CS/COE1541: Introduction to Computer Architecture

Pipelining

Sangyeun Cho

Computer Science Department
University of Pittsburgh

Five instruction execution steps

- Instruction fetch
- Instruction decode and register read
- Execution, memory address calculation, or branch completion
- Memory access or R-type instruction completion
- Write-back

- Instruction execution takes 3~5 cycles!

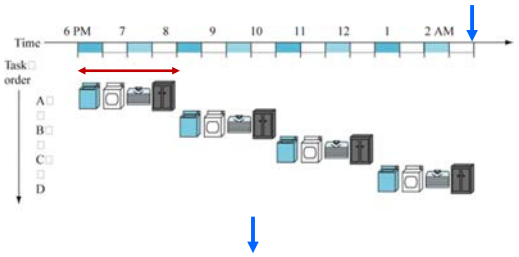
Single-cycle vs. multi-cycle

- Resource usage
  - Single-cycle:
  - Multi-cycle:
- Control design
  - Single-cycle:
  - Multi-cycle:
- Performance (CPI×CCT)
  - Single-cycle
  - Multi-cycle

Goal of pipelining is…

- Throughput!
- Observation
  - Instructions follow "steps"
  - Some steps are common (e.g., fetch, decode)
    - Because we "program" ALU, the execution step is also common
- Pipelining
  - Basic idea: overlap instruction execution
    - While instruction #1 is in the decode stage, fetch instruction #2
    - Provide more resources such that overlapping is not hindered by sharing of resources

Laundry analogy



Each task's latency is not reduced!

Because instruction throughput is improved, program latency is reduced!

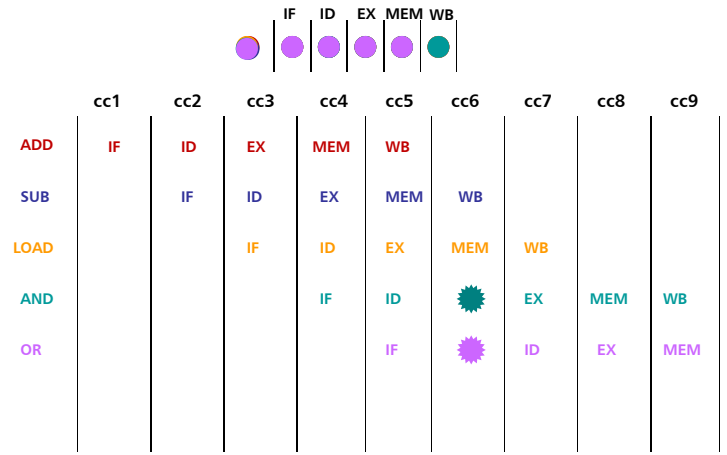
Pipelining instruction execution

- Consider instruction execution steps
  - Fetch instruction from memory
    - Separate instruction memory (Harvard architecture) vs. single memory (von Neumann)
  - Decode instruction
  - Read operands from register file
  - Perform specified computation
  - Access memory if required
    - Need to compute effective address beforehand
  - Store result to specified register if needed
- Make sure different pipeline stages can simultaneously work on different instructions
  - Pipelined datapath
  - Pipelined control

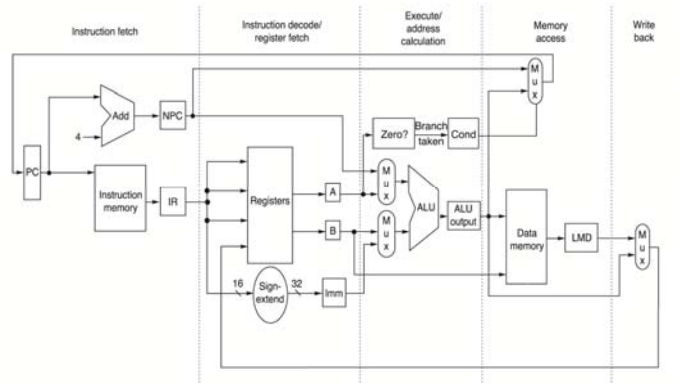
Five pipeline stages

- This is a "vanilla" design
- In fact, some commercial processors follow this design
  - Early MIPS design
  - ARM9 (very popular embedded processor core)
- Fetch (F)
  - Fetch instruction
- Decode (D)
  - Decode instruction and read operands
- Execute (X)
  - ALU operation, address calculation in the case of memory access
- Memory access (M)
- Write back (W)
  - Update register file

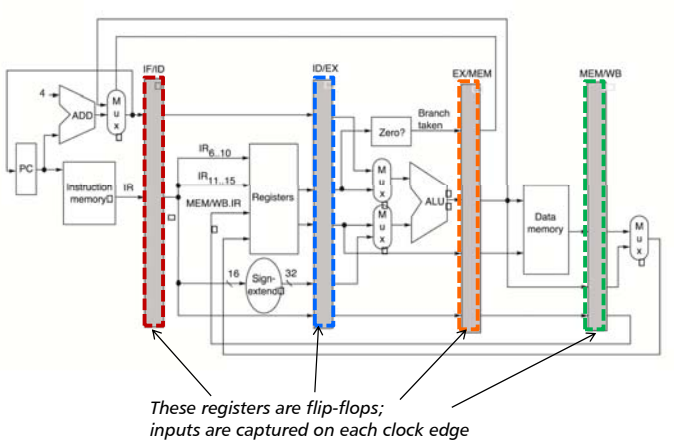
Pipelining instruction execution



Multi-cycle to pipelined datapath

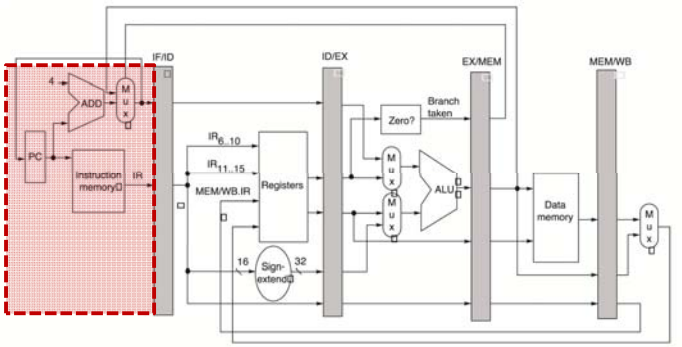


Multi-cycle to pipelined datapath



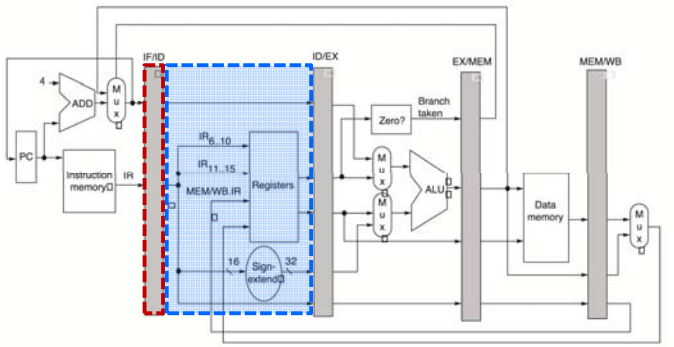
These registers are flip-flops; inputs are captured on each clock edge

lw in the "F" stage



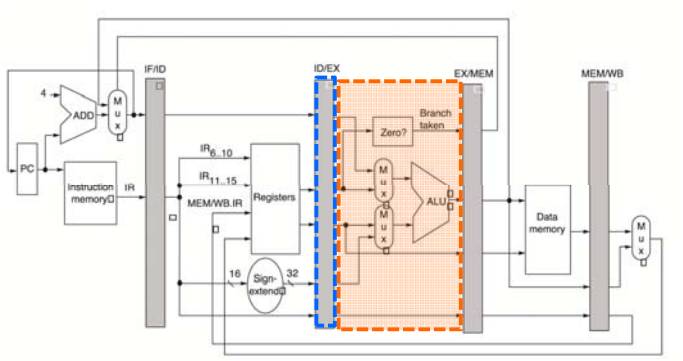
Read an instruction from instruction memory; address is in PC; compute (PC+4) to update PC

lw in the "D" stage



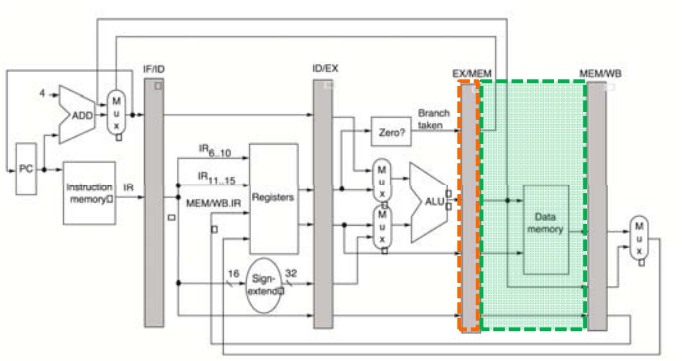
Read operands from register file; sign-extend the immediate field

lw in the "X" stage



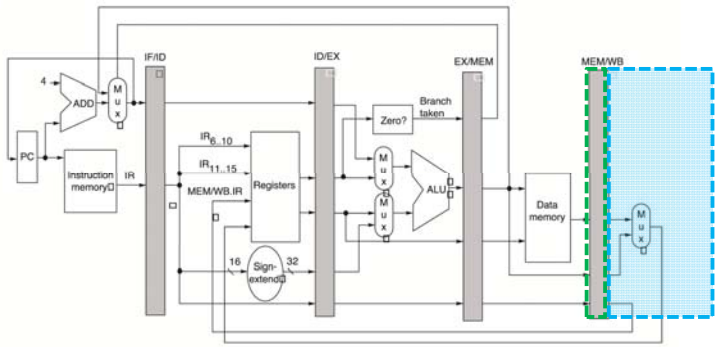
Add the base register value and the immediate value to form memory access address;

lw in the "M" stage



Read a value from memory

lw in the "W" stage

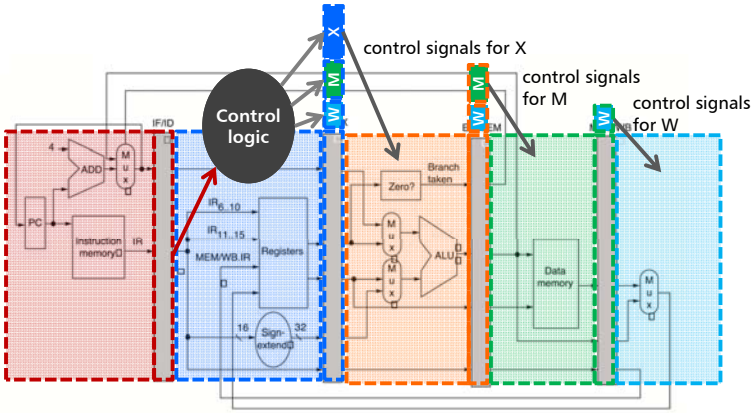


Update register file

Pipeline control

- There are multiple instructions in flight (in different pipeline stages)
- Hence, control signals for an instruction should flow through the pipeline stages with the instruction
- Alternatively, with the instruction information in each pipeline register, one can generate control signals by decoding the information
- Pipeline control becomes more complex than previous designs because of potential dependences between instructions in flight

Pipeline control



Pipelining performance

- Time between instructions
  - Pipelining: 1 cycle
    - ~Time for one instruction (# cycles) / # pipeline stages
  - Non-pipelined
    - Single-cycle: 1 cycle (but it is LONG)
    - Multi-cycle: N cycles (depending on instruction)
- Pipelining does NOT improve latency of a single instruction
  - In fact, instruction execution latency can be even longer (why?)
  - Pipelining improves "throughput" (what is throughput?)
  - It improves the program execution time (why?)

MIPS ISA and pipelining

- Fixed instruction length (4 bytes)
  - Allows simple fetching (c.f., IA32)
- Few instruction formats; source register fields fixed
  - Quick register operand fetching from register file
- Load/store architecture
  - No need to place arithmetic operation after memory (c.f., IA32)
- Memory operands are aligned

Pipelining facts summary

- Pipelining increases throughput, not latency (of instruction)
- Clock cycle time is limited by the longest pipeline stage
- Potential speedup = # of pipeline stages
- Time to fill and time to drain pipeline can affect performance
- Can you explain the trade-offs between different processor implementation strategies?
  - Single-cycle vs. multi-cycle vs. pipelining

Pipeline hazards

- Hazards are the conditions that hinder seamless instruction execution through pipeline stages
- Three types of hazards
  - Structural: hardware can't support a particular sequence of instructions (due to lack of resources)
  - Data: an instruction depends on a prior instruction (to produce its result) still in execution
    - E.g., lw followed by an add instruction using the loaded value
  - Control: can't decide if this instruction should be executed due to a prior branch instruction in execution

Tackling hazards

- Hardware approach – pipeline "interlock"
  - Detection: continuously check conditions that lead to a hazard
  - Treatment: insert a "bubble" in the pipeline to delay instruction execution such that the condition disappears
  - The bubble is also called "pipeline stall"
- Software approach
  - Detection: compiler inspects the generated code and sees if there is an instruction sequence that will lead to a pipeline hazard
  - Treatment: insert a "NOP" instruction to avoid the hazard condition
  - Compiler must have knowledge about the hardware (pipeline)
- Trade-offs?

Structural hazard

- Register file example
  - Let's assume that we decided to support (R+R) addressing mode in memory instructions
    - E.g., lw $4, ($5+$3) and sw $4, ($5+$3)
  - In the case of the store instruction above, we need to read from the register file three times
  - If we have only two read ports, what will happen?
  - What about some other addressing mode like post-increment?
    - lw $4, 16($5+)
- ALU example
  - Can we use ALU for branch target computation?

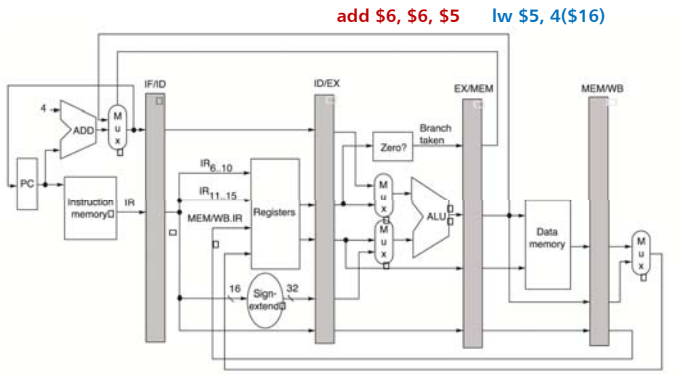
Data hazard

- Consider the following code sequence:

```
1    BACK:
2    lw $5, 4($16)
3    add $6, $6, $5
4    addi $16, $16, 4
5    bne $16, $4, BACK
```

- add (in line 3) has $5 as its input operand, while lw has $5 as its output
  - add can execute only after lw produces its result in $5
- In pipelined execution, when add is in its execution stage, lw is in its memory stage and has not produced its result!

Data hazard



How many bubbles do we need?

Tackling data hazards

- Source of the problem is *dependency between nearby instructions*
  - add **$6**, $5, $4
  - sub $3, **$6**, $8

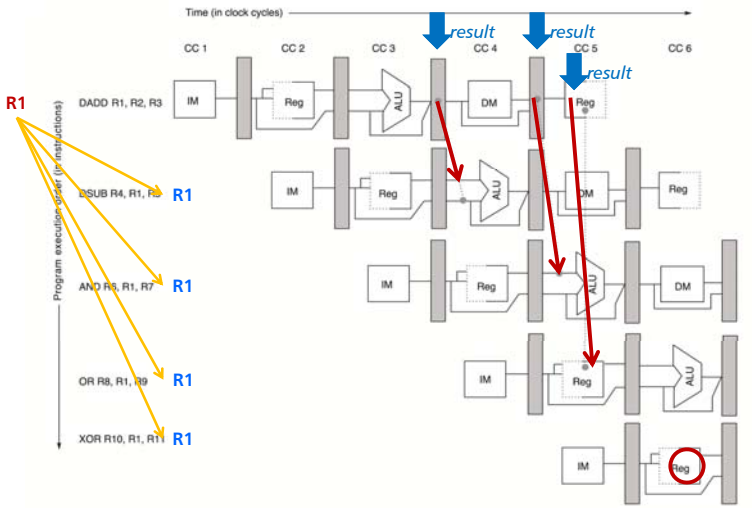
```
add $6, $5, $4
nop
nop
sub $3, $6, $8
```

- Solutions
  - Software approach: reorder instructions in the program so that dependent instructions are apart; insert NOPs if reordering is not sufficient
  - Hardware approach: detect dependence and stall the second instruction (dependent on the first one) in the decoding stage until when the data becomes ready in the register file
- What are trade-offs between these two approaches?

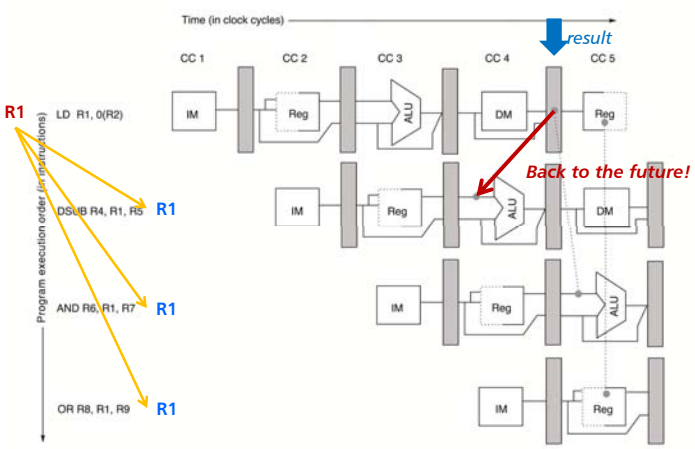
Tackling data hazards

- The severity of the problem is determined by the difference in the timing of register file update (first instruction) and the timing of register file read (second instruction)
  - In our 5-stage design, register update is in the 5th stage and register read is in the 2nd stage
  - If register update and read can be done within a single cycle, basically we need two bubbles (unused instruction execution slots!) for the two dependent instructions back-to-back
- This severity can be tackled by *passing the result of the first instruction to the second directly without going through the register file*
  - This is called **data forwarding** (or sometimes **bypassing**)

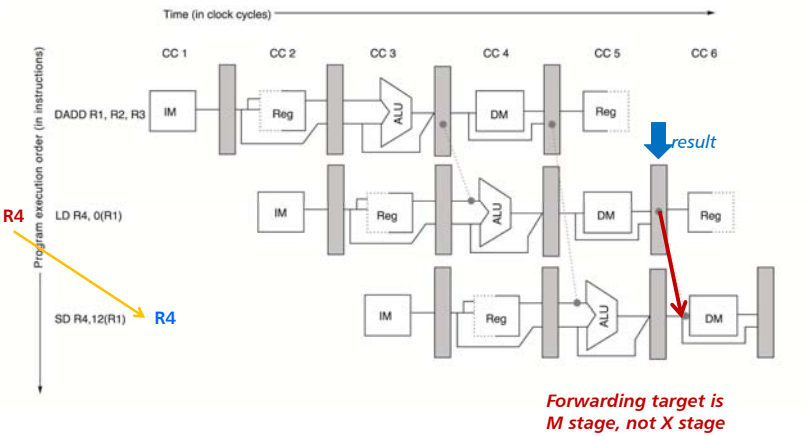
Data forwarding



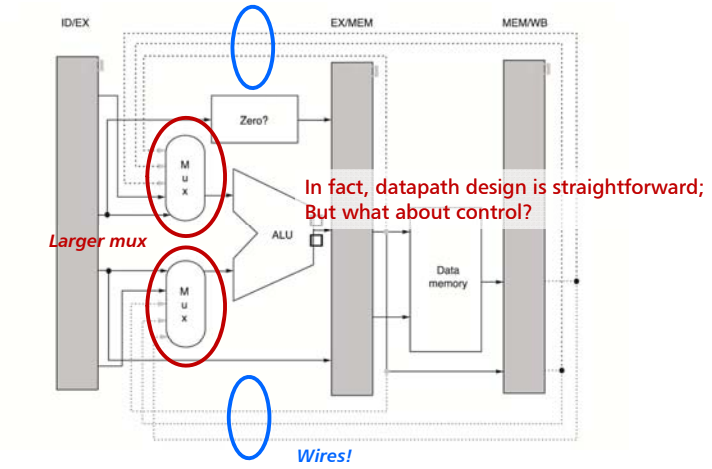
Data forwarding



Data forwarding



Data forwarding hardware



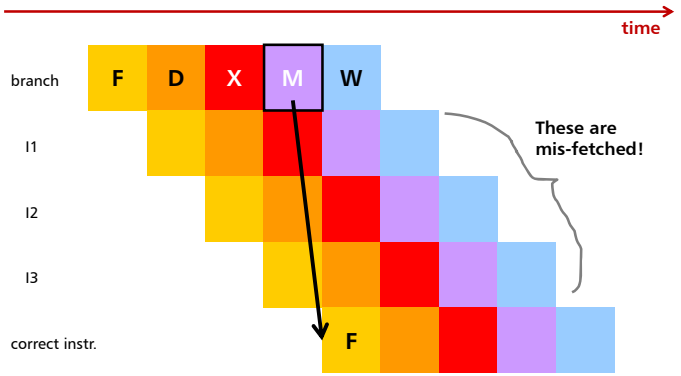
Control hazard

- Consider the following code sequence:

```
1       beq $4, $5, FOO
2       add $6, $6, $10
3       j BAR
4       FOO:
5       sub $6, $6, $10
6       BAR:
```

- After fetching beq (in line 1), do we know where to go, say to line 4 or line 2?
  - Before beq reaches its execution stage to evaluate ($4==$5) if it will branch or not is not known
- What shall we do?

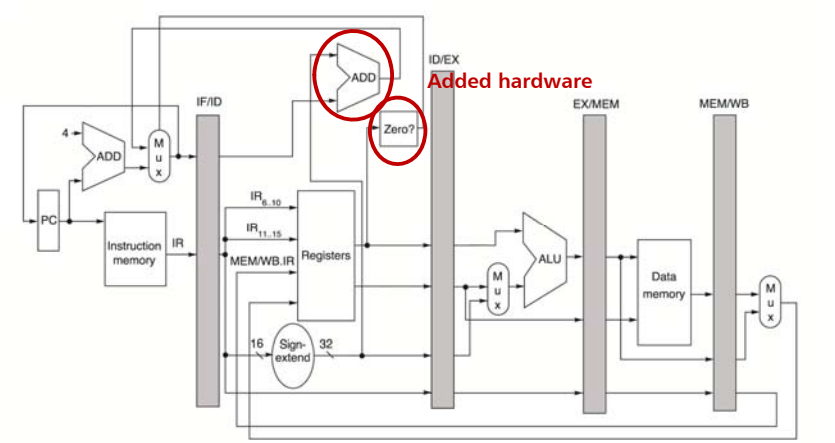
Control hazard



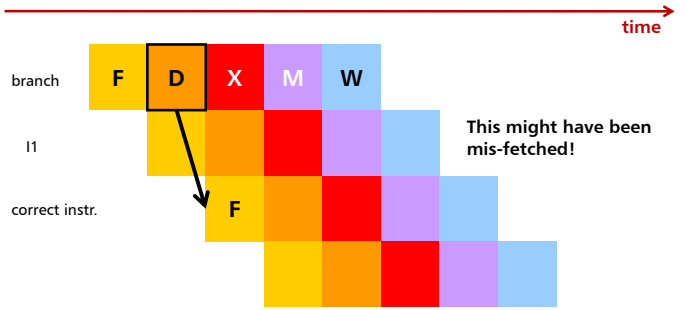
Impact of branch stalls

- If CPI=1, 20% of instructions are branches, 3 cycle stalls
  - CPI = 1 + 0.2×3=1.6
- Solution
  - Determine early if branch is taken or not
  - Compute branch target address early
- Example
  - Move zero test to ID stage
  - Separate adder to calculate new PC in ID stage

Modified pipeline



Control hazard



Pipeline depth vs. branch penalty

- Today's processors employ a deep pipeline (possibly more than 20 stages!) to increase the clock rate
  - Many stages means smaller amount of work per stage ⇒ shorter time needed per stage ⇒ higher clock rate!
- But what about branch penalty?
  - Penalty depends on the pipeline length!
  - Branches represent 15~20% of all instructions executed
- Situation is compounded by the increased issue bandwidth (superscalar processors)
  - Lost instruction execution opportunities are branch penalty × issue width
  - Accurate branch prediction mechanism is needed

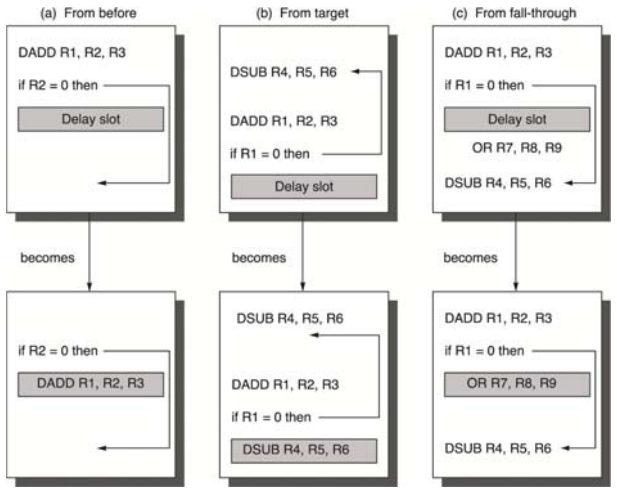
Branch handling strategies

- Stall until branch direction is known
- Predict "NOT TAKEN"
  - Execute fall-off instructions that follow
  - Squash (or cancel) instructions in pipeline if branch is actually taken
  - (PC+4) already computed, so use it to get next instruction
- Predict "TAKEN"
  - 67% MIPS branches taken on average
  - Start fetching from the taken path as soon as the target address becomes available
- Delayed branch

Delayed branch

- Assume that we have "N" fetch slots after a branch before fetching the branch target
  - The instructions in the slots are executed regardless of the branch outcome
  - N is set to be the number of cycles needed to resolve the branch
- It's compiler's job to find instructions to fill the slots
  - If you don't have instructions to fill the slots, put NOPs there.
- Simple hardware – no branch interlock
- Possibly more performance – if slots are filled with useful instructions
- Code size will increase

Delayed branch



Branch prediction

- Goal
  - Predict the branch outcome (taken or not taken) and the branch target (when taken, where should we go?)
  - In short, just figure out what's the next PC?
- When do we predict?
  - Predict the next PC when we are fetching from the current PC
- How?
  - We will study this…

What to predict – T/NT

- Let's first focus on predicting taken (T)/not taken (NT)
- Static prediction
  - Associate with each branch a hint
    - Always taken
    - Always not taken
    - (Don't know)
  - Forward not taken, backward taken
  - Compiler hint
- Dynamic prediction
  - Simple 1-bit predictor
    - Remember the last behavior (taken/not taken)
  - 2-bit predictor
    - Bias added
  - Combined
    - Choose between two predictors

1-bit predictor

- Remember the last behavior of the branch

```
for (i=0; i< 100; i++) {
  A[i] = B[i] * C[i];
  D[i] = E[i] / F[i];
}
```

Assume that this is a branch

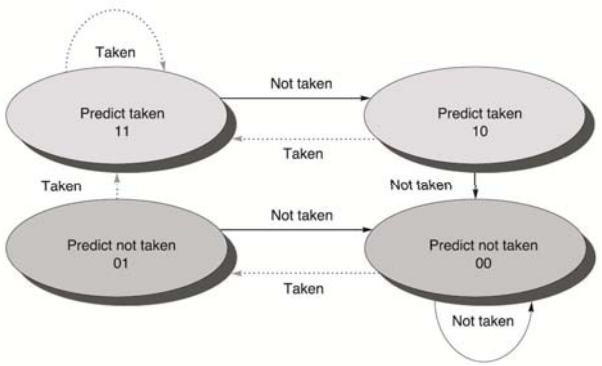
- How many prediction hits and misses? Prediction accuracy?

```
for (j=0; j<10; j++) {
  for (i=0; i< 100; i++) {
    A[i] = B[i] * C[i];
    D[i] = E[i] / F[i];
  }
}
```

- Any shortcoming in this scheme?

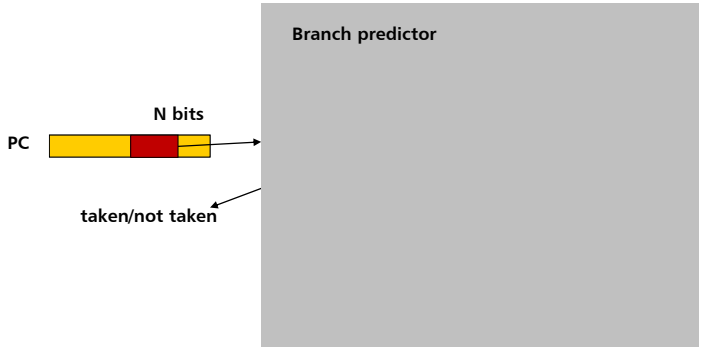
2-bit predictor

- Requires two consecutive mispredictions to flip direction

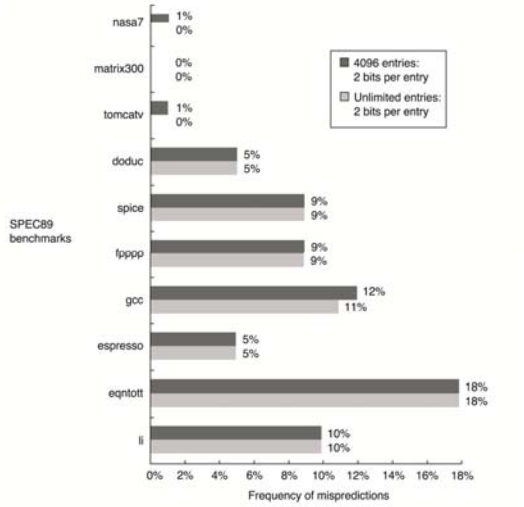


Branch prediction buffer

- Special cache to keep 2^N 2-bit counters, indexed with (part of) PC



2-bit predictor performance



Correlating predictor

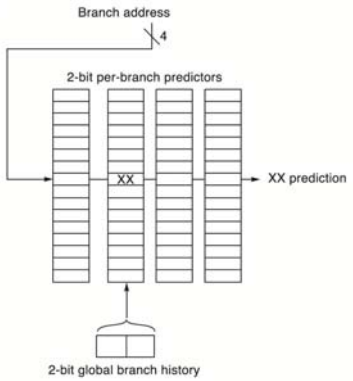
- Behavior of a branch can be correlated with other branches

```
if (aa==2)
  aa = 0;
if (bb==2)
  bb = 0;
if (aa!=bb) {
  …
}
```

- Branch history
  - N-bit vector keeping the last N branches' outcomes (it's a shift register)
  - 11001101 = TTNNTTNT (the rightmost being the most recent)
- Also called a "global" predictor
- How shall we utilize this branch history?

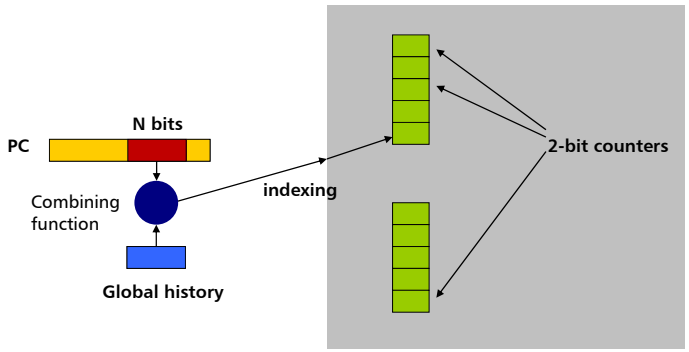
(m,n) predictor

- Considers last m branches (m-bit global history)
- Choose from 2^m separate BP arrays, each has n-bit predictor

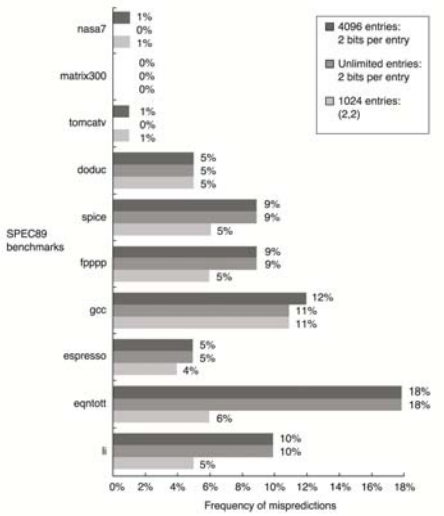


Another way of utilizing history

- Form index from PC and GH



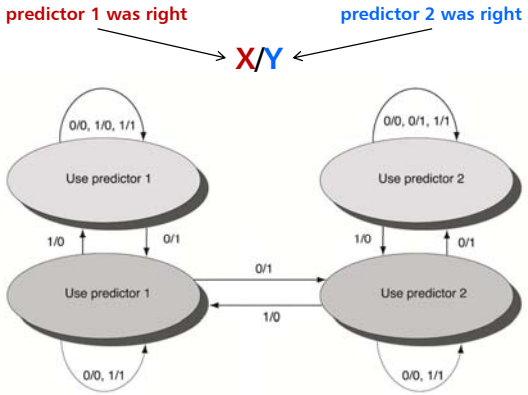
(2,2) predictor performance



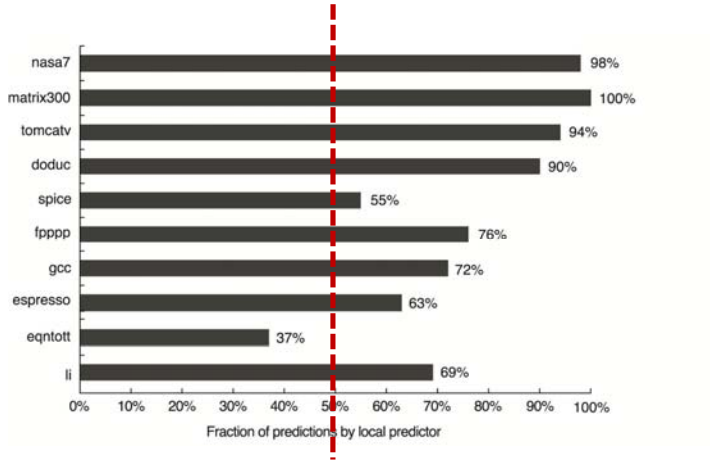
Combined predictor

- Question: "Is a local predictor better or a global predictor better for a specific branch?"
- Choose between a local and a global predictor
- Selector – is a predictor to choose between the two predictors
- Alpha 21264 example
  - A 4K-entry global predictor (2-bit counters)
    - Indexed by 12-bit global history
  - A hierarchical local predictor
    - 1K 10-bit pattern table
    - 1K-entry 3-bit counters
  - A tournament predictor (selector)
    - 4K-entry 2-bit counters

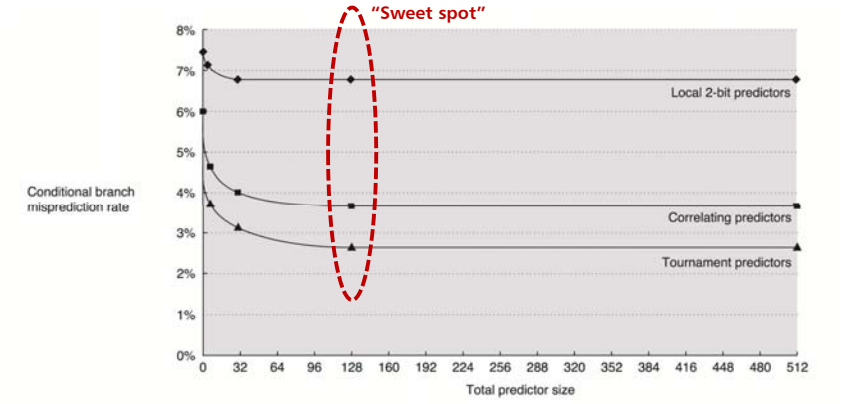
Selector design



Local vs. global?



Combined predictor performance



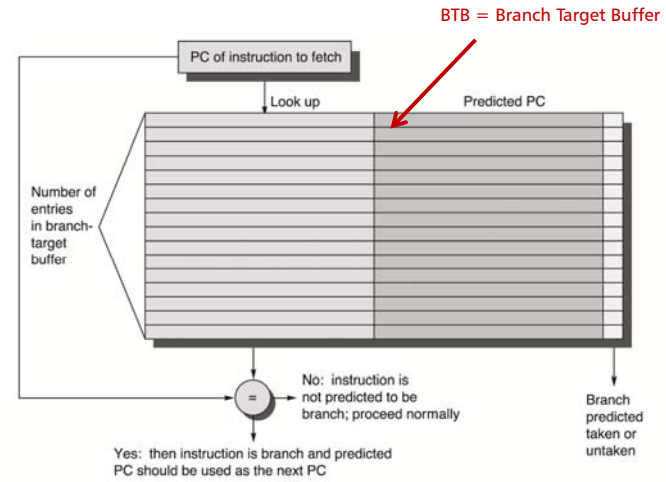
What to predict – target

- Remember – the goal of branch prediction is to determine the next PC (target of fetching) every cycle
- Requirements
  - When fetching a branch, we need to predict (simultaneously with fetching) if it's going to be taken or not ⇐ we talked about this
  - At the same time, we need to determine the target of the branch if the branch is predicted taken ⇒ we are going to talk about this

Target prediction

- It's much more difficult to "predict" target
  - Taken/Not taken – just two cases
  - A 32-bit target has 2^{32} possibilities!
- But taken target remains the same!
  - Just remember the last target then…

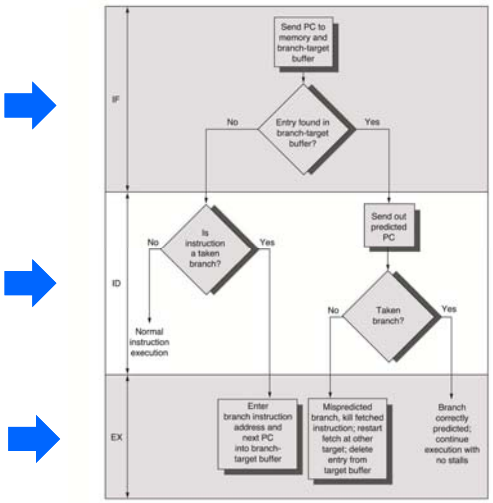
Target prediction w/ BTB



Target prediction w/ BTB

- Use "PC" to look up – why PC?
  - "Match" means it's a branch for sure
  - All-bit matching needed; why?
- If *match* and *Predicted Taken*, use the stored target for the next PC
- When no match and it's a branch (detected later)
  - Use some other prediction
  - Assume it's not taken
- After processing a branch, update BTB with correct information

Branch prediction & pipelining



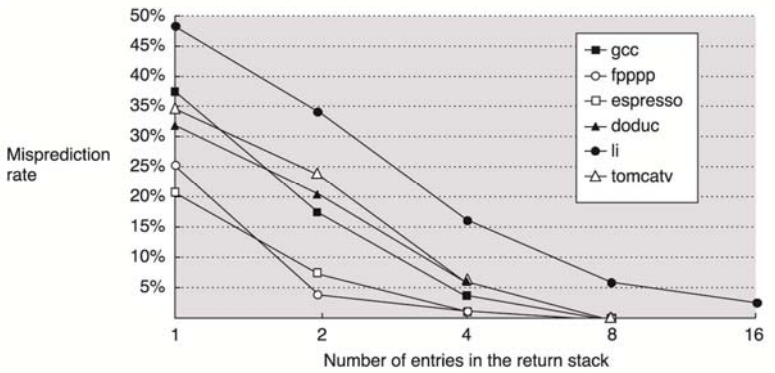
BTB performance

- Two bad cases
  - Wrong prediction (or "misprediction")
  - A branch not found in BTB
- Example: Prediction accuracy is 90%, hit rate in BTB is 90%, 2-cycle penalty, 60% of branches taken
  - Probability (branch in buffer & misprediction) = 90%×10% = 0.09
  - Probability (branch not in buffer & taken) = 10%×60% = 0.06
  - Branch penalty = (0.09+0.06)×2 = 0.30 (cycles)

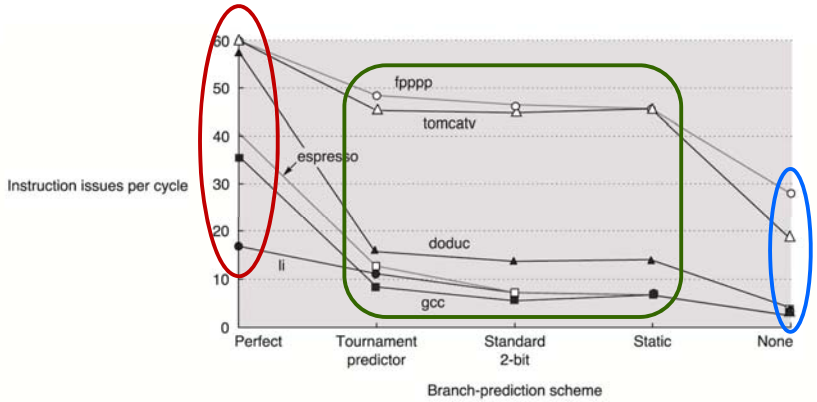
What about "indirect jumps"?

- Indirect jumps
  - Branch target is not unique
  - E.g., jr $31
- BTB has a single target PC entry – can't store multiple targets…
- With multiple targets stored, how do we choose the right target?
- Fortunately, most indirect jumps are for *function return*
- Return target can be predicted using a stack ⇒ Return Address Stack (RAS)
  - The basic idea is to keep storing all the return addresses in a *Last In First Out* manner

RAS performance



Performance of branch prediction



- On a hypothetical "64-issue" superscalar processor model with 2k instruction window

Some key points

- Hazards result in pipeline bubbles
  - Performance degradation – non-ideal, higher-than-1 CPI
- Data hazards require dependent instruction wait for the operands to become available
  - Forwarding helps reduce the penalty
  - Stalls still required in some cases
- Control hazards require control-dependent instructions to wait for the branch to be resolved
- Branch prediction is a integral mechanism found in all high-performance processors
  - Program performance is very sensitive to branch prediction accuracy in modern pipelined processors

What is instruction level parallelism?

- Execute *independent instructions* in parallel
  - Provide more hardware function units (e.g., adders, cache ports)
  - Detect instructions that can be executed in parallel (in hardware or software)
  - Schedule instructions to multiple function units (in hardware or software)
- Goal is to improve instruction throughput

- How does it differ from general parallel processing?
- How does it differ from pipelining?
  - Ideal CPI of single pipeline is 1
  - W/ ILP we want CPI < 1 (or IPC > 1)

Key questions

- How do we find parallel instructions?
  - Static, compile-time vs. dynamic, run-time
  - Data dependence and control dependence place high bars

- What is the role of ISA for ILP packaging?
  - VLIW approach (static compiler approach)
  - Superscalar approach (dynamic hardware approach)

- How can we exploit ILP at run time?
  - Minimal hardware support (w/ compiler support)
  - Dynamic OOO (out-of-order) execution support

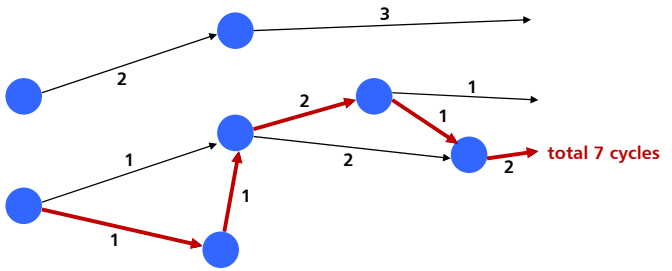
Data dependence

- Instructions consume values (operands) created by previous instructions

```
add r1, r2, r3
mul r4, r1, r5
```

- Given a sequence of instructions to execute, form a directed graph using producer-consumer relationships (not instruction order in the program)
  - Nodes are instructions
  - Edges represent dependences, possibly labeled with related information such as latency, etc.

Data dependence



- What is the minimum execution time, given unlimited resources?

- Other questions
  - Can we have a directed cycle?
  - What is the max. # of instructions that can be executed in parallel?
  - How do we map instructions to (limited) resources? In what order?

List scheduling (example impl.)

- Common compiler instruction scheduling algorithm
- Procedure
  - Build DPG (data precedence graph)
  - Assign priorities to nodes
  - Perform scheduling
    - Start from cycle 0, schedule "ready" instructions
    - When there are multiple ready instructions, choose the one w/ highest priority

List scheduling (example impl.)

```
cycle = 0
ready-list = root nodes in DPG
inflight-list = empty list
while ( ready-list or inflight-list not empty, and an issue slot is available )
    for op = (all nodes in ready-list in descending priority order)
        if (a functional unit exists for op to start at cycle)
            remove op from ready-list and add to inflight-list
            add op to schedule at time cycle
            if (op has an outgoing anti-edge)
                Add all targets of op's anti-edges that are ready to ready-list
            endif
        endif
    endfor
    cycle = cycle + 1
    for op = (all nodes in inflight-list)
        if (op finishes at time cycle)
            remove op from inflight-list
            check nodes waiting for op in DPG and add to ready-list
                if all operands available
        endif
    endfor
endwhile
```

(Cooper et al. '98)

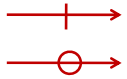
Name dependence

- Data dependence is "true" dependence
  - The consumer instruction can't be scheduled before the producer one
  - "Read-after-write" (RAW)
- Dependences may be caused by the name of the storage used in the instructions, not by the producer-consumer relationship
  - These are "false" dependences

```
add r1, r2, r3
mul r2, r4, r5
```

```
add r2, r3, r4
…
mul r2, r5, r6
```

- Anti dependence ("write-after-read" or WAR)
- Output dependence ("write-after-write" or WAW)



Name dependence

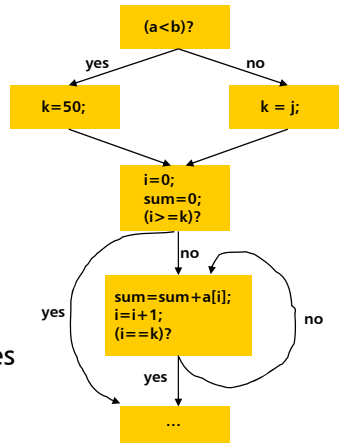
- Name dependences may be removed if we have plenty of storage (*i.e.*, many registers)
- Compilers typically assume that there are unlimited registers
  - You can dump GCC internal representations (before register allocation) to confirm this
  - Compiler maps such "virtual" registers to "architected" registers
  - Compiler may generate code to store temporary values to memory when there are not enough registers
- Hardware can do the opposite – mapping architected registers ("virtual") to some physical register – to remove name dependences ⇒ register renaming

Control dependence

- It determines (limits) the ordering of *an instruction i* with respect to *a branch instruction* so that the instruction i is executed in correct program order and only when it should be
- Why are control dependences barriers for extracting more parallelism and performance?
  - Pipelined processor
  - Compiler scheduling

Control flow graph (CFG)

```
if (a<b)
  k=50;
else
  k = j;
for (sum=0,i=0;i < k; i++) {
  sum+=a[i];
}
```



- Nodes: basic blocks
- Edges: possible control changes
- How can you construct CFG?

Static vs. dynamic scheduling

- Static scheduling
  - Schedule instructions at compiler time to get the best execution time
- Dynamic scheduling
  - Hardware changes instruction execution sequence in order to minimize execution time
  - Dependences must be honored
  - For the best result, false dependences must be removed
  - For the best result, control dependences must be tackled
- Implementing precise exception is important

Dynamic scheduling

- Components
  - Check for dependences ⇒ "do we have ready instructions?"
  - Select ready instructions and map them to multiple function units
  - The procedure is similar to find parallel instructions from DPG
- Instruction window
  - When we look for parallel instructions, we want to consider many instructions (in "instruction window") for the best result
  - Branches hinder forming a large, accurate window

Across branches

- A loop example

```
for (i=0; i< 1000; i++) {
  A[i] = B[i] * C[i];
  D[i] = A[i] / F[i];
}

for (j=1; j< 1000; j++) {
  A[j] = A[j – 1] / C[j];
}

for (k=0; k< 1000; k++) {
  Y = Y + A[k] / F[k];
}
```


Loop unrolling

```
for (i=0; i< 1000; i++) {
  A[i] = B[i] * C[i];
  D[i] = A[i] / F[i];
}
```

```
for (i=0; i< 1000; i+=4) {
  A[i] = B[i] * C[i];
  A[i+1] = B[i+1] * C[i+1];
  A[i+2] = B[i+2] * C[i+2];
  A[i+3] = B[i+3] * C[i+3];
  D[i] = A[i] / F[i];
  D[i+1] = A[i+1] / F[i+1];
  D[i+2] = A[i+2] / F[i+2];
  D[i+3] = A[i+3] / F[i+3];
}
```


Handling exceptions

- *Precise exception*
  - Based on the *sequential program execution model*
  - Execute all instructions before the faulting instruction
  - Do not execute the faulting instruction and the following ones
  - Allow resuming from the faulting instruction after the exception is taken care of
- What are needed?
  - Recording the faulting instruction's PC in a special place
  - Recording the cause of the exception
  - Squashing the instructions (inclusive of the faulting one)
  - Jumping to a pre-determined exception handling routine

Handling exceptions

- Exceptions/interrupts are an important part of a processor
- Modern OS depends on the hardware support for exceptions
  - Debuggers, too
- Interrupt mechanism allows efficient use of CPU cycles by getting hold of it only when it's necessary
  - C.f. polling
- Implementing the interrupt mechanism (e.g., precise exception) usually is complicated by processor implementation artifacts such as pipelining; it requires a lot of validation efforts in a processor design process