Universiteit Leiden

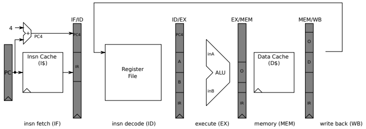
liacs Leiden Institute of Advanced Computer Science

EssCS - Topic 2

Introduction to Computer Architecture

Lecture 9, 29.10.2024
Nuša Zidarič

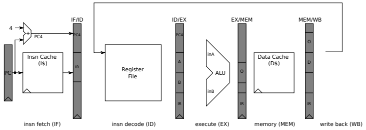
Recap: CPU Datapath



- **Optimizations are not free!** We have observed many trade-offs
  - single cycle datapath (large t_{CLK}, $Tput = 1$, $\mathrm{CPI} = 1$)
  - multicycle datapath (small t_{CLK}, $Tput =$small, $\mathrm{CPI} > 1$)
  - pipelined padapath (small t_{CLK}, $Tput = 1$, $\mathrm{CPI} = 1$)
    NOTE: actual CPI and Tput worse due to stalls

    data dependencies (RAW, WAR, WAW, RAR) $\Rightarrow$ RAW is a DATA HAZARD!
    - solution #1: stalling (NOPs, bubbles) $\rightarrow$ smaller Tput, bigger CPI, longer execution time, stalling logic (bigger)
    - solution #2: bypassing $\rightarrow$ bypassing logic (bigger)

Recap: CPU Datapath



- **Optimizations are not free!** We have observed many trade-offs
  - single cycle datapath (large t_{CLK}, $Tput = 1$, $\text{CPI} = 1$)
  - multicycle datapath (small t_{CLK}, $Tput =$small, $\text{CPI} > 1$)
  - pipelined padapath (small t_{CLK}, $Tput = 1$, $\text{CPI} = 1$)
    NOTE: actual CPI and Tput worse due to stalls

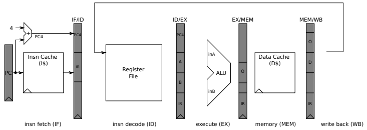
    data dependencies (RAW, WAR, WAW, RAR) $\Rightarrow$ RAW is a DATA HAZARD!
    - solution #1: stalling (NOPs, bubbles) $\rightarrow$ smaller Tput, bigger CPI, longer execution time, stalling logic (bigger)
    - solution #2: bypassing $\rightarrow$ bypassing logic (bigger)

    CONTROL HAZARDS: branch instructions
    condition is checked in EX and we already have two insns in IF and ID $\Rightarrow$ we have to "NOP them out"
    $\rightarrow$ smaller Tput, bigger CPI, longer execution time, flushing logic (bigger)

    alternative solutions: static insn scheduling to avoid hazards (reorder insns, insert NOPs),
    dynamic insn scheduling (out-of-order execution[1]), multithreading[1] (take independent insns from another program)

[1] not possible on MIPS, advanced topic, will not be tested (but might be on a quiz)

Recap: CPU Datapath



- **Optimizations are not free!** We have observed many trade-offs
  - single cycle datapath (large t_{CLK}, $Tput = 1$, $\text{CPI} = 1$)
  - multicycle datapath (small t_{CLK}, $Tput =$small, $\text{CPI} > 1$)
  - pipelined padapath (small t_{CLK}, $Tput = 1$, $\text{CPI} = 1$)
    NOTE: actual CPI and Tput worse due to stalls

    data dependencies (RAW, WAR, WAW, RAR) $\Rightarrow$ RAW is a DATA HAZARD!
    - solution $\#1$: stalling (NOPs, bubbles) $\rightarrow$ smaller Tput, bigger CPI, longer execution time, stalling logic (bigger)
    - solution $\#2$: bypassing $\rightarrow$ bypassing logic (bigger)

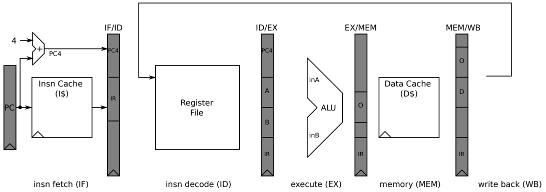
    CONTROL HAZARDS: branch instructions
    condition is checked in EX and we already have two insns in IF and ID $\Rightarrow$ we have to "NOP them out"
    $\rightarrow$ smaller Tput, bigger CPI, longer execution time, flushing logic (bigger)

    alternative solutions: static insn scheduling to avoid hazards (reorder insns, insert NOPs),
    dynamic insn scheduling (out-of-order execution[2]), multithreading[1] (take independent insns from another program)

- **Amdahl's law**: make the common case fast!
- **locality principle**: make the large and slow main memory appear as a small and fast cache

[2] not possible on MIPS, advanced topic, will not be tested (but might be on a quiz)

Pipelined Datapath - control hazards



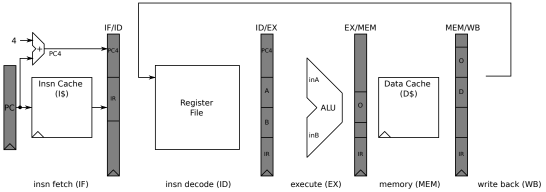
CONTROL HAZARDS: branch instructions

condition is checked in EX and we already have two insns in IF and ID $\Rightarrow$ we have to “NOP them out” (branch penalty)

$\rightarrow$ smaller Tput, bigger CPI, longer execution time, flushing logic (bigger)

goal: minimize branch penalty and maximize Tput $\Rightarrow$ branch prediction

Pipelined Datapath - control hazards



CONTROL HAZARDS: branch instructions

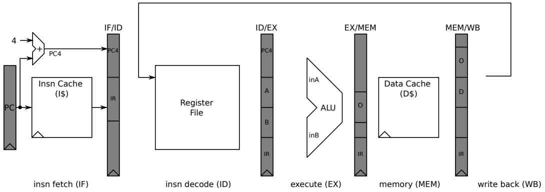
condition is checked in EX and we already have two insns in IF and ID $\Rightarrow$ we have to "NOP them out" (branch penalty)

$\rightarrow$ smaller Tput, bigger CPI, longer execution time, flushing logic (bigger)

goal: minimize branch penalty and maximize Tput $\Rightarrow$ branch prediction

- solution #1: assume not taken (described above)
- solution #2: fast branches: move resolving logic into ID stage (a lot of additional hardware, can only test simple conditions)

Pipelined Datapath - control hazards



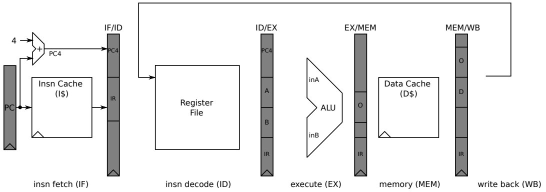
CONTROL HAZARDS: branch instructions
condition is checked in EX and we already have two insns in IF and ID $\Rightarrow$ we have to "NOP them out" (branch penalty)
$\rightarrow$ smaller Tput, bigger CPI, longer execution time, flushing logic (bigger)

goal: minimize branch penalty and maximize Tput $\Rightarrow$ branch prediction

- solution #1: assume not taken (described above)
- solution #2: fast branches: move resolving logic into ID stage (a lot of additional hardware, can only test simple conditions)
- solution #3: static prediction and delayed branch
  compiler moved two independent insns from before branch into the two[3] "branch slots" (always executed!)

[3] # of slots=# stages before EX = penalty

Pipelined Datapath - control hazards



CONTROL HAZARDS: branch instructions
condition is checked in EX and we already have two insns in IF and ID $\Rightarrow$ we have to “NOP them out” (branch penalty)
$\rightarrow$ smaller Tput, bigger CPI, longer execution time, flushing logic (bigger)

goal: minimize branch penalty and maximize Tput $\Rightarrow$ branch prediction

- solution #1: assume not taken (described above)
- solution #2: fast branches: move resolving logic into ID stage (a lot of additional hardware, can only test simple conditions)
- solution #3: static prediction and delayed branch
  compiler moved two independent insns from before branch into the two[4] “branch slots” (always executed!)
- solution #4: dynamic branch prediction (add some hardware to IF stage)
  example: branch target buffer (BTB) - predicts direction (taken, not taken) and target address
  if not stored or wrong prediction: penalty (minimize branch penalty and maximize accuracy)

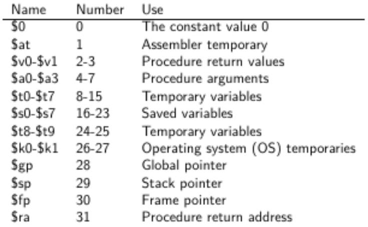
[4] # of slots=# stages before EX = penalty

Pipelined Datapath - change of PC

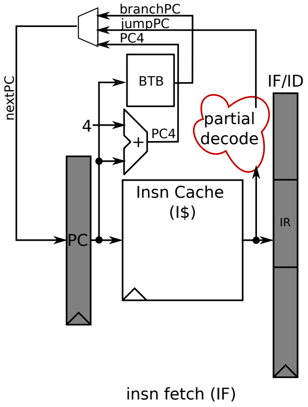
- Final piece of the puzzle:
  - to get the nextPC faster we add *partial decode* logic to the IF stage
  - this will take care of `jump` insns

    `j Label`: 26-bit Imm is left-shifted and concatenated to the MSB 4 bits of PC

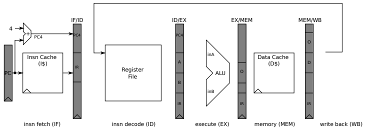
    `jr RA`: the new PC is the content of the RA register
- RegFile in more detail:
  - so far we assumed simplified RegFile, were we only mentioned R0 as holding 0
  - actual MIPS registers:



Name	Number	Use
$0	0	The constant value 0
$at	1	Assembler temporary
$v0-$v1	2-3	Procedure return values
$a0-$a3	4-7	Procedure arguments
$t0-$t7	8-15	Temporary variables
$s0-$s7	16-23	Saved variables
$t8-$t9	24-25	Temporary variables
$k0-$k1	26-27	Operating system (OS) temporaries
$gp	28	Global pointer
$sp	29	Stack pointer
$fp	30	Frame pointer
$ra	31	Procedure return address



Exceptions



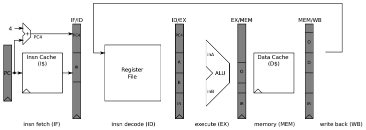
- so far: we assumed “normal” flow of insns
- Exceptions[5] will break the “normal flow”

- Some sources of exceptions:
  - IO exception: a device needs access to the CPU
  - OS, for example change to supervisor mode
  - ALU: for example divide by 0
  - misaligned memory access

  - undefined or illegal insn
  - hardware malfunctions

[5] we will not differentiate between faults, interrupts, traps

Exceptions



- so far: we assumed “normal” flow of insns
- Exceptions[6] will break the “normal flow”

- Some sources of exceptions:
  - IO exception: a device needs access to the CPU
  - OS, for example change to supervisor mode
  - ALU: for example divide by 0
  - misaligned memory access

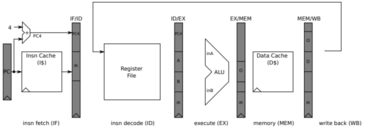
  $\Rightarrow$ resume normal insn flow after exception is handled (fix and restart)

  - undefined or illegal insn
  - hardware malfunctions

  $\Rightarrow$ must terminate

[6] we will not differentiate between faults, interrupts, traps

Exceptions



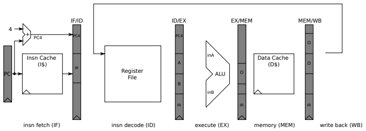
- What happens in CPU when exception occurs:

1. stop fetching
2. decide between:
   - draining the pipeline (complete the insns that are already in the pipeline)
   - flushing the pipeline (discard all insns in the pipeline)
3. save state (the the stack[7] - recall stack pointer)
4. invoke the handler (a procedure stored in main memory on a specified[8] location)
5. resume execution (the procedure will stop with jr RA) - if possible!

[7] allows nesting of exceptions

[8] similar as RegFile in not entirely available to the user, main memory is also not entirely available

Exceptions



- What happens in CPU when exception occurs:

1. stop fetching
2. decide between:
   - draining the pipeline (complete the insns that are already in the pipeline)
   - flushing the pipeline (discard all insns in the pipeline)
3. save state (the the stack[9] - recall stack pointer)
4. invoke the handler (a procedure stored in main memory on a specified[10] location)
5. resume execution (the procedure will stop with jr RA) - if possible!

- Other concerns:
  - how to ensure transparency?
  - where to get the address of the handler procedure?
  - how to notify the external source (such as IO controller)?

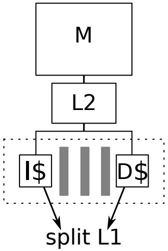
[9] allows nesting of exceptions

[10] similar as RegFile in not entirely available to the user, main memory is also not entirely available

Recap: Memory Hierarchy

- **from the view-point of the memory:** in a given time-frame, certain addresses are more likely then others !
- the sequence of addresses for a given program is not random
- different programs will form different sequences of addresses and will exhibit different degrees of locality
  - usual: $A(i+1) = A(i) + 1$
  - deviations: jumps, branches, swapping between insn and data access
  - we are likely to find repeating insns (e.g. loops) and data (e.g. arrays)
- **temporal locality:** recently accessed will likely be accessed in the near future
- **spatial locality:** next address will be close to the current one
- **GOAL:** optimize average access time by making a big, slow main memory look like the small, fast L1 cache (we have seen a **split I$ and D$**, we will consider higher levels of hierarchy as unified cache)

hierarchy	L1		L2		M
coherence	L1	$\subset$	L2	$\subset$	M
size	L1	$<$	L2	$<$	M
speed	L1	$>$	L2	$>$	M


M
L2
I$
D$
split L1