The Story so far:
- Instruction Set Architectures
- Performance issues
- ALU
- Simple Cycle CPU
- Multicycle CPU: datapath, control
- Microprogramming
- Exceptions
  - Pipelining
  - Basic datapath
  - Control for pipelining
  - Structural hazards: memory
  - Data hazards: forwarding, stalling
  - Branching hazards: prediction, exceptions
  - Out of order execution, speculative execution
  - Superscalar machines etc.

CPU

Pipelining

Pipelining Lessons
- Pipelining doesn’t help latency of single task, it helps throughout of entire workload
- Multiple tasks operating simultaneously using different resources
- Potential speedup = Number pipe stages
- Pipeline rate limited by slowest pipeline stage
- Unbalanced lengths of pipe stages reduces speedup
- Time to “fill” pipeline and time to “drain” it reduces speedup
- Stall for Dependences

Single Cycle CPU

Multicycle CPU
The Multicycle Processor

The Five Stages of Load

- Fetch: Instruction Fetch
- Reg/Dec: Registers Fetch and Instruction Decode
- Exec: Calculate the memory address
- Mem: Read the data from the Data Memory
- Wr: Write the data back to the register file

Conventional Pipelined Execution Representation

- Suppose we execute 100 instructions, CPI=4.6, 45ns vs. 10ns cycle time.
- Single Cycle Machine: 45 ns/cycle x 1 CPI x 100 inst = 4500 ns
- Multicycle Machine: 10 ns/cycle x 4.6 CPI (due to inst mix) x 100 inst = 4600 ns
- Ideal pipelined machine: 10 ns/cycle x (? CPI x 100 inst + 4 cycle drain) = 1040 ns

Pipelining

- Improve performance by increasing instruction throughput

Ideal speedup is number of stages in the pipeline. Do we achieve this?
Basic Idea

- What do we need to add to actually split the datapath into stages?

Graphically Representing Pipelines

Can help with answering questions like:
- how many cycles does it take to execute this code?
- what is the ALU doing during cycle 4?
- use this representation to help understand datapaths

Pipelined execution

Mixed Instructions in Pipeline

Principles of pipelining

- All instructions that share a pipeline must have the same stages in the same order:
  - therefore, add does nothing during Mem stage
  - see does nothing during WB stage
- All intermediate values must be latched each cycle.
- There is no functional block reuse
  - So, like the single cycle design, we now need two adders + one ALU &

Pipelined Datapath
lw $12, 1000($4) add $10, $1, $2

lw $12, 1000($4) add $10, $1, $2

lw $12, 1000($4) add $10, $1, $2

lw $12, 1000($4) add $10, $1, $2

lw $12, 1000($4) add $10, $1, $2

lw $12, 1000($4) add $10, $1, $2
What about control?

- can't use microprogram
- FSM not really appropriate
- Combinational Logic!
  - signals generated once, but follow instruction through the pipeline

Pipelined system with control logic

Can pipeline get us into trouble?

- Yes: Pipeline Hazards:
  - structural hazards: attempt to use the same resource two different ways at the same time
  - data hazards: attempt to use item before it is ready
    - instruction depends on result of prior instruction still in the pipeline
  - control hazards: attempt to make a decision before condition is evaluated
  - branch instructions
- Can always resolve hazards by waiting
  - Worst case the machine behaves like a multi-cycle machine!
  - pipeline control must detect the hazard
  - take action (or delay action) to resolve hazards

Single Memory is a Structural Hazard

Pipelined execution: mixed instructions?

- Remember mixed instructions?
Data Hazards

- Suppose initially register 1 holds the number 2.
  - $S1 = 20$
  - $S2 = 22$
  - $S3 = 6$
  - $S7 = 14$
  - $S8 = 16$

- What happens when:
  add $S7, S10, S11$ - this should add 20 + 22, putting result 42 into r7
  lw $S2, 50 (S3)$ - this should load it from memory location 42 + 50 = 92
  sub $S11, S8, S7$ - this should subtract 14 from that just loaded value.

Data Hazards

- When a result is needed in the pipeline before it is available, a "data hazard" occurs.

Data Hazards on r1:

- Dependencies backwards in time are hazards
Data Hazards

- In Software
  - inserting independent instructions
- In Hardware
  - inserting bubbles (stalling the pipeline)
  - data forwarding

Data Hazards are caused by instruction dependences. For example, the `add` is data-dependent on the `subtract`:

`sub $5, $4, #45`
`add $8, $5, $2`

Handling Data Hazards: Software

Assume a standard 5-stage pipeline:

- How many data hazards in this piece of code?
- How many "no-ops" did you need?
- What if you are allowed to execute out-of-order?

Handling Data Hazards: Hardware: Bubbles

- To insure proper pipeline execution in light of register dependences, we must:
  - Detect the hazard
  - Stall the pipeline
    - prevent the IF and ID stages from making progress
      - the ID stage because we can't go on until the dependent instruction completes correctly
      - the IF stage because we do not want to lose any instructions.
    - insert "no-ops" into later stages

Handling Data Hazards: Hardware: Pipeline Stalls

- First half-cycle of cycle 5: register 2 loaded
- Second half-cycle: new value is read into pipeline state
How to stall a pipeline in two quick steps!

- Prevent the IF and ID stages from proceeding
  - don't write the PC (PCWrite = 0)
  - don't rewrite IF/ID register (IF/IDWrite = 0)
- Insert "nops"
  - set all control signals propagating to EX/MEM/WB to zero

Handling Data Hazards: Hardware: Pipeline Stalls

Forwarding (just shown) handles two types of data hazards
- EX hazard
- MEM hazard
- We've already handled the third type (WB) hazard by using a transparent reg file
  - if the register file is asked to read and write the same register in the same cycle, the reg file allows the write data to be forwarded to the output.

Data Hazard Solution:
- "Forward" result from one stage to another
Data Hazard Solution: With Forwarding

- Solve this using forwarding?

Data Hazard Solution: What about this stream?

- Still need bubbles ???
- Loads are always the problem?

Forwarding (or Bypassing):

• Dependencies backwards in time are hazards

Data Hazards: Compiler Help?

How many No-ops?

- sub $2, $1,$3
- and $4, $2,$5
- or $8, $2,$6
- add $9, $4,$2
- slt $1, $6,$7

With re-ordering?

- sub $2, $1,$3
- and $4, $2,$5
- or $8, $3,$6
- slt $1, $6,$7
- add $9, $2,$8
- slt $1, $6,$7

Data Hazards: Key points

- Pipelining provides high throughput, but does not handle data dependencies easily.
- Data dependencies cause data hazards.
  - Data hazards can be solved by:
    - software (nops)
    - hardware stalling
    - hardware forwarding
- All modern processors, use a combination of forwarding and stalling.
Branch Hazards

or

"Which way did he go?"

Data dependence: one instruction is dependent on another instruction to provide its operands.

Control dependence (aka branch dependences): one instructions determines whether another gets executed or not.

Control dependences are particularly critical with conditional branches.

• Software solution
  – Insert no-ops (Not popular)

• Hardware solutions
  – Stall until you know which direction branch goes
    – Guess which direction, start executing chosen path (but be prepared to undo any mistakes!)
  – Static branch prediction: base guess on instruction type
  – Dynamic branch prediction: base guess on execution history
  – Reduce the branch delay

• Software/hardware solution
  – delayed branch: always execute instruction after branch.
  – Compiler puts something useful (or a no-op) there.

When are branches resolved?

When are branches resolved?

Control Hazards: The stall solution

Delay for 3 bubbles EVERY time you see a branch instruction!

• All branches waste 3 cycles.
  – Seems wasteful, particularly when the branch isn’t taken.
  – It’s better to guess whether branch will be taken
  – Easiest guess is "branch isn’t taken"
Control Hazards: Speculative Execution

- Case 0: Branch not taken
  - works pretty well when you're right – no wasted cycles

Control Hazards: Speculative Execution

- Case 1: Branch taken
  - Same performance as stalling

1. Assume backwards branch is always taken, forward branch never is
   - "backwards" = negative displacement field
   - loops (which branch backwards) are usually executed multiple times.
   - "if-then-else" often takes the "then" (no branch) clause.
2. Compiler makes educated guess
   - sets "predict taken/not taken" bit in instruction

Reducing the cost of the branch delay

- It's easy to reduce stall to 2-cycles

Mis-prediction penalty is down to one cycle!

- Target computation & equality check in ID phase.
  - This figure also shows flushing lines.

Static speculation: look at instruction only
The branch delay slot is only useful if you can find something to put there.
- Need earlier instruction that doesn’t affect the branch
- If you can’t find anything, you must put a `nop` to ensure correctness.

Worked well for early RISC machines.
- Doesn’t help recent processors much
- E.g. MIPS R10000, has a 5-cycle branch penalty, and executes 4 instructions per cycle.
- Still works for the ARM7 (3 stage pipe)
  - But not for the ARM9/10 (5/7 stage pipes)

Delayed branch is a permanent part of the MIPS ISA.

Branch Prediction: non-static or dynamic

- Static branch prediction isn’t good enough when mispredicted branches waste 10 or 20 instructions.
- Dynamic branch prediction keeps a brief history of what happened at each branch.
- Branch-history tables and such like machinery!

Back to branch prediction

- Always assuming the branch is not taken is a crude form of branch prediction.
- What about loops that are taken 95% of the time?
  - We would like the option of assuming not taken for some branches, and taken for others, depending on ???

- Mispredict because either:
  - Wrong guess for that branch
  - Got branch history of wrong branch when index the table
- 4096 entry table programs vary from 1% misprediction (nasa, tomcat) to 18% (postgres), with spike at 9% and gc at 12%.
- 4096 about as good as infinite table, but 4096 is a lot of HW
Research goes on, in this space.

Dynamic Branch Prediction: Two bit are better than one!

This one means "the last two branches of this location were taken."

Branch Prediction: The POWER4

Branch Prediction: To help mitigate the effects of the long pipeline necessitated by the high frequency design, POWER4 invests heavily in branch prediction mechanisms.

- The branch prediction logic scans the fetched instructions looking for up to two branches each cycle.
- Depending upon the branch type found, various branch prediction mechanisms engage to help predict the branch direction or the target address of the branch or both.
- Branch direction for unconditional branches are not predicted.
- All conditional branches are predicted, even if the condition register bits upon which they are dependent are known at instruction fetch time.
- Branch target addresses for the PowerPC branch to link register (btl) and branch to count register (btc) instructions can be predicted using a hardware implemented btl, stack and count cache mechanism, respectively.
- Target addresses for absolute and relative branches are computed directly as part of the branch scan function.

As branch instructions flow through the rest of the pipeline, all ultimately execute in the branch execution unit, the actual outcome of the branches are determined. At that point if the branch prediction was found to be correct, the branch instructions are simply completed like all other instructions.

In the event that a prediction is found to be incorrect, the instruction fetch logic causes the mispredicted instructions to be discarded and starts re-fetching instructions along the alternative path.

Need Address @ Same Time as Prediction

- Branch Target Buffer (BTB): Address of branch index to get prediction AND branch address (if taken)
- Return instruction addresses predicted with stack

Branch Hazards: Summary

Branch (or control) hazards arise because we must fetch the next instruction before we know if we are branching or not.

Branch hazards are detected in hardware.

We can reduce the impact of branch hazards through:
- Computing branch target and testing early
- Branch delay slots
- Branch prediction = static or dynamic

What about Interrupts, Traps, Faults?

- Exceptions represent another form of control dependence.
- Therefore, they create a potential branch hazard.
- Exceptions must be recognized early enough in the pipeline that subsequent instructions can be flushed before they change any permanent state.
- As long as we do that, everything else works the same as before.
- Exception handling that always correctly identifies the offending instruction is called precise interrupts.

- External Interrupts:
  - Allow pipeline to drain,
  - Load PC with interrupt address
- Faults (within instruction, restartable)
  - Force trap instruction into IF
  - disable writes till trap hits WIB
  - must save multiple PCs or PG + state

Interrupts, Traps, Faults?

- External Interrupts:
  - Allow pipeline to drain,
  - Load PC with interrupt address

- Faults (within instruction, restartable)
  - Force trap instruction into IF
  - disable writes till trap hits WIB
  - must save multiple PCs or PG + state

- External Interrupts:
  - Allow pipeline to drain,
  - Load PC with interrupt address

- Faults (within instruction, restartable)
  - Force trap instruction into IF
  - disable writes till trap hits WIB
  - must save multiple PCs or PG + state
One Solution: Freeze & Bubble

Interrupts, Traps, Faults?
• Exceptions/Interrupts: 5 instructions executing in 5 stage pipeline
  – How to stop the pipeline?
  – Restart?
  – Who caused the interrupt?
Stage Problem interrupts occurring
IF Page fault on instruction fetch; misaligned memory access; memory-protection violation
ID Undefined or illegal opcode
EX Arithmetic exception
MEM Page fault on data fetch; misaligned memory access; memory-protection violation; memory error
• Load with data page fault, Add with instruction page fault?
• Solution 1: interrupt vector/Instruction 2: interrupt ASAP, restart everything incomplete

What about the real world?
• Not fundamentally different than the techniques we discussed
  – Deeper pipelines
• Pipelining is combined with
  – superscalar execution
  – out-of-order execution
  – VLIW (very-long-instruction-word)

Deeper Pipelines
• How much deeper is productive? What are the limiting effects?
  – Pipeline latching overhead
  – Losses due to stalls and hazards
  – Clock speeds achievable

Superscalar Execution
• To execute four instructions in the same cycle, we must find four independent instructions
  • If the four instructions fetched are guaranteed by the compiler to be independent, this is a VLIW machine
  • If the four instructions fetched are only executed together if hardware confirms that they are independent, this is an in-order superscalar processor.
  • If the hardware actively finds four (not necessarily consecutive) instructions that are independent, this is an out-of-order superscalar processor.
Example: Simple Superscalar

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

### Type Pipe Stages

- Integer (Int) instruction
- Floating-Point (FP) instruction

### Example: Complex Superscalar: Multiple Pipes

- Reg. File ports
- Detecting Data Dependences
- Bypassing
- RAW Hazard
- WAR Hazard
- Multiple load/store ops?
- Branches

Software Pipelining

- Observation: if iterations from loops are independent, then can get ILP by taking instructions from different iterations
- Software pipelining: reorganizes loops so that each iteration is made from instructions chosen from different iterations of the original loop

Unrolled Loop that Minimizes Stalls for Scalar

1. Loop: 1
   - LD F0, 0(R1)
   - ADD F4,F0,F2
   - SD 0(R1),F4
   - ADD F8,F0,F2
   - LD F10,-16(R1)
   - SUBI R1,R1,#8
   - BNEZ R1,loop

2. Loop: 2
   - LD F0, 0(R1)
   - ADD F4,F0,F2
   - SD 0(R1),F4
   - ADD F8,F0,F2
   - LD F10,-16(R1)
   - SUBI R1,R1,#8
   - BNEZ R1,loop

3. Loop: 3
   - LD F0, 0(R1)
   - ADD F4,F0,F2
   - SD 0(R1),F4
   - ADD F8,F0,F2
   - LD F10,-16(R1)
   - SUBI R1,R1,#8
   - BNEZ R1,loop
Out of order execution

- Issues (begins execution of) an instruction as soon as all of its dependences are satisfied, even if prior instructions are stalled.

Reservation stations:
Used to manage dynamic scheduling

lw $6, 36($2)
add $5, $6, $4
lw $7, 1000($5)
sub $9, $12, $5
sw $5, 200($6)
add $3, $9, $9
and $11, $7, $6

Out of order execution at the hardware level

Issues in pipeline design

- Pipelining
- Super-pipeline
  - Issue one instruction per (fast) cycle
  - ALU takes multiple cycles
- Super-scalar
  - Issue multiple scalar instructions per cycle
- VLW ("EPIC")
  - Each instruction specifies multiple scalar operations
  - Compiler determines parallelism
- Vector operations
  - Each instruction specifies series of identical operations

Limitations

Issue rate, FU stalls, FU depth
Clock skew, FU stalls, FU depth
Hazard resolution
Packing
Applicability

Issues in pipeline design

- Pipelines pass control information down the pipe just as data moves down pipe
- Forwarding/Stats handled by local control
- Exceptions stop the pipeline
- MIPS I instruction set architecture made pipeline visible (delayed branch, delayed load)
- More performance from deeper pipelines, parallelism

ET = Number of instructions * CPI * cycle time
- Data hazards and branch hazards prevent CPI from reaching 1.0, but forwarding and branch prediction get it pretty close.
- Data hazards and branch hazards need to be detected by hardware.
- Pipeline control uses combinational logic. All data and control signals move together through the pipeline.
- Pipelining attempts to get CPI close to 1. To improve performance we must reduce CycleTime (superpipelining) or CPI below one (superscalar, VLW).