Pipelining, Branch Prediction, Trends

10.1-10.4

Topics

- 10.1 Quantitative Analyses of Program Execution
- 10.2 From CISC to RISC
- 10.3 Pipelining the Datapath

  Branch Prediction, Delay Slots
- 10.4 Overlapping Register Windows

Quantitative Analysis

- CISC approach
  - Belief that the **semantic gap** should be shortened
    - The gap between machine-level instructions and high-level language statements
    - Examples
      - VAX Sort instruction
      - IBM 360 MVC instruction (move character)
        - » checked if strings overlapped but this was rare
        - » Could be faster if assumed strings did not overlap
  - Sounds reasonable, but is this assumption correct?
    - Quantitatively measure programs to see

Quantitative Analysis

- Work by Knuth/Hennessy/Patterson
  - Confirmed that most complex instruction and addressing modes largely unused by compilers
    - Difficult for compiler to take advantage of these modes
    - Used by assembly programmers
    - But most programmers used a high-level language

Frequency of Instructions

- Frequency of occurrence of instruction types for a variety of languages/benchmark programs

Statement	Average Percent of Time
Assignment	47
If	23
Call	15
Loop	6
Goto	3
Other	7

Arithmetic and other “powerful” instructions only 7%

Complexity of Assignments/Procedures

	Percentage of Number of Terms in Assignments	Percentage of Number of Locals in Procedures	Percentage of Number of Parameters in Procedure Calls
0	–	22	41
1	80	17	19
2	15	20	15
3	3	14	9
4	2	8	7
≥ 5	0	20	8

80% of assignments involve one term;
80% of procedures could be handled supported 4 locals

Quantitative Analysis Results

- Bulk of computer programs are very simple at the instruction level
- Little payoff in making complex instructions
- RISC idea
  - Make the common case go fast; by making simple instructions fast, most programs will go fast
  - Load/Store architecture
    - Only way to communicate with memory is via Load/Store from register file. E.g., an ADD can't have an operand be a memory address
    - Simplifies communications and pipelining (coming up)
    - Means we need a lot of registers
      - Tradeoff: simpler CPU means there is space to put more registers on the chip

Speedup and Efficiency

- **Speedup *S* is the ratio of the time needed to execute a program without an enhancement to the time required with an enhancement.**

$$S = \frac{T_{wo}}{T_w} \qquad S = \frac{T_{wo} - T_w}{T_w} \times 100$$

- **Time *T* is computed as the instruction count *IC* times the number of cycles per instruction *CPI* times the cycle time τ.**

$$T = IC \times CPI \times \tau$$

- **Substituting *T* into the speedup percentage calculation above yields:**

$$S = \frac{IC_{wo} \times CPI_{wo} \times \tau_{wo} - IC_w \times CPI_w \times \tau_w}{IC_w \times CPI_w \times \tau_w} \times 100$$

Speedup Example

- ***Example:*** **Estimate the speedup obtained by replacing a CPU having an average CPI of 5 with another CPU having an average CPI of 3.5, with the clock period increased from 100 ns to 120 ns.**
- **The previous equation becomes:**

$$S = \frac{5 \times 100 - 3.5 \times 120}{3.5 \times 120} \times 100 = 19\%$$

Using benchmarks, we can estimate the impact of a new architecture before we actually build it!

Pipelining

- The RISC approach lends itself well to a technique that can greatly improve processor performance called **pipelining**
- We will see why this is more difficult with CISC instructions as we continue…

Instruction Prefetch

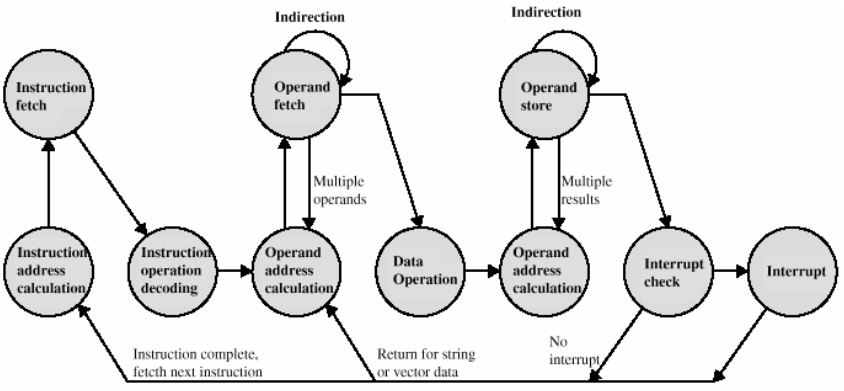
- Simple version of Pipelining – treating the instruction cycle like an assembly line

- Fetch accessing main memory
- Execution usually does not access main memory
- Can fetch next instruction during execution of current instruction
- Called instruction prefetch

Improved Performance

- But not doubled:
  - Fetch usually shorter than execution
    - Prefetch more than one instruction?
  - Any jump or branch means that prefetched instructions are not the required instructions
- Add more stages to improve performance
  - But more stages can also hurt performance…

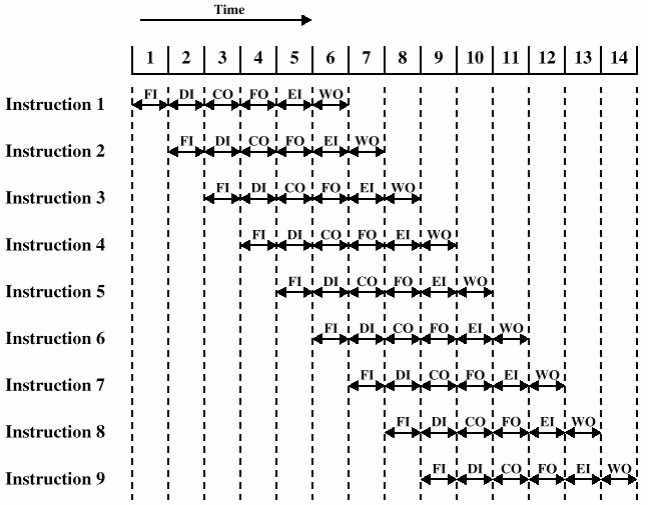
Instruction Cycle State Diagram



Pipelining

- Consider the following decomposition for processing the instructions
  - Fetch instruction – Read into a buffer
  - Decode instruction – Determine opcode, operands
  - Calculate operands (i.e. EAs) – Indirect, Register indirect, etc.
  - Fetch operands – Fetch operands from memory
  - Execute instructions - Execute
  - Write result – Store result if applicable
- Overlap these operations to make a 6 stage pipeline
- The textbook uses a 5 stage pipeline (Fetch/Decode/Operand Fetch/Execute/Write Back)

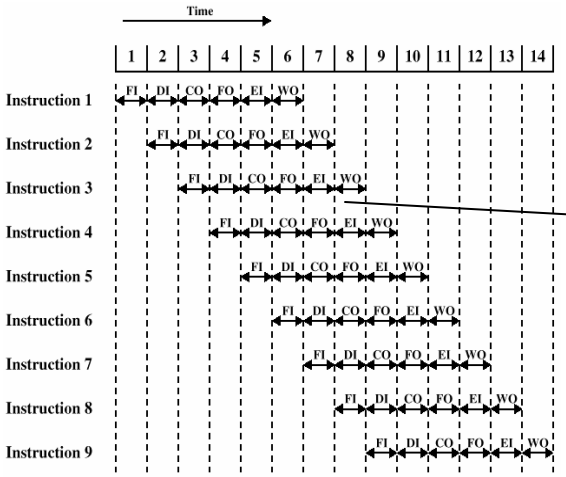
Timing of Pipeline



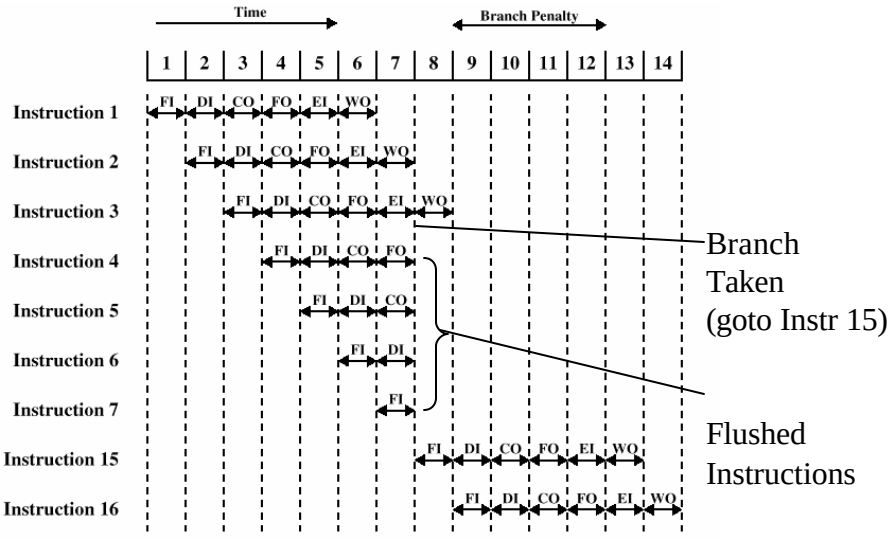
Pipeline

- In the previous slide, we completed 9 instructions in the time it would take to sequentially complete two instructions!
- Assumptions for simplicity
  - Stages are of equal duration
- Things that can mess up the pipeline
  - Structural Hazards – Can all stages can be executed in parallel?
    - What stages might conflict? E.g. access memory
  - Data Hazards – One instruction might depend on result of a previous instruction
    - E.g. INC R1 ADD R2,R1
  - Control Hazards - Conditional branches break the pipeline
    - Stuff we fetched in advance is useless if we take the branch

Branch Not Taken



Branch in a Pipeline – Flushed Pipeline



Dealing with Branches

- Multiple Streams
- Prefetch Branch Target
- Loop buffer
- Branch prediction
- Delayed branching

Multiple Streams

- Have two pipelines
- Prefetch each branch into a separate pipeline
- Use appropriate pipeline

- Leads to bus & register contention
- Still a penalty since it takes some cycles to figure out the branch target and start fetching instructions from there
- Multiple branches lead to further pipelines being needed
  - Would need more than two pipelines then
- More expensive circuitry

Prefetch Branch Target

- Target of branch is prefetched in addition to instructions following branch
  - Prefetch here means getting these instructions and storing them in the cache
- Keep target until branch is executed
- Used by IBM 360/91

Loop Buffer

- Very fast memory
- Maintained by fetch stage of pipeline
- Remembers the last N instructions
- Check buffer before fetching from memory
- Very good for small loops or jumps
- c.f. cache
- Used by CRAY-1

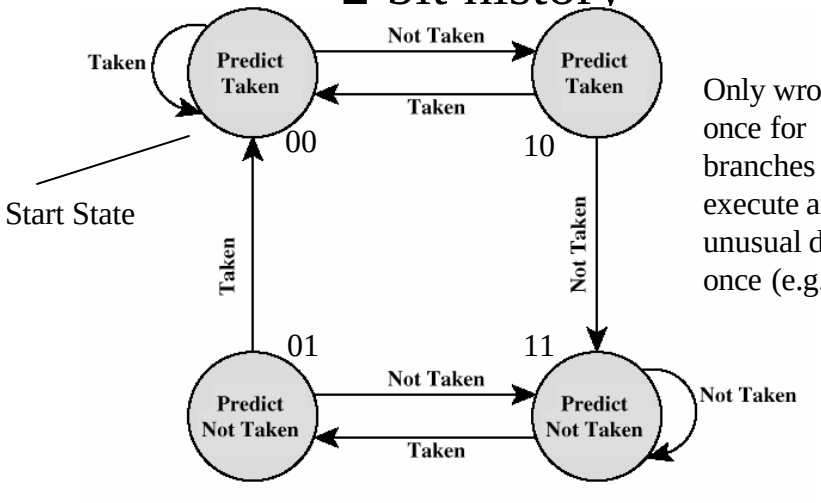
Branch Prediction (1)

- Predict never taken
  - Assume that jump will not happen
  - Always fetch next instruction
  - 68020 & VAX 11/780
  - VAX will not prefetch after branch if a page fault would result (O/S v CPU design)
- Predict always taken
  - Assume that jump will happen
  - Always fetch target instruction
  - Studies indicate branches are taken around 60% of the time in most programs

Branch Prediction (2)

- Predict by Opcode
  - Some types of branch instructions are more likely to result in a jump than others (e.g. LOOP vs. JUMP)
  - Can get up to 75% success
- Taken/Not taken switch – 1 bit branch predictor
  - Based on previous history
    - If a branch was taken last time, predict it will be taken again
    - If a branch was not taken last time, predict it will not be taken again
  - Good for loops
  - Could use a single bit to indicate history of the previous result
  - Need to somehow store this bit with each branch instruction
  - Could use more bits to remember a more elaborate history

Branch Prediction State Diagram – 2 bit history



Only wrong once for branches that execute an unusual direction once (e.g. loop)

Branch Prediction

- State not stored in memory, but in a special high-speed history table

Branch Instruction Address	Target Address	State
FF0103	FF1104	11
…		

Dealing with Branches – RISC Approach

- Delayed Branch – used with RISC machines
  - Requires some clever rearrangement of instructions
  - Burden on programmers but can increase performance
  - Most RISC machines: Doesn't flush the pipeline in case of a branch
  - Called the Delayed Branch
    - This means if we take a branch, we'll still continue to execute whatever is currently in the pipeline, at a minimum the next instruction
    - Benefit: Simplifies the hardware quite a bit
    - But we need to make sure it is safe to execute the remaining instructions in the pipeline
    - Simple solution to get same behavior as a flushed pipeline: Insert NOP – No Operation – instructions after a branch
      - Called the Delay Slot

RISC Pipeline with Delay Slot

Using a Five Stage pipeline:
IF = Fetch, ID = Decode, EX = Execute
MEM = Memory access, WB = Write back register values

In this example: CPU knows if branches are to be taken after the ID stage (implications if not known until after the EX stage?)

Untaken branch instruction	IF	ID	EX	MEM	WB				
Branch delay instruction (i + 1)		IF	ID	EX	MEM	WB			
Instruction i + 2			IF	ID	EX	MEM	WB		
Instruction i + 3				IF	ID	EX	MEM	WB	
Instruction i + 4					IF	ID	EX	MEM	WB

Taken branch instruction	IF	ID	EX	MEM	WB				
Branch delay instruction (i + 1)		IF	ID	EX	MEM	WB			
Branch target			IF	ID	EX	MEM	WB		
Branch target + 1				IF	ID	EX	MEM	WB	
Branch target + 2					IF	ID	EX	MEM	WB

Normal vs. Delayed Branch

Address	Normal	Delayed
100	LOAD X,A	LOAD X,A
101	ADD 1,A	ADD 1,A
102	JUMP 105	JUMP 106
103	ADD A,B	NOOP
104	SUB C,B	ADD A,B
105	STORE A,Z	SUB C,B
106		STORE A,Z

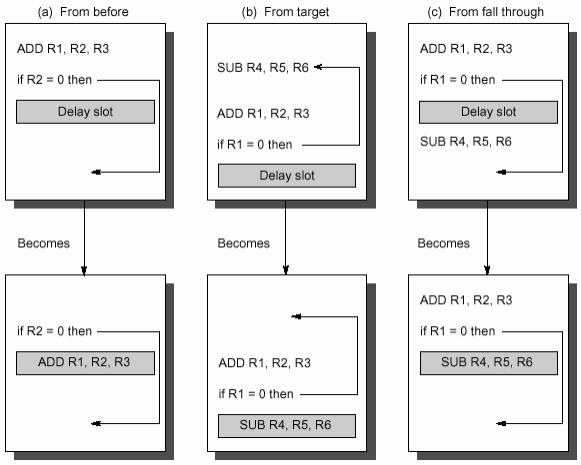
One delay slot - Next instruction is always in the pipeline. "Normal" path contains an implicit "NOP" instruction as the pipeline gets flushed. Delayed branch requires explicit NOP instruction placed in the code!

Optimized Delayed Branch

But we can optimize this code by rearrangement! Notice we always Add 1 to A so we can use this instruction to fill the delay slot

Address	Normal	Delayed	Optimized
100	LOAD X,A	LOAD X,A	LOAD X,A
101	ADD 1,A	ADD 1,A	JUMP 105
102	JUMP 105	JUMP 106	ADD 1,A
103	ADD A,B	NOOP	ADD A,B
104	SUB C,B	ADD A,B	SUB C,B
105	STORE A,Z	SUB C,B	STORE A,Z
106		STORE A,Z	

Example: Delay Slot Scheduling



B) and C) execute code that may or may not be used, but better than a NOP

Form of branch prediction – compiler predicts based on context

Delay Slot Effectiveness

- On benchmarks
  - Delay slot allowed branch hazards to be hidden 70% of the time
  - About 20% of delay slots filled with NOPs
  - Delay slots we can't easily fill: when target is another branch
- Philosophically, delay slots good?
  - No longer hides the pipeline implementation from the programmers (although it will if through a compiler)
  - Does allow for compiler optimizations, other schemes don't
  - Not very effective with modern machines that have deep pipelines, too difficult to fill multiple delay slots

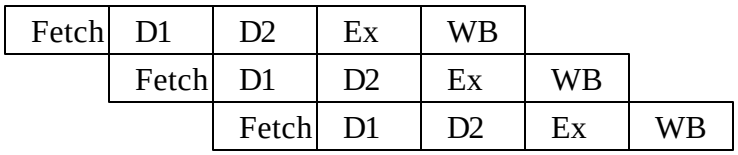
Other Pipelining Overhead

- Each stage of the pipeline has overhead in moving data from buffer to buffer for one stage to another. This can lengthen the total time it takes to execute a single instruction!
- The amount of control logic required to handle memory and register dependencies and to optimize the use of the pipeline increases enormously with the number of stages. This can lead to a case where the logic between stages is more complex than the actual stages being controlled.
- Need balance, careful design to optimize pipelining

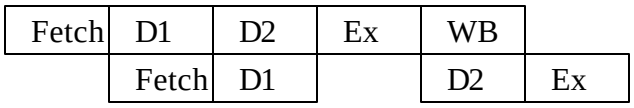
Pipelining on the 486/Pentium

- 486 has a 5-stage pipeline
  - Fetch
    - Instructions can have variable length and can make this stage out of sync with other stages. This stage actually fetches about 5 instructions with a 16 byte load
  - Decode1
    - Decode opcode, addressing modes – can be determined from the first 3 bytes
  - Decode2
    - Expand opcode into control signals and more complex addressing modes
  - Execute
  - Write Back
    - Store value back to memory or to register file

486 Pipelining Examples



MOV R1, M
MOV R1, R2
MOV M, R1

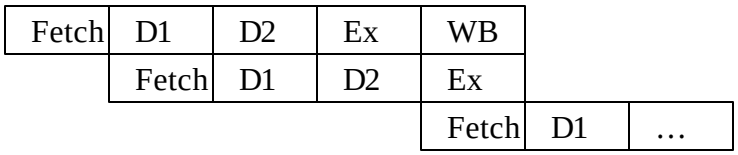


MOV R2, M
MOV R1, (R2)

Need R2 written back to use as addr for second instruction in stage D2

Normally this data is not available until after the WB stage, but bypass circuitry allows us to send the proper data directly to EX of the next stage (this is called **forwarding**)

486 Pipelining Examples



CMP R1,Imm
JCC Target
Target

Target address known after D2 phase
Runs a speculative Fetch on the target during EX hoping we will execute it (predict taken)

Also fetches next consecutive instruction if branch not taken

Pentium II/IV Pipelining

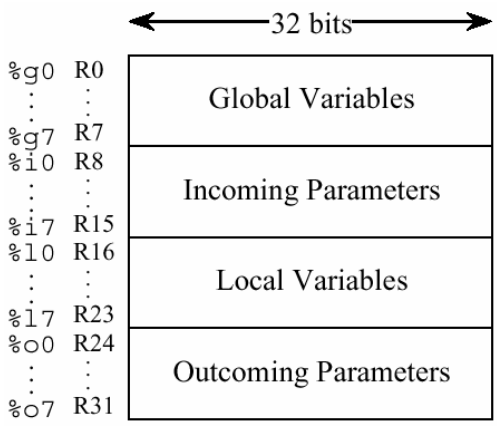
- Pentium II
  - 12 pipeline stages
  - Dynamic execution incorporates the concepts of out of order and speculative execution
  - Two-level, adaptive-training, branch prediction mechanism
- Pentium IV
  - 20 stage pipeline
  - Combines different branch prediction mechanisms to keep the pipeline full

Register Windows

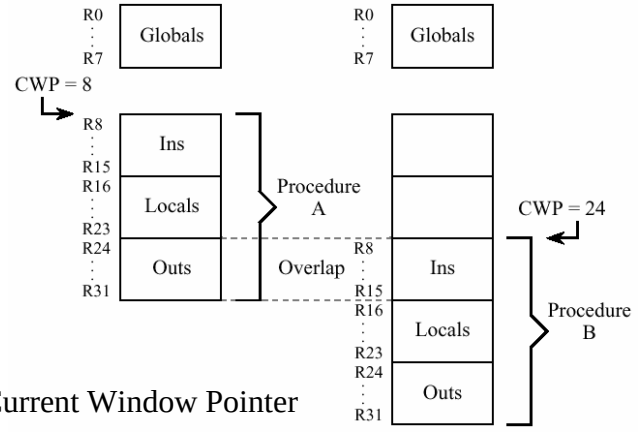
- This technique was motivated by quantitative analysis of how procedures pass parameters back and forth
- Normal parameter passing: Uses the stack
  - But this is slow
  - Would be faster to use registers
  - Benchmarks indicate that
    - Most procedures only pass a few parameters
    - A nesting depth of more than 5 is rare

User View of Registers

- Used on SPARC



Overlap Register Windows



CWP = Current Window Pointer

Register Windows

- Parameters are “passed” by simply updating the window pointer
  - All parameter access in registers, very fast
  - In the rare event we exceed the number of registers available, can use main memory for overflow