completes earlier) produces a data value that destroys a data value used as a source in an earlier instruction (that issues later)
- The constraint is similar to that of true data dependencies, except reversed
  - Instead of the later instruction using a value (not yet) produced by an earlier instruction (read before write), the later instruction produces a value that destroys a value that the earlier instruction (has not yet) used (write before read)

Dependencies Review

- Each of the three data dependencies …
  - True data dependencies (read before write)
  - Antidependencies (write before read)
  - Output dependencies (write before write)

  storage conflicts
- … manifests itself through the use of registers (or other storage locations)
- True dependencies represent the flow of data and information through a program
- Anti- and output dependencies arise because the limited number of registers mean that programmers reuse registers for different computations
- When instructions are issued out-of-order, the correspondence between registers and values breaks down and the values conflict for registers

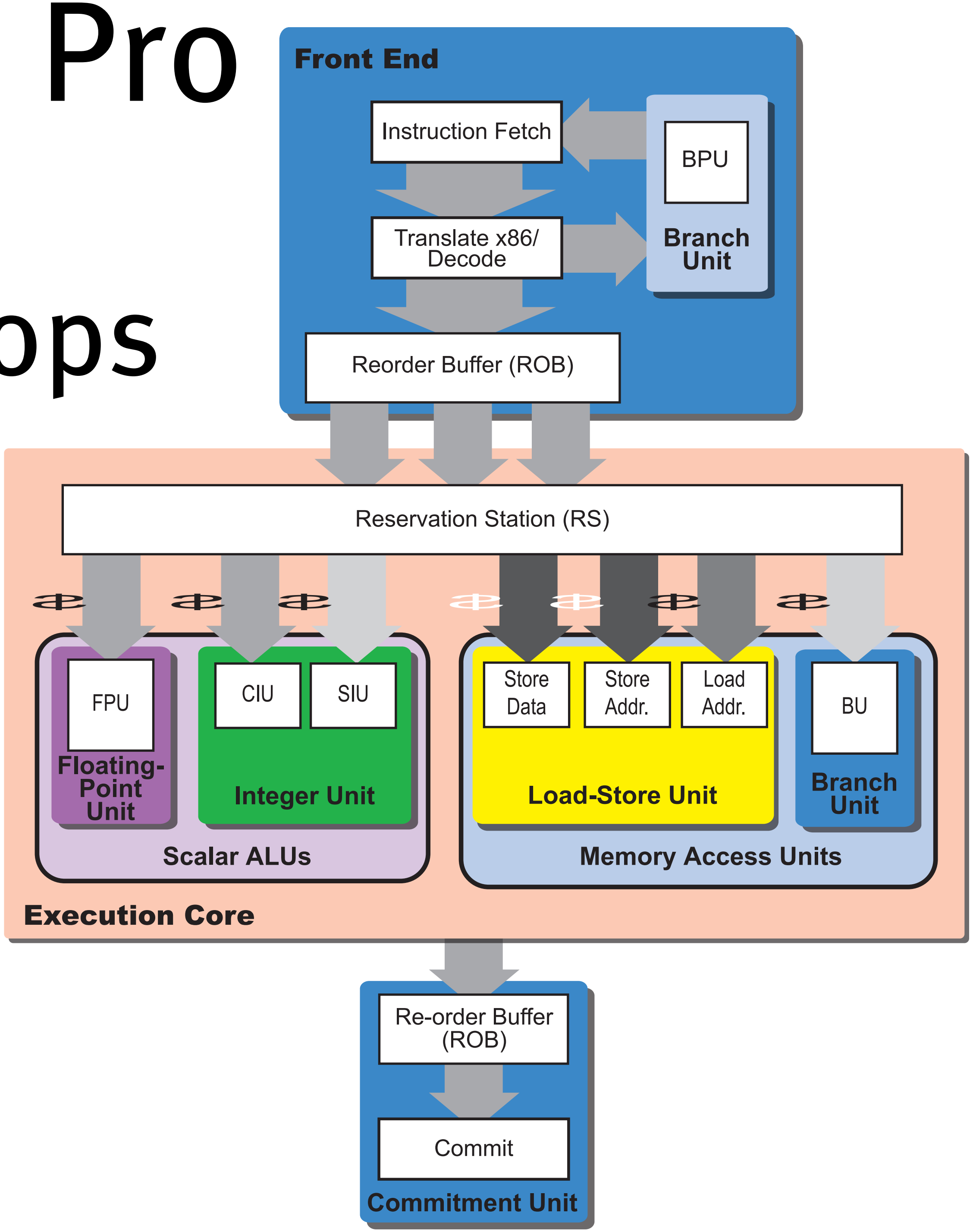
Storage Conflicts and Register Renaming

- Storage conflicts can be reduced (or eliminated) by increasing or duplicating the troublesome resource
  - Provide additional registers that are used to reestablish the correspondence between registers and values
    - Allocated dynamically by the hardware in SS processors
- Register renaming — the processor renames the original register identifier in the instruction to a new register (one not in the visible register set)

  -
    ```
    R3 := R3 * R5        R3b := R3a * R5a
    R4 := R3 + 1         R4a := R3b + 1
    R3 := R5 + 1         R3c := R5a + 1
    ```
- The hardware that does renaming assigns a “replacement” register from a pool of free registers and releases it back to the pool when its value is superseded and there are no outstanding references to it [future lecture!]

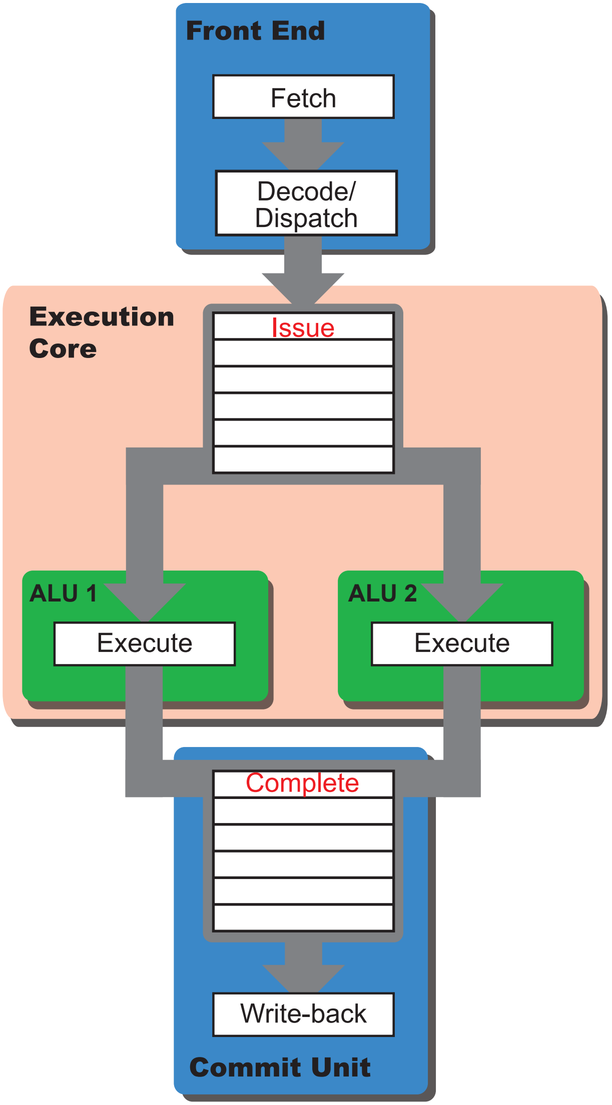
Pentium Pro

uops



Pentium Pro

1. Fetch	In order
2. Decode/dispatch	In order
3. Issue	Reorder
4. Execute	Out of order
5. Complete	Reorder
6. Writeback (commit)	In order

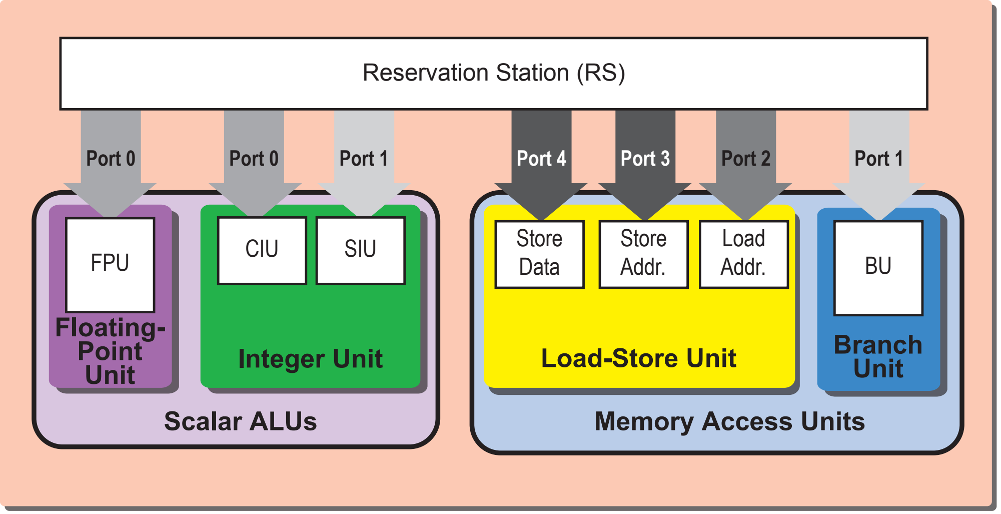


P6 Pipeline

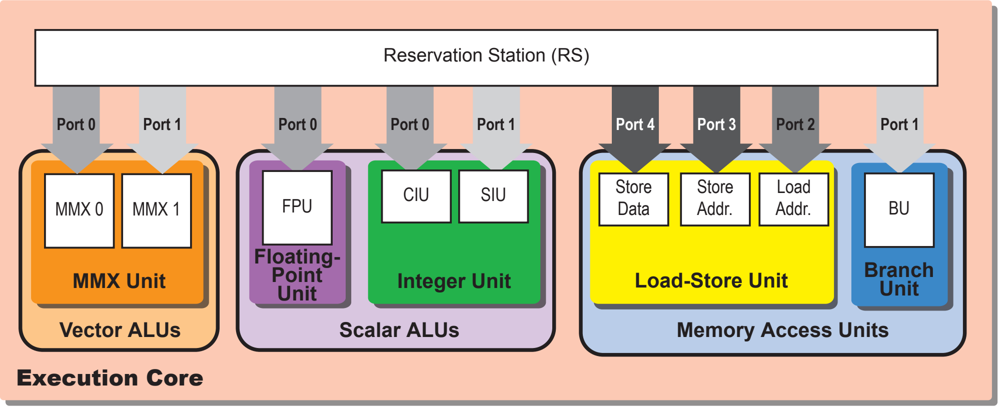
- Instruction fetch, BTB access (3.5 stages)
  - 2 cycles for instruction fetch
- Decode, x86->uops (2.5 stages)
- Register rename (1 stage)
- Write to reservation station (1 stage)
- Read from reservation station (1 stage)
- Execute (1+ stages)
- Commit (2 stages)

Pentium Pro backends

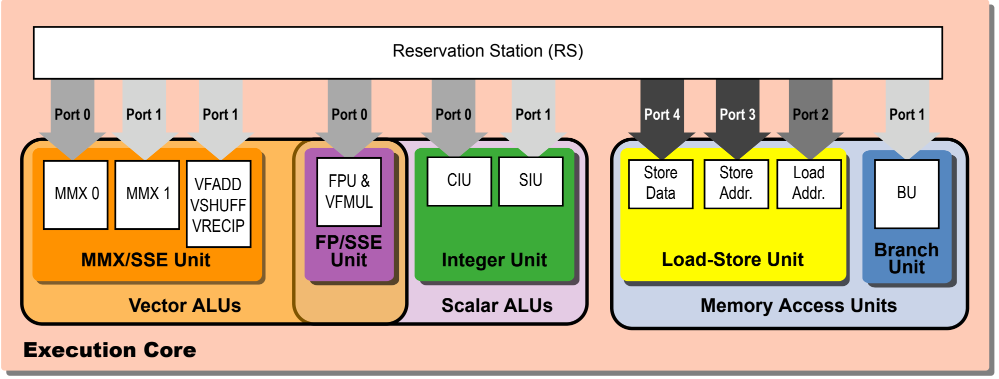
- Pentium Pro



- Pentium 2



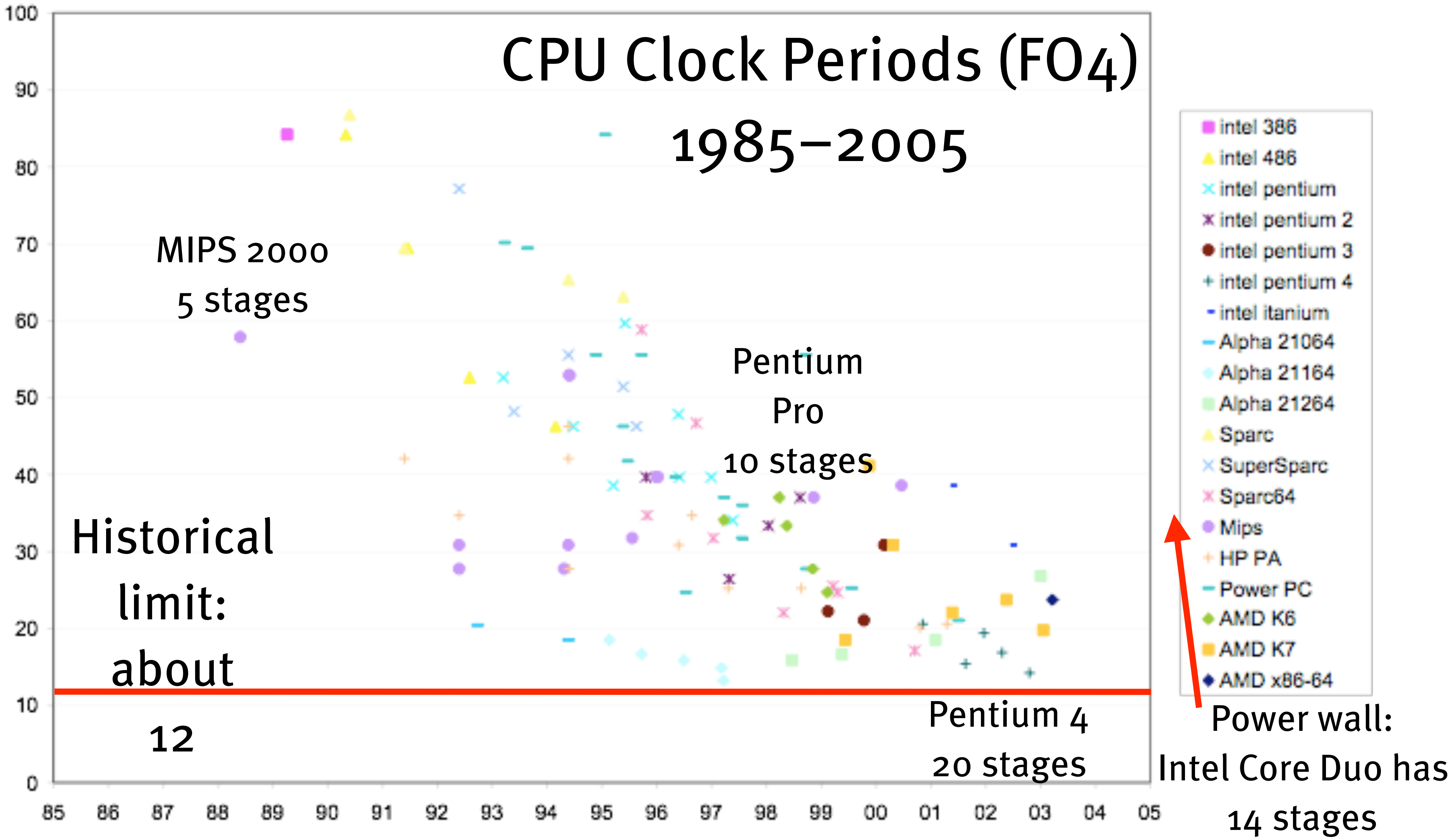
- Pentium 3



Where Do We Get ILP?

- All of these techniques require that we have ample instruction level parallelism
    - Original P4 has 20 stages, 6 μops per cycle
    - Lots of instructions in flight!

Hardware limits to superpipelining?



CPU Clock Periods (FO4)

1985–2005

MIPS 2000
5 stages

Pentium
Pro
10 stages

Historical
limit:
about
12

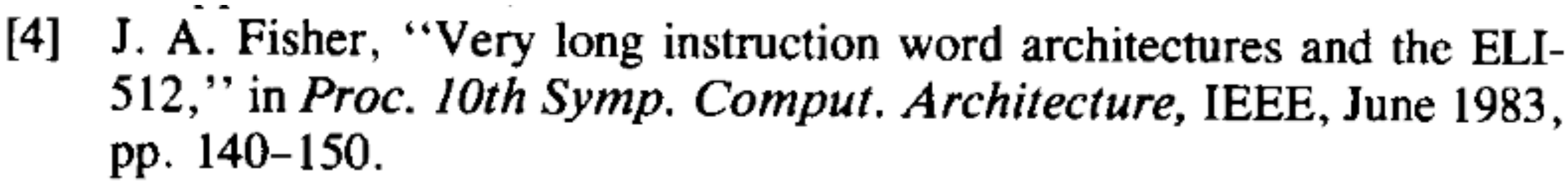
Pentium 4
20 stages

Power wall:
Intel Core Duo has
14 stages

courtesy François Labonte, Stanford

VLIW Beginnings

- VLIW: Very Long Instruction Word



[4] J. A. Fisher, "Very long instruction word architectures and the ELI-512," in *Proc. 10th Symp. Comput. Architecture*, IEEE, June 1983, pp. 140–150.

- Josh Fisher: idea grew out of his Ph.D (1979) in compilers
- Led to a startup (MultiFlow) whose computers worked, but which went out of business ... the ideas remain influential.

History of VLIW Processors

- Started with (horizontal) microprogramming
  - Very wide microinstructions used to directly generate control signals in single-issue processors (e.g., IBM 360 series)
- VLIW for multi-issue processors first appeared in the Multiflow and Cydrome (in the early 1980's)
- Current commercial VLIW processors
  - Intel i860 RISC (dual mode: scalar and VLIW)
  - Intel I-64 (EPIC: Itanium and Itanium 2) [future lecture]
  - Transmeta Crusoe
  - Lucent/Motorola StarCore, ADI TigerSHARC, Infineon (Siemens) Carmel

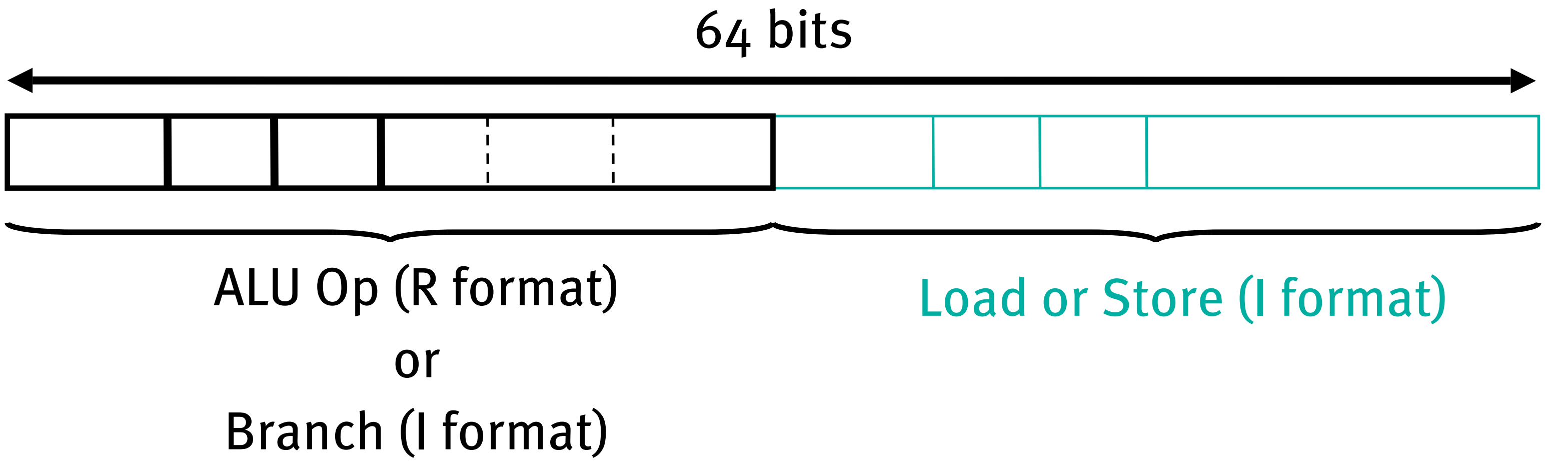
Static Multiple Issue Machines (VLIW)

- Static multiple-issue processors (aka VLIW) use the compiler to decide which instructions to issue and execute simultaneously
    - Issue packet—the set of instructions that are bundled together and issued in one clock cycle—think of it as one large instruction with multiple operations
    - The mix of instructions in the packet (bundle) is usually restricted—a single “instruction” with several predefined fields
    - The compiler does static branch prediction and code scheduling to reduce (ctrl) or eliminate (data) hazards

Static Multiple Issue Machines (VLIW)

- VLIW's have
  - Multiple functional units (like SS processors)
  - Multi-ported register files (again like SS processors)
  - Wide program bus

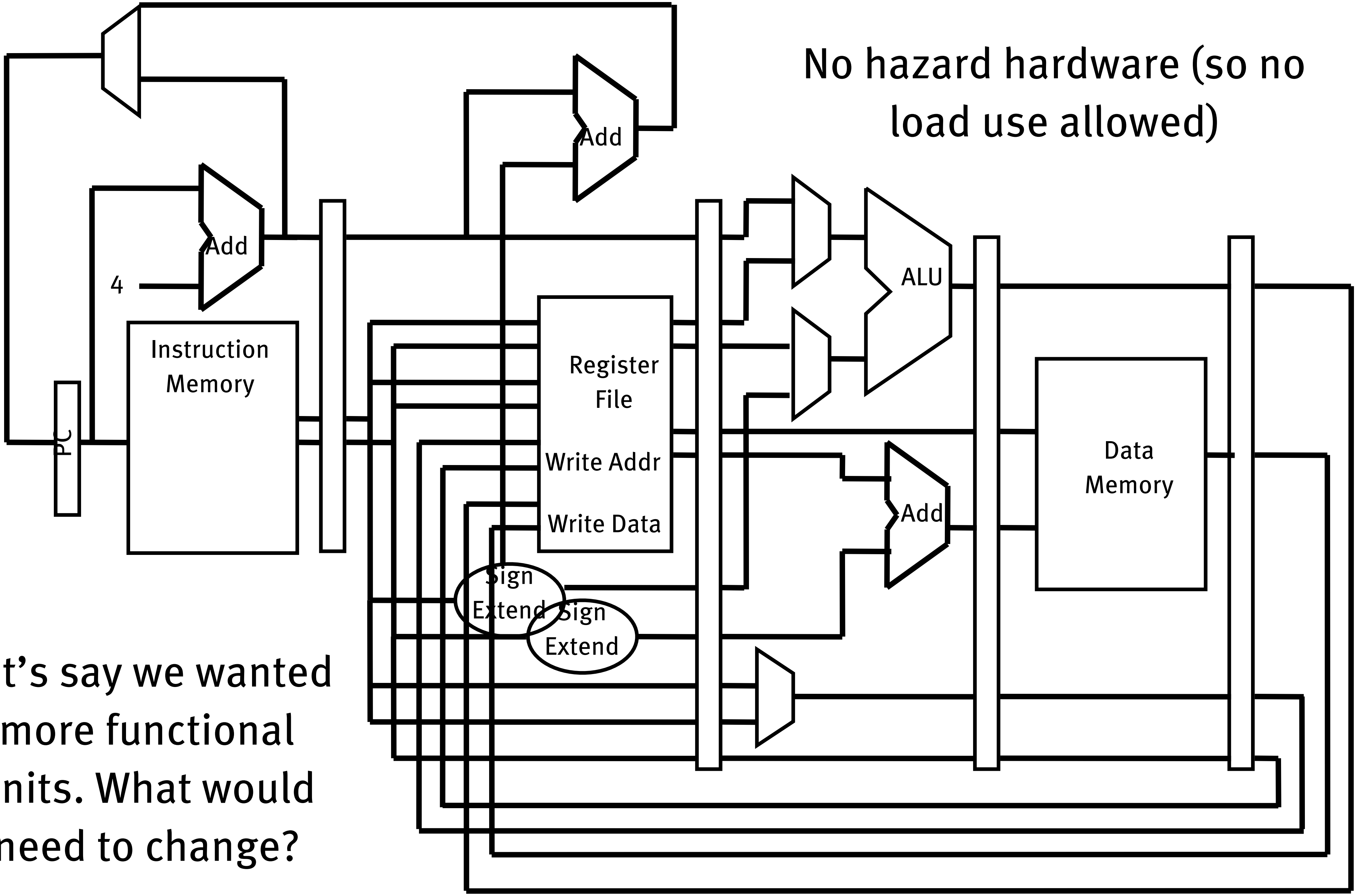
An Example: A VLIW MIPS



- Consider a 2-issue MIPS with a 2 instr bundle
- Instructions are always fetched, decoded, and issued in pairs
  - If one instr of the pair can not be used, it is replaced with a noop
  - Need 4 read ports and 2 write ports and a separate memory address adder

A MIPS VLIW (2-issue) Datapath

No hazard hardware (so no load use allowed)



Let's say we wanted more functional units. What would need to change?

Code Scheduling Example

- Consider the following loop code:

```
lp:   lw     $t0,0($s1)     # $t0=array element
      addu   $t0,$t0,$s2    # add scalar in $s2
      sw     $t0,0($s1)     # store result
      addi   $s1,$s1,-4     # decrement pointer
      bne    $s1,$0,lp      # branch if $s1 != 0
```

- Must “schedule” the instructions to avoid pipeline stalls
  - Instructions in one bundle must be independent
  - Must separate load use instructions from their loads by one cycle
  - Notice that the first two instructions have a load use dependency, the next two and last two have data dependencies
  - Assume branches are perfectly predicted by the hardware

The Scheduled Code (Not Unrolled)

	ALU or branch	Data transfer	CC
lp:			1
			2
			3
			4
			5

- How many clock cycles?
- How many instructions?
- CPI? Best case?
- IPC? Best case?

Loop Unrolling

- Loop unrolling—multiple copies of the loop body are made and instructions from different iterations are scheduled together as a way to increase ILP
- Apply loop unrolling (4 times for our example) and then schedule the resulting code
  - Eliminate unnecessary loop overhead instructions
  - Schedule so as to avoid load use hazards
- During unrolling the compiler applies register renaming to eliminate all data dependencies that are not true dependencies

Unrolled Code Example

-
  ```
  lp: lw $t0,0($s1)     # $t0=array element
      lw $t1,-4($s1)    # $t1=array element
      lw $t2,-8($s1)     # $t2=array element
      lw $t3,-12($s1)   # $t3=array element
      addu $t0,$t0,$s2  # add scalar in $s2
      addu $t1,$t1,$s2  # add scalar in $s2
      addu $t2,$t2,$s2  # add scalar in $s2
      addu $t3,$t3,$s2  # add scalar in $s2
      sw $t0,0($s1)    # store result
      sw $t1,-4($s1)    # store result
      sw $t2,-8($s1)    # store result
      sw $t3,-12($s1)   # store result
      addi $s1,$s1,-16  # decrement pointer
      bne  $s1,$0,lp    # branch if $s1 != 0
  ```

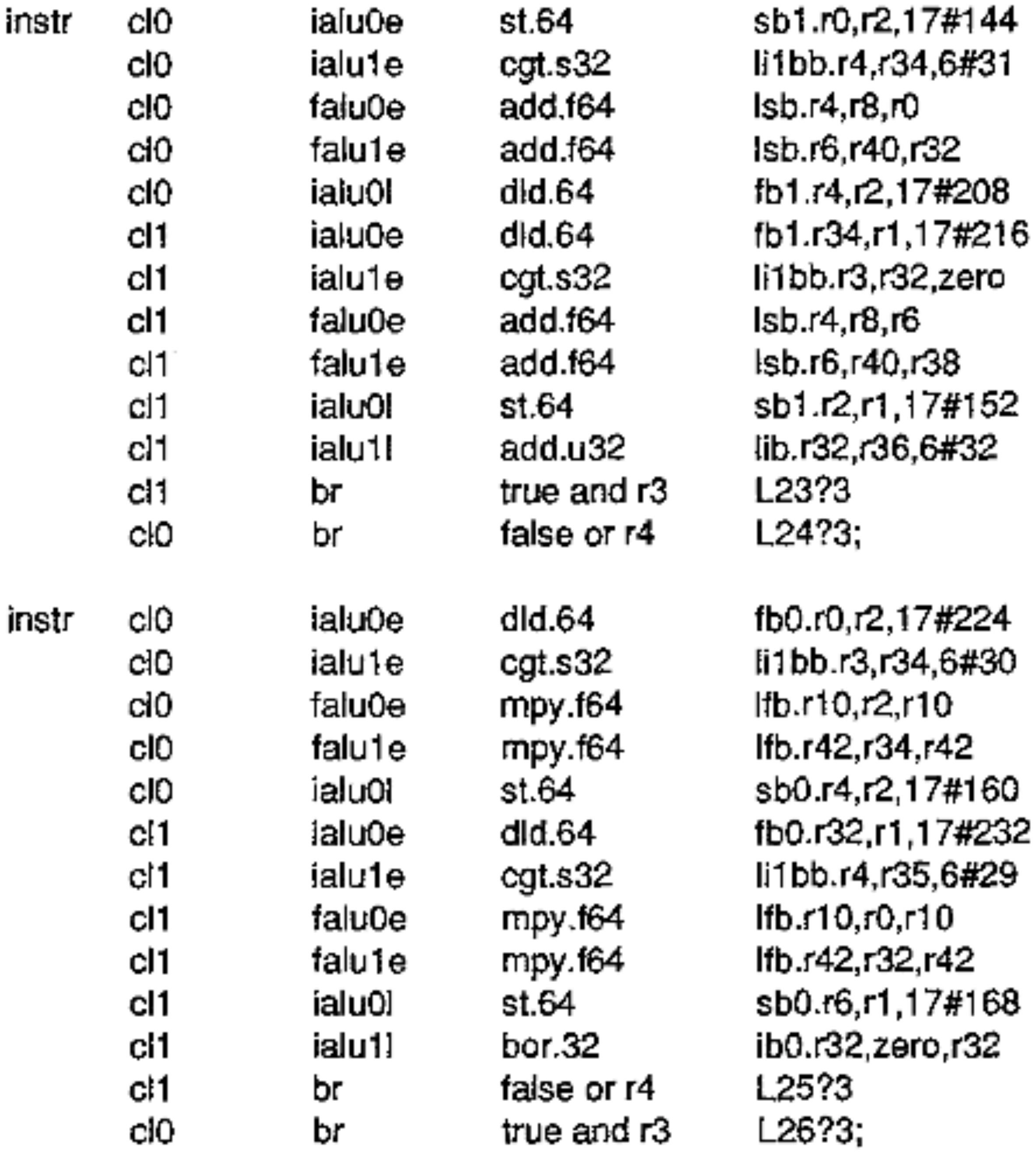
The Scheduled Code (Unrolled)

	ALU or branch	Data transfer	CC
lp:	addi $s1,$s1,-16	lw $t0,0($s1)	1
		lw $t1,12($s1)	2
	addu $t0,$t0,$s2	lw $t2,8($s1)	3
	addu $t1,$t1,$s2	lw $t3,4($s1)	4
	addu $t2,$t2,$s2	sw $t0,16($s1)	5
	addu $t3,$t3,$s2	sw $t1,12($s1)	6
		sw $t2,8($s1)	7
	bne $s1,$0,lp	sw $t3,4($s1)	8

- Eight clock cycles to execute 14 instructions for a
  - CPI of 0.57 (versus the best case of 0.5)
  - IPC of 1.8 (versus the best case of 2.0)

What does N = 14 assembly look like?

- Two instructions from a scientific benchmark (Linpack) for a MultiFlow CPU with 14 operations per instruction.

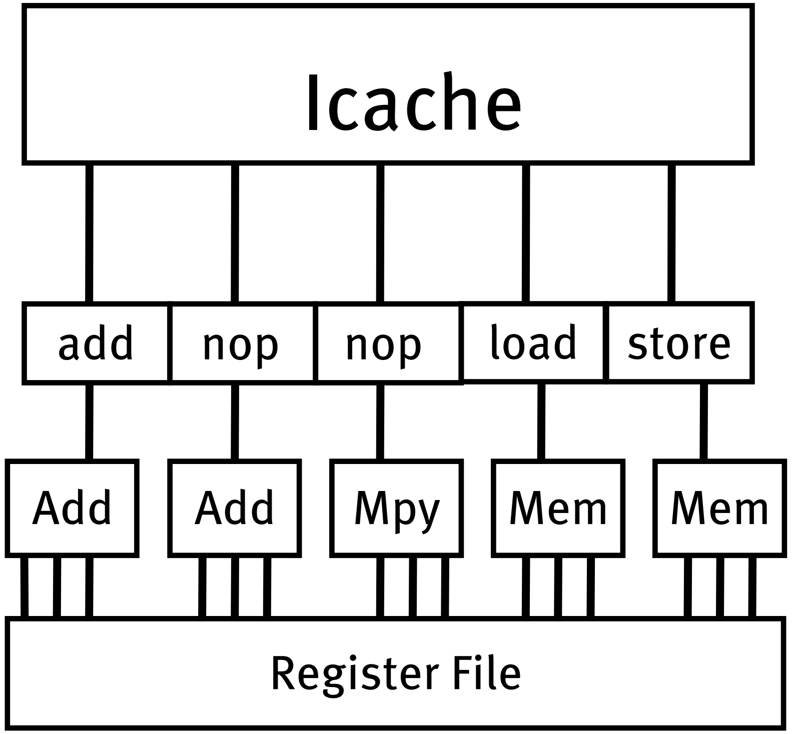


```
instr   cl0     ialu0e   st.64          sb1.r0,r2,17#144
        cl0     ialu1e   cgt.s32        li1bb.r4,r34,6#31
        cl0     falu0e   add.f64        lsb.r4,r8,r0
        cl0     falu1e   add.f64        lsb.r6,r40,r32
        cl0     ialu0l   dld.64         fb1.r4,r2,17#208
        cl1     ialu0e   dld.64         fb1.r34,r1,17#216
        cl1     ialu1e   cgt.s32        li1bb.r3,r32,zero
        cl1     falu0e   add.f64        lsb.r4,r8,r6
        cl1     falu1e   add.f64        lsb.r6,r40,r38
        cl1     ialu0l   st.64          sb1.r2,r1,17#152
        cl1     ialu1l   add.u32        lib.r32,r36,6#32
        cl1     br       true and r3    L23?3
        cl0     br       false or r4    L24?3;

instr   cl0     ialu0e   dld.64         fb0.r0,r2,17#224
        cl0     ialu1e   cgt.s32        li1bb.r3,r34,6#30
        cl0     falu0e   mpy.f64        lfb.r10,r2,r10
        cl0     falu1e   mpy.f64        lfb.r42,r34,r42
        cl0     ialu0l   st.64          sb0.r4,r2,17#160
        cl1     ialu0e   dld.64         fb0.r32,r1,17#232
        cl1     ialu1e   cgt.s32        li1bb.r4,r35,6#29
        cl1     falu0e   mpy.f64        lfb.r10,r0,r10
        cl1     falu1e   mpy.f64        lfb.r42,r32,r42
        cl1     ialu0l   st.64          sb0.r6,r1,17#168
        cl1     ialu1l   bor.32         ib0.r32,zero,r32
        cl1     br       false or r4    L25?3
        cl0     br       true and r3    L26?3;
```

Defining Attributes of VLIW

- Compiler:
  - 1. MultiOp: instruction containing multiple independent operations
  - 2. Specified number of resources of specified types
  - 3. Exposed, architectural latencies



VLIW instruction = 5 independent operations

Compiler Support for VLIW Processors

- The compiler packs groups of independent instructions into the bundle
  - Because branch prediction is not perfect, done by code re-ordering (trace scheduling)
  - We'll cover this in a future lecture
- The compiler uses loop unrolling to expose more ILP
- The compiler uses register renaming to solve name dependencies and ensures no load use hazards occur

Compiler Support for VLIW Processors

- While superscalars use dynamic prediction, VLIW’s primarily depend on the compiler for extracting ILP
  - Loop unrolling reduces the number of conditional branches
  - Predication eliminates if-the-else branch structures by replacing them with predicated instructions
    - We’ll cover this in a future lecture as well
- The compiler predicts memory bank references to help minimize memory bank conflicts

VLIW Advantages

- Advantages
  - Simpler hardware (potentially less power hungry)
  - Potentially more scalable
    - Allow more instr’s per VLIW bundle and add more FUs

VLIW Disadvantages

- Programmer/compiler complexity and longer compilation times
  - Deep pipelines and long latencies can be confusing (making peak performance elusive)
- Lock step operation, i.e., on hazard all future issues stall until hazard is resolved (hence need for predication)
- Object (binary) code incompatibility
- Needs lots of program memory bandwidth
- Code bloat
  - Noops are a waste of program memory space
  - Loop unrolling to expose more ILP uses more program memory space

Review: Multi-Issue Datapath Responsibilities

- Must handle, with a combination of hardware and software
    - Data dependencies – aka data hazards
        - True data dependencies (read after write)
            - Use data forwarding hardware
            - Use compiler scheduling
        - Storage dependence (aka name dependence)
            - Use register renaming to solve both
                - Antidependencies (write after read)
                - Output dependencies (write after write)

Review: Multi-Issue Datapath Responsibilities

- Must handle, with a combination of hardware and software
    - Procedural dependencies – aka control hazards
        - Use aggressive branch prediction (speculation)
        - Use predication
        - Future lecture

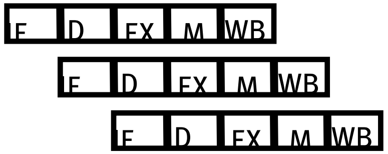
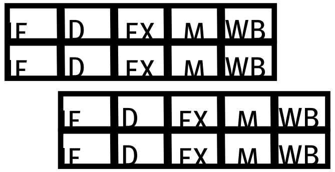
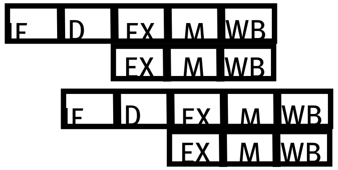
Review: Multi-Issue Datapath Responsibilities

- Must handle, with a combination of hardware and software
    - Resource conflicts—aka structural hazards
        - Use resource duplication or resource pipelining to reduce (or eliminate) resource conflicts
        - Use arbitration for result and commit buses and register file read and write ports

Review: Multiple-Issue Processor Styles

- Dynamic multiple-issue processors (aka superscalar)
  - Decisions on which instructions to execute simultaneously (in the range of 2 to 8 in 2005) are being made dynamically (at run time by the hardware)
  - E.g., IBM Power 2, Pentium 4, MIPS R10K, HP PA 8500 IBM
- Static multiple-issue processors (aka VLIW)
  - Decisions on which instructions to execute simultaneously are being made statically (at compile time by the compiler)
  - E.g., Intel Itanium and Itanium 2 for the IA-64 ISA – EPIC (Explicit Parallel Instruction Computer)
    - 128 bit "bundles" containing 3 instructions each 41 bits + 5 bit template field (specifies which FU each instr needs)
    - Five functional units (IntALU, MMedia, DMem, FPALU, Branch)
    - Extensive support for speculation and predication

CISC vs RISC vs SS vs VLIW

	CISC	RISC	Super-scalar	VLIW
Instr size				
Instr format				
Registers				
Memory reference				
Key Issues				
Instruction flow			 IF D EX M WB IF D EX M WB IF D EX M WB IF D EX M WB	

What is a basic block?

- “Experiments and experience indicated that only a factor of 2 to 3 speedup from parallelism was available within basic blocks. (A basic block of code has no jumps in except at the beginning and no jumps out except at the end.)”
  - — “Very Long Instruction Word Architectures and the ELI-512”, Joseph A. Fisher

Branches Limit ILP

- Programs average about 5 instructions between branches
    - Can’t issue instructions if you don’t know where the program is going
    - Current processors issue 4–6 operations/cycle
- Conclusion: Must exploit parallelism across multiple basic blocks

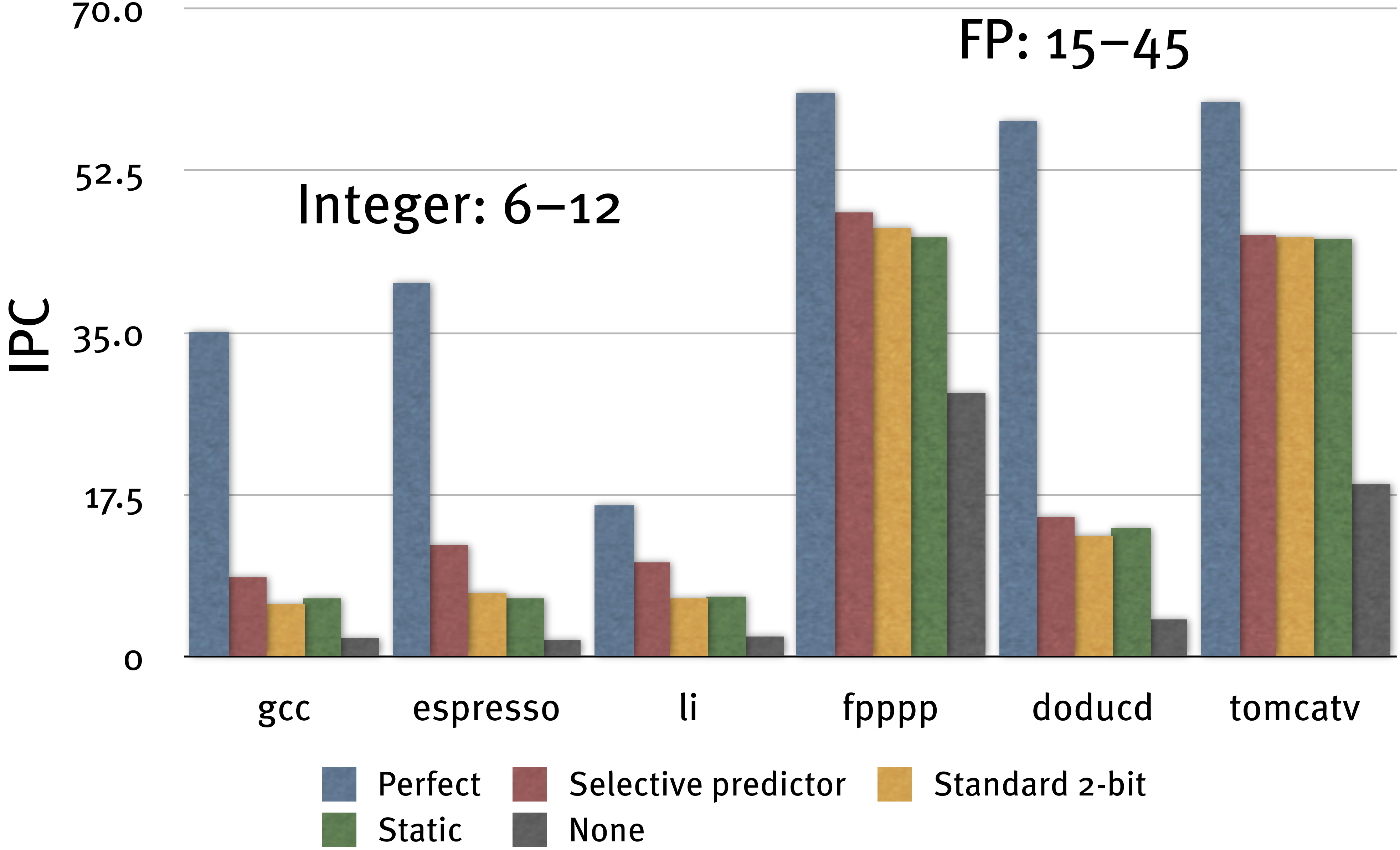
Branch Prediction Matters

- 21264:

Benchmark	Misprediction Rate	Performance Penalty
go	16.5%	40%
compress	9%	30%
gcc	7%	20%

- (From Ranganathan and Jouppi, via Dan Connors)

Branch Prediction Impact



Compiler: Static Prediction

- Predict at compile time whether branches will be taken before execution
- Schemes
    - Predict taken
        - Would be hard to squeeze into our pipeline
            - Can’t compute target until ID

Compiler: Static Prediction

- Predict at compile time whether branches will be taken before execution
- Schemes
  - Backwards taken, forwards not taken
    - Why is this a good idea?

Compiler: Static Prediction

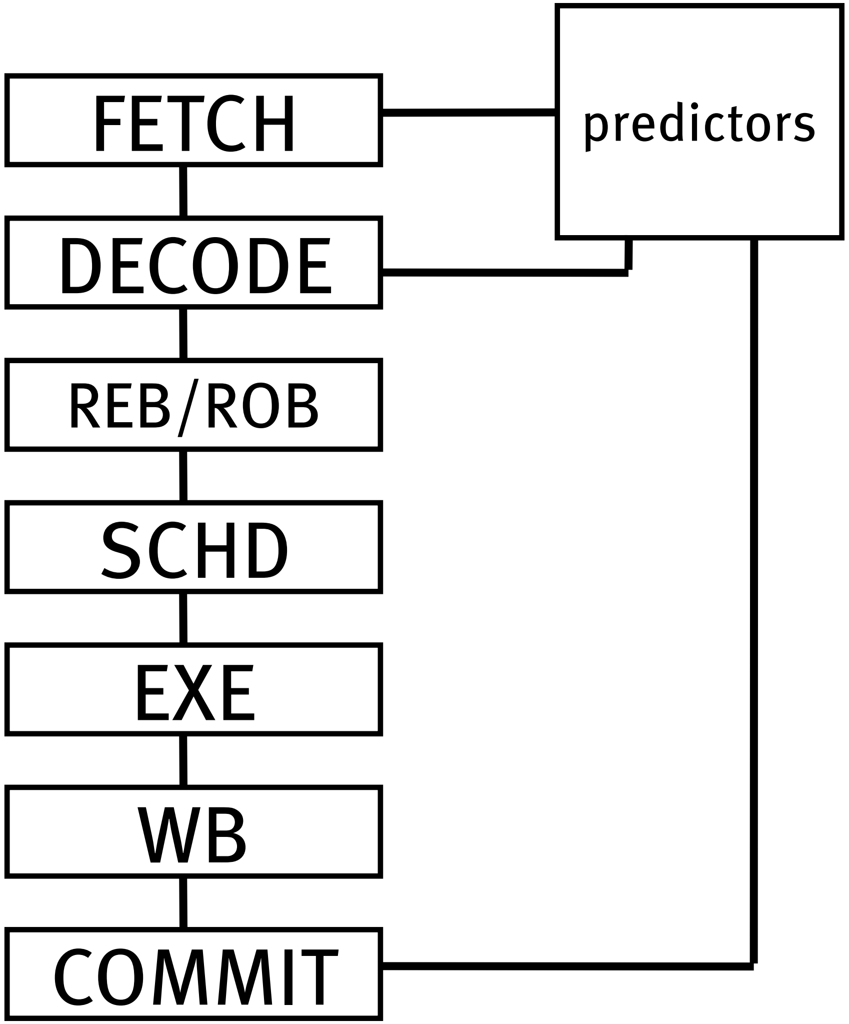
- Predict at compile time whether branches will be taken before execution
- Schemes
    - Predict taken
    - Backwards taken, forwards not taken (good performance for loops)
- No run-time adaptation: bad performance for data-dependent branches
    - if (a == 0) b =3; else b=4;

Hardware-based Dynamic Branch Prediction

- Single level (Simple counters) – predict outcome based on past branch behavior
    - FSM (Finite State Machine)
- Global Branch Correlation – track relations between branches
    - GAs
    - Gshare
- Local Correlation – predict outcome based on past branch behavior PATTERN
    - PAs
- Hybrid predictors (combination of local and global)
- Miscellaneous
    - Return Address Stack (RAS)
    - Indirect jump prediction

Mis-prediction Detections and Feedbacks

- Detections:
  - At the end of decoding
    - Target address known at decoding, and does not match
    - Flush fetch stage
  - At commit (most cases)
    - Wrong branch direction or target address does not match
    - Flush the whole pipeline
- Feedbacks:
  - Any time a mis-prediction is detected
  - At a branch's commit



1-bit “Self Correlating” Predictor

- Let’s consider a simple model. Store a bit per branch: “last time, was the branch taken or not”.
- Consider a loop of 10 iterations before exit:

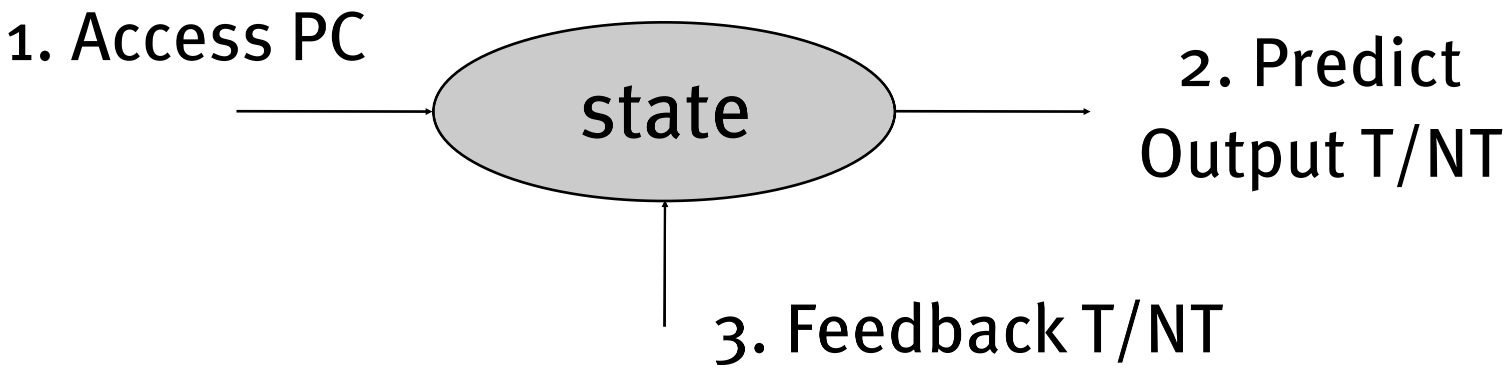
  -
    ```
    for (…)
      for (i=0; i<10; i++)
        a[i] = a[i] * 2.0;
    ```
- What’s the accuracy of this predictor?

Dynamic Branch Prediction

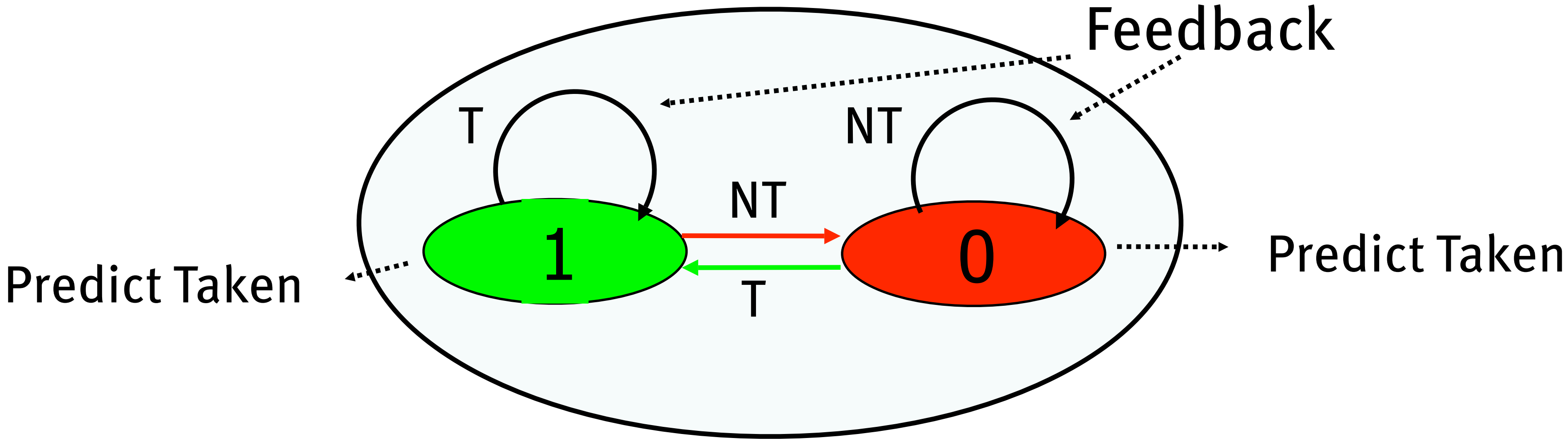
- Performance = *f*(accuracy, cost of misprediction)
- Branch History Table: Lower bits of PC address index table of 1-bit values
  - Says whether or not branch taken last time
  - No address check
- Problem: in a loop, 1-bit BHT will cause two mispredictions (avg is 9 iterations before exit):
  - End of loop case, when it exits instead of looping as before
  - First time through loop on next time through code, when it predicts exit instead of looping

Predictor for a Single Branch

General Form

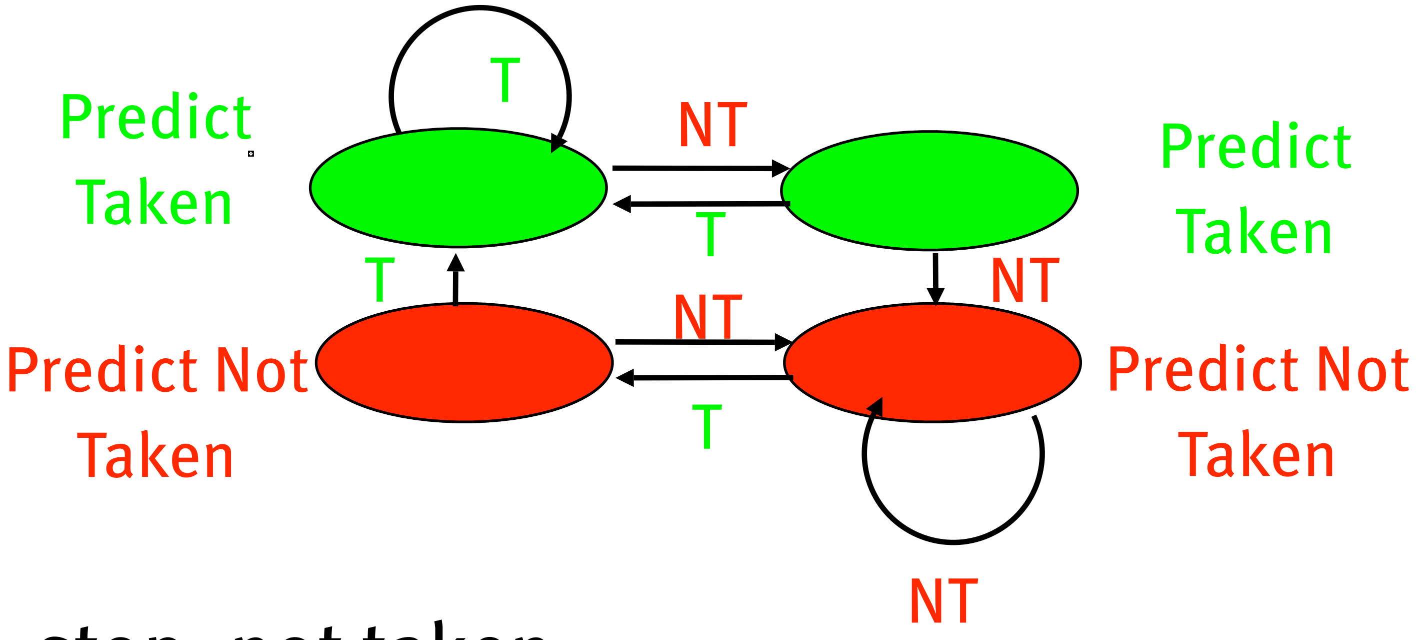


1-bit prediction



Dynamic Branch Prediction (Jim Smith, 1981)

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

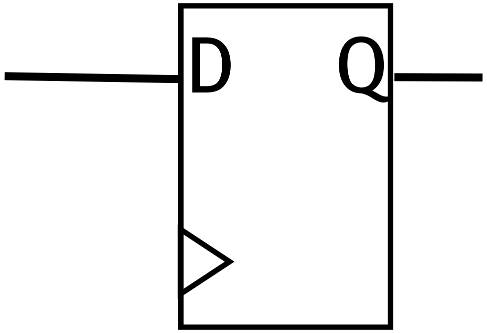


- Red: stop, not taken
- Green: go, taken
- Adds hysteresis to decision making process

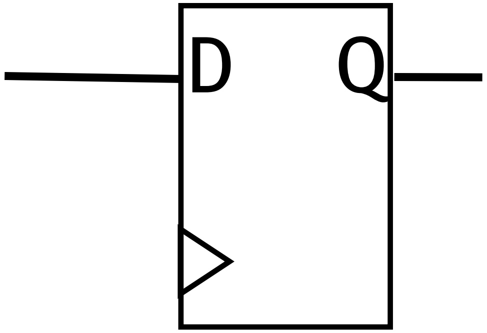
Simple (“2-bit”) Branch History Table Entry

Prediction for next branch.
(1 = take, 0 = not take)
Initialize to 0.

Was last prediction correct?
(1 = yes, 0 = no)
Initialize to 1.



After we “check” prediction ...

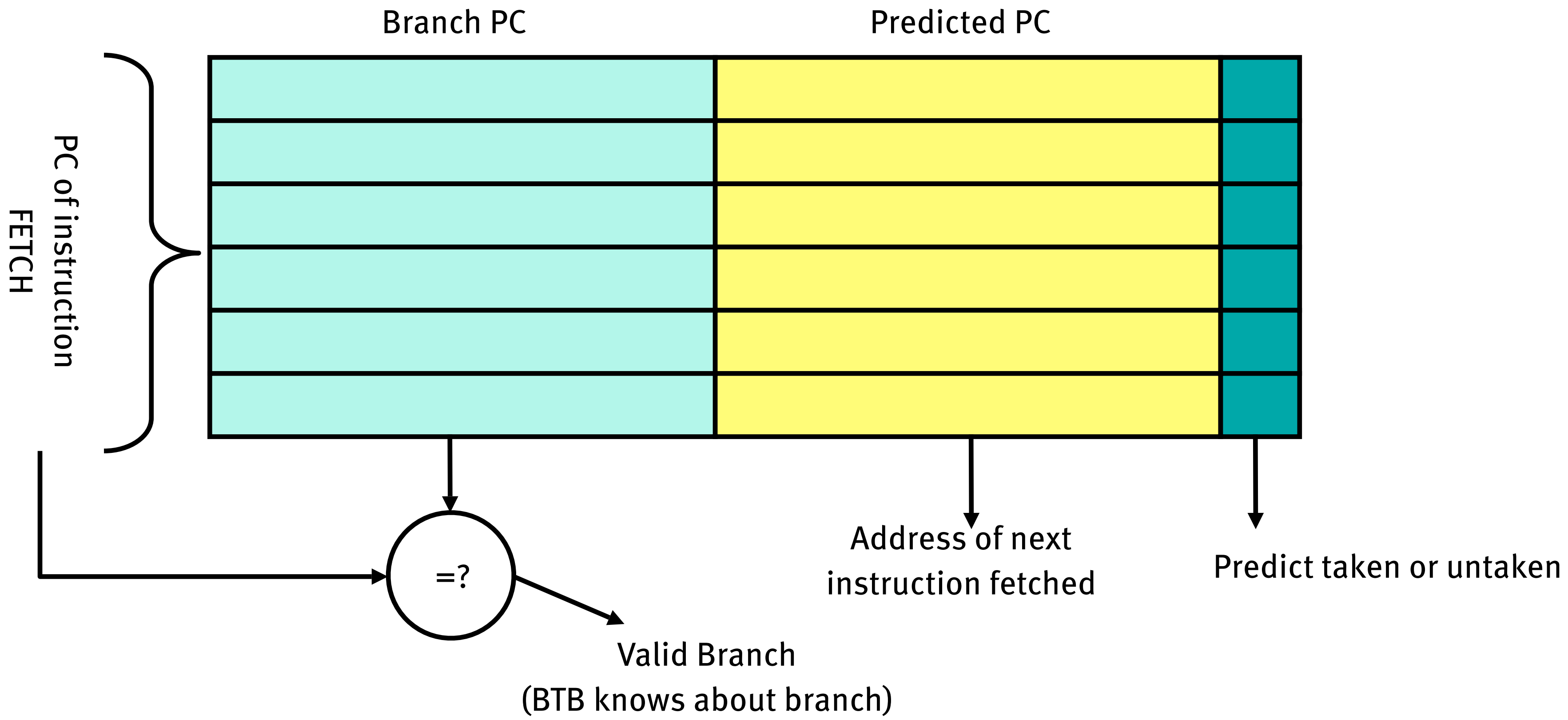


Flip bit if prediction is not correct and “last predict correct” bit is 0.

Set to 1 if prediction bit was correct.
Set to 0 if prediction bit was incorrect.
Set to 1 if prediction bit flips.

Branch prediction hardware

- Branch Target Buffer (BTB): Address of branch index to get prediction AND branch address (if taken)
  - Note: must check for branch match now, since can't use wrong branch address



Some Interesting Patterns

- Format: Not-taken (N) (0), Taken (T)(1)
- TTTTTTTTTT
    - 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 … : Should give perfect prediction
- NNTTNNTTNNTT
    - 0 0 1 1 1 1 0 1 0 0 1 1 0 0 1 1 1 1 1 0 0 0 1 0 … : Will mispredict 1/2 of the time
- N*N[TNTN]
    - 0 0 0 0 0 0 0 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 … : Should alternate incorrectly
- N*T[TNTN]
    - 0 0 0 0 0 0 0 1 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 … : Should alternate incorrectly

Pentium 4 Branch Prediction

- Critical to Performance
    - 20 cycle penalty for misprediction
- Branch Target Buffer
    - 2048 entries
    - 12 bits of history
    - Adaptive algorithm
        - Can recognize repeated patterns, e.g., alternating taken–not taken
- Handling BTB misses
    - Detect in cycle 6
    - Predict taken for negative offset, not taken for positive (why?)

Branch Prediction Summary

- Consider for each branch prediction
  - Hardware cost
  - Prediction accuracy
  - Warm-up time
  - Correlation
  - Interference
  - Time to generate prediction
- Application behavior determines number of branches
  - More control intensive program…more opportunity to mispredict
- What if a compiler/architecture could eliminate branches?