William Stallings Computer Organization and Architecture

Chapter 11.4, 11.5

Pipelining, Interrupts

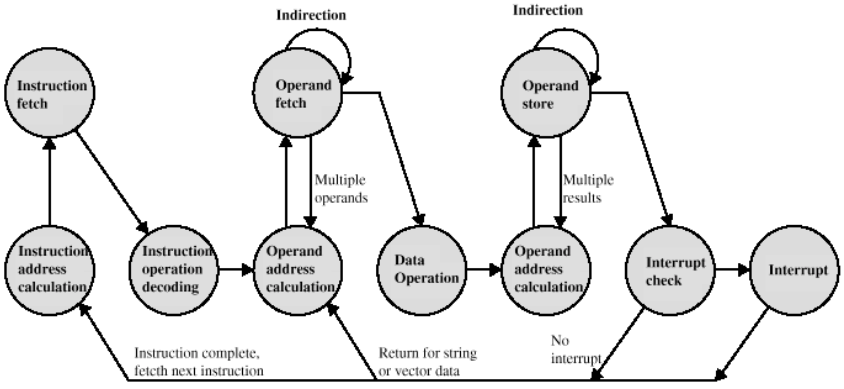
Prefetch

- 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...

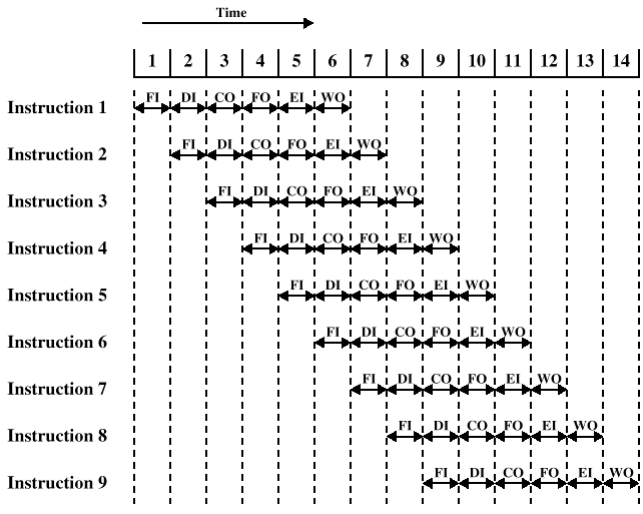
Instruction Cycle State Diagram



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

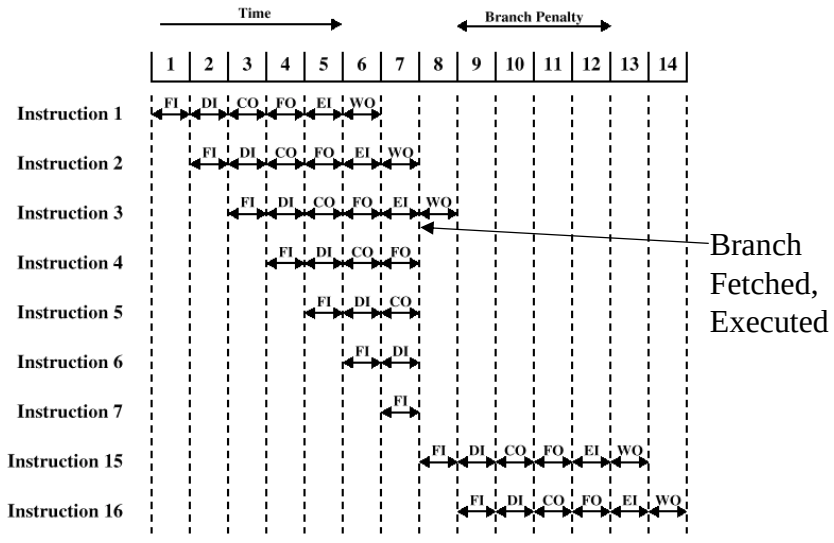
Timing of Pipeline



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 in a 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

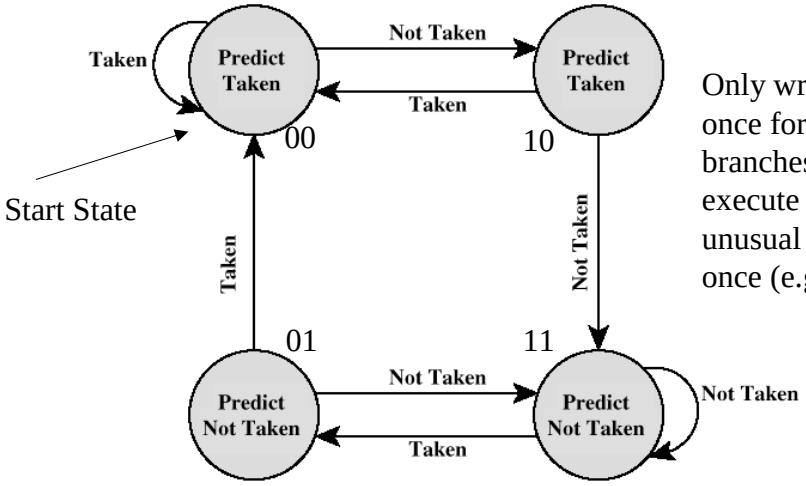
Branch Prediction (1)

- 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

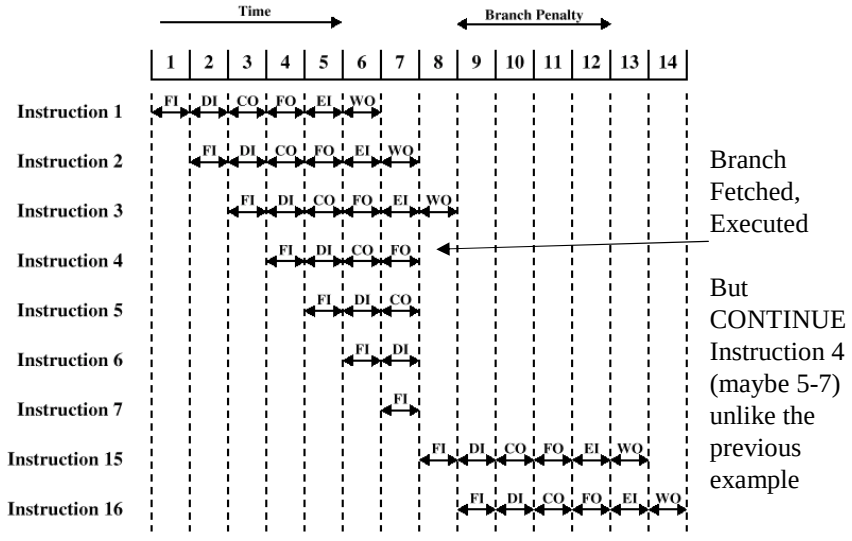
- State not stored in memory, but in a special high-speed history table

Branch Instruction Address	Target Address	State
FF0103	FF1104	11
…		

Dealing with Branches

- 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



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

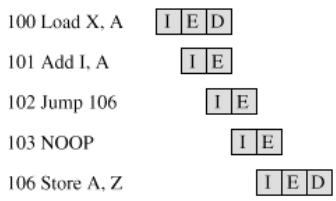
One delay slot - Next instruction is always in the pipeline. "Normal" path contains an implicit "NOP" instruction as the pipeline gets flushed. Delayed branch requires explicit NOP instruction placed in the code!

Optimized Delayed Branch

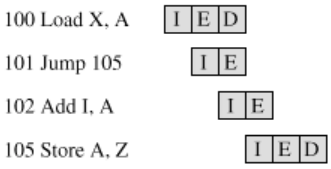
But we can optimize this code by rearrangement! Notice we always Add 1 to A so we can use this instruction to fill the delay slot

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	

Use of Delayed Branch



(a) Inserted NOOP



(b) Reversed instructions

I = Instruction Fetch
E = Instruction Execute
D = Memory Access

Both cases: Note no pipeline delays
Can sometimes be hard to optimize and fill the delay slot

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!

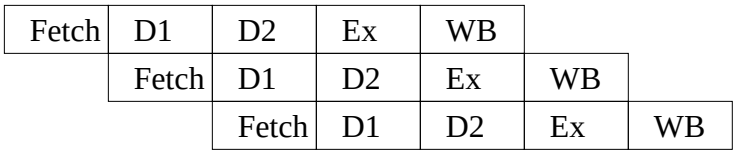
⌘ The amount of control logic required to handle memory and register dependencies and to optimize the use of the pipeline increases enormously with the number of stages. This can lead to a case where the logic between stages is more complex than the actual stages being controlled.

⌘ Need balance, careful design to optimize pipelining

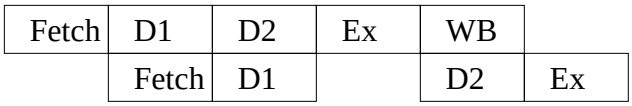
Pipelining on the 486/Pentium

- 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

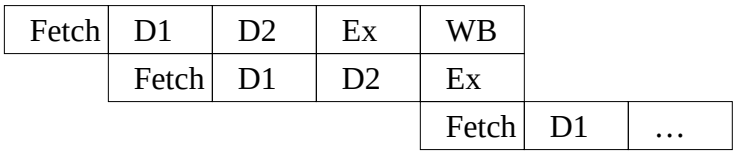


MOV R2, M
MOV R1, (R2)

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

Bypass circuitry allows us to read this value in the same stage

486 Pipelining Examples



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

Interrupt Processing on the x86

- Interrupts are primarily to support the OS – it allows a program to be suspended and later resumed (e.g. for printing, I/O, etc.)
- Interrupts – Hardware driven
  - Maskable interrupts
    - INTR pin, recognized only if interrupt enable set
  - Nonmaskable interrupts
    - NMI pin, always recognized
- Exceptions – Software driven
  - Processor-detected
    - Floating point exception
  - Programmed exceptions
    - INT, BOUND

Interrupt Vectors

- Interrupt processing refers to an **interrupt vector table**
  - 256 32-bit entries
  - Each entry is a full address for the interrupt vector handler, code that processes the interrupt
    - E.g. Vector 0 = Divide by zero, Vector 12 = Stack Exceeded
- Interrupt Handling - proceeds just like a procedure CALL
    - Push SS, SP to stack
    - Push FLAGS onto stack
    - Clear IF and TF flags to disable INTR interrupts
    - Push CS, IP onto stack
    - Possibly push error code onto stack (for interrupt handler to process)
    - Fetch interrupt vector contents, load into CS and IP
    - Upon an IRET instruction, pop values off stack, resume at old CS and IP