Review

- Last Class:
  - Control hazard
  - Multi-cycle pipelines
- Today's class:
  - Register renaming
  - Branch prediction
- Announcement and reminder
  - In person class starts next Monday (SENSQ 5313), Jan 31st
  - No recording
  - Only approved remote participation (by the instructor and DRS)

(1)

Dealing with Exceptions

- Exceptions (aka interrupts) are just another form of control hazard. Exceptions arise from
  - R-type arithmetic overflow
  - Trying to execute an undefined instruction
  - An I/O device request
  - An OS service request (e.g., a page fault, TLB exception)
  - A hardware malfunction
- The pipeline has to stop executing the offending instruction in midstream, let all prior instructions complete, flush all following instructions, set a register to show the cause of the exception, save the address of the offending instruction, and then jump to a prearranged address (the address of the exception handler code)
- The software (OS) looks at the cause of the exception and "deals" with it

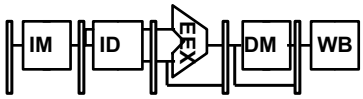
(2)

Two Types of Exceptions

- Interrupts – asynchronous to program execution
  - caused by external events
  - may be handled between instructions, so can let the instructions currently active in the pipeline *complete* before passing control to the OS interrupt handler
  - simply suspend and resume user program
- Traps – synchronous to program execution
  - caused by internal events
  - condition must be remedied by the trap handler for that instruction, so must stop the offending instruction *midstream* in the pipeline and pass control to the OS trap handler
  - the offending instruction may be retried (or simulated by the OS) and the program may continue or it may be aborted

(3)

Where in the Pipeline Exceptions Occur

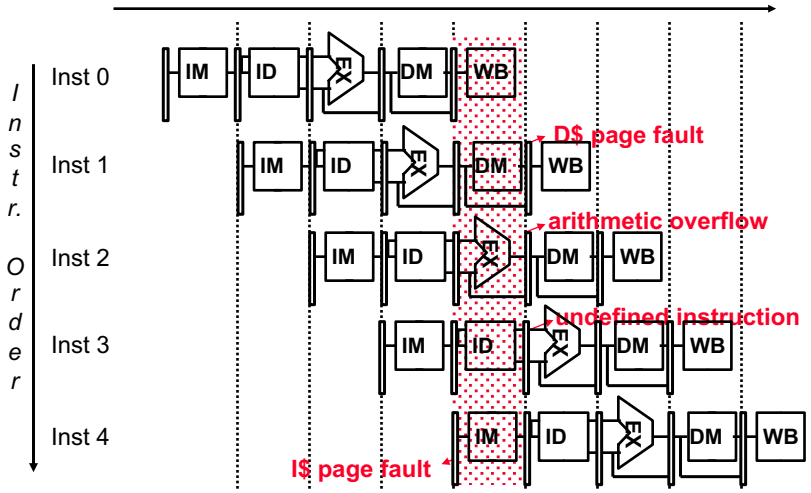


	Stage(s)?	Synchronous?
• Arithmetic overflow	EX	yes
• Undefined instruction	ID	yes
• TLB or page fault	IF, MEM	yes
• I/O service request	any	no
• Hardware malfunction	any	no

Be aware that multiple exceptions can occur simultaneously in a *single* clock cycle!

(4)

Multiple Simultaneous Exceptions



- Hardware sorts the exceptions so that the earliest instruction (D$ page fault) is the one interrupted first

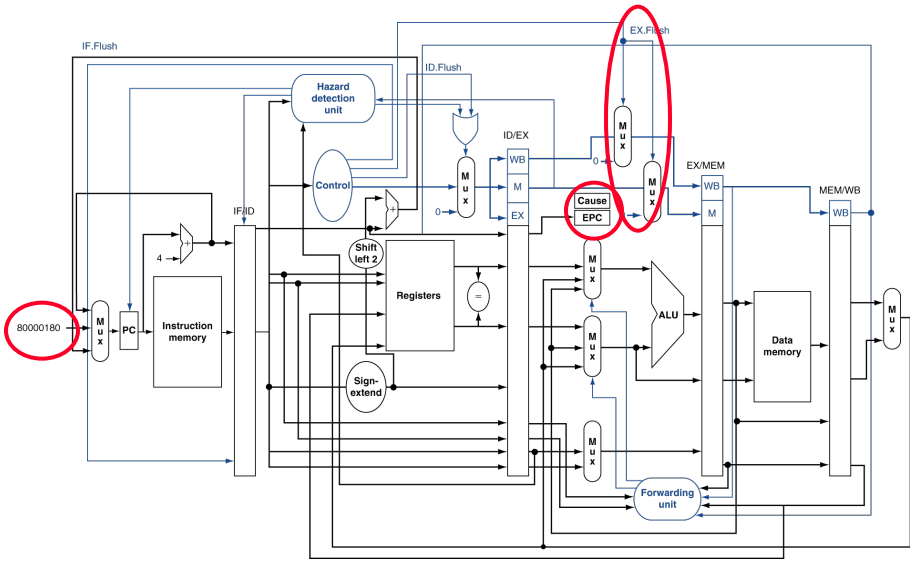
(5)

Additions to MIPS to Handle Exceptions

- Cause register (records exceptions) – hardware to record in Cause the exceptions and a signal to control writes to it (`CauseWrite`)
- EPC register (records the addresses of the offending instructions) – hardware to record in EPC the address of the offending instruction and a signal to control writes to it (`EPCWrite`)
  - Exception software must match exception to instruction
- A way to load the PC with the address of the exception handler
  - Expand the PC input mux where the new input is hardwired to the exception handler address - (e.g., 8000 0180_{hex} for arithmetic overflow, 8000 0000_{hex} for undefined instruction)
- A way to flush offending instruction and the ones that follow it

(6)

Pipeline with Exception Extensions



(7)

Instruction set design and pipelining (C.4)

- Variable instruction length and execution time leads to
  - imbalance among stages,
  - complicate hazard detection and precise exceptions
- Caches have similar effects (imbalance pipes)
  - may freeze the entire pipeline on a cache miss
- Complex addressing modes
  - may change register values
  - may require multiple memory access
- Implicitly set condition codes complicates pipeline control hazards

(9)

Review

- Performance evaluation
  - MIPS rate
  - CPU time equations
  - CPI, IPC
- MIPS ISA
  - CISC, RISC, RISC-V
  - Different types of instructions
  - Addressing modes
  - Datapath and control path of different types of instructions
- Pipelined
  - Single cycle processor → CCT problem
  - Pipelined execution → improved ILP
  - Hazards hurt performance
  - Multi-cycle pipelines

(10)

Chapter 3: Instruction Level Parallelism (ILP) and its exploitation

- Pipeline CPI = Ideal pipeline CPI + stalls due to hazards
  - invisible to programmer (unlike process level parallelism)
- ILP: overlap execution of unrelated instructions
  - invisible to programmer (unlike process level parallelism)
- Parallelism within a basic block is limited (a branch every 3-6 instructions)
  - Hence, must explore ILP across basic blocks
- May explore loop level parallelism (fake control dependences) through
  - Loop unrolling (static, compiler based solution)
  - Using vector processors or SIMD architectures
  - Dynamic loop unrolling
- The main challenge is overcoming data and control dependencies
- Further improving performance through ILP in today's processors is difficult (why?)

(11)

Types of dependences

- **True data dependences:** may cause RAW hazards.
  - Instruction *Q* uses data produced by instruction *P* or by an instruction which is data dependent on *P.*
  - dependences are properties of programs, while hazards are properties of architectures.
  - easy to determine for registers but hard to determine for memory locations since addresses are computed dynamically

    EX: is 100(R1) the same location as 200(R2) or even 100(R1)?

- **False (Name) dependences:** two instructions use the same name but do not exchange data (no data dependency**)**
  - **Anti-dependence**: Instruction *P* reads from a register (or memory) followed by instruction *Q* writing to that register (or memory).

    May cause WAR hazards.
  - **Output dependence**: Instructions *P* and *Q* write to the same location.

    May cause WAW hazards.

(12)

Overview of Dependence Analysis

- To what extent can the compiler (or the datapath) reorder instructions? Are there execution-order constraints?

original	possible?	possible?

`instr 1`
`instr 2`
consecutive
	`instr 2`
	`instr 1`
	consecutive
		`instr 1` and `instr 2`
		simultaneous

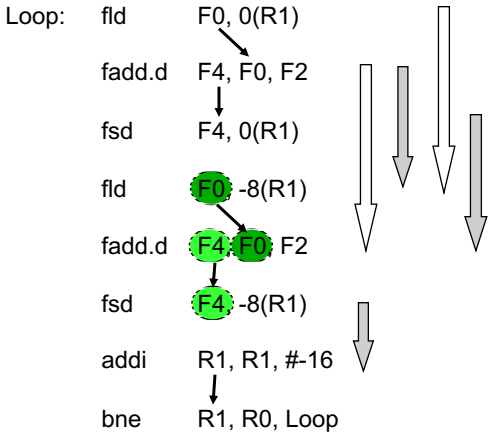
- Instruction dependencies imply that reordering instructions is not possible
  - true dependence (or, data dep., flow dep.) (cannot reorder)

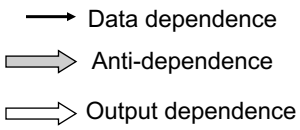
    `a = .`
    `. = a` **RAW, read after write**
  - anti-dependence (renaming allows reordering)

    `. = a`
    `a = .` **WAR, write after read**
  - output dependence (renaming allows reordering)

    `a = .`
    `a = .` **WAW, write after write**

(13)

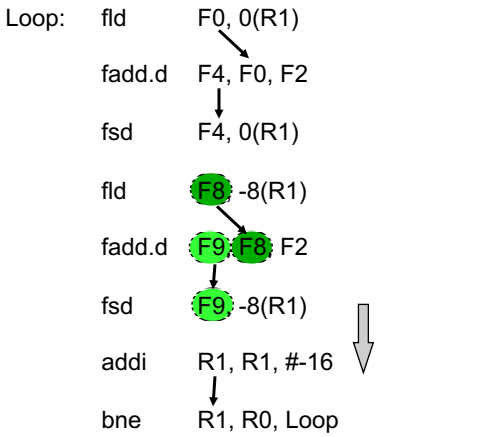


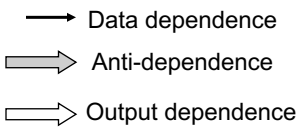


How can you remove name dependences?

Rename the dependent uses of F0 and F4

(14)





R1 is also renamed and the R1 in bne and next iteration is updated to use the renamed register

(16)

Register Renaming

- Can use register renaming to eliminate (WAW, WAR) (register) data dependencies – conceptually write each register once
  - Removes **false** dependences (WAW and WAR)
  - Leaves **true** dependences (RAW) intact
- "Architected" vs "Physical" registers
  - Architected (ISA) register names: `$t0,$s1,$s1,$s2,` etc
  - Physical register names: `p1,p2,p3,p4,p5,p6,p7`
- Need two hardware structures to enable renaming
  - A Map Table showing the architected register that the physical register is currently "impersonating"
  - A Free List of physical registers not currently in use
- When can a physical register be put back on the Free List?

(18)

Which Register to Free at Commit ?

- The over-written (physical) register can be freed at Commit (i.e., added back to the Free List), so we have to keep track of it during Rename
- We also need to keep track of the over-written (physical) register so that it can be restored in the Map Table on a recovery from mis-predicted branches and recovery from exceptions

(19)

Renaming Example: Initial State

```
lw    $t0,0($s1)
addu  $t0,$t0,$s2
sw    $t0,0($s1)
sub   $t0,$s1,$s2
addi  $s1,$s1,-4
bne   $s1,$0,lp
```

$s1	**p1**
$s2	**p2**
$t0	**p3**

Map Table

p4
p5
p6
p7
p8

Free List

(20)

Renaming Example: `lw` Renaming

Over-written Reg

```
lw    $t0,0($s1)  ⟶
addu  $t0,$t0,$s2
sw    $t0,0($s1)
sub   $t0,$s1,$s2
addi  $s1,$s1,-4
bne   $s1,$0,lp
```

$s1	**p1**
$s2	**p2**
$t0	**p3**

Map Table

p4
p5
p6
p7
p8

Free List

(21)

Renaming Example: `lw` Renaming

Over-written Reg

```
lw    $t0,0($s1)  ⟶  lw    p4,0(p1)
addu  $t0,$t0,$s2
sw    $t0,0($s1)
sub   $t0,$s1,$s2
addi  $s1,$s1,-4
bne   $s1,$0,lp
```

$s1	p1
$s2	p2
$t0	p3

Map Table

p4
p5
p6
p7
p8

Free List

(22)

Renaming Example: `lw` Renaming

Over-written Reg

```
lw    $t0,0($s1)  ⟶  lw    p4,0(p1)
addu  $t0,$t0,$s2
sw    $t0,0($s1)
sub   $t0,$s1,$s2
addi  $s1,$s1,-4
bne   $s1,$0,lp
```

$s1	p1
$s2	p2
$t0	p4

Map Table

p5
p6
p7
p8

Free List

(23)

Renaming Example: `lw` Renaming

Over-written Reg

```
lw    $t0,0($s1)  ⟶  lw    p4,0(p1)      [p3]
addu  $t0,$t0,$s2
sw    $t0,0($s1)
sub   $t0,$s1,$s2
addi  $s1,$s1,-4
bne   $s1,$0,lp
```

$s1	p1
$s2	p2
$t0	p4

p3

Map Table

p5
p6
p7
p8

Free List

(24)

Renaming Example: `addu` Renaming

Over-written Reg

```
lw    $t0,0($s1)
addu  $t0,$t0,$s2  ⟶  addu  p5,p4,p2
sw    $t0,0($s1)
sub   $t0,$s1,$s2
addi  $s1,$s1,-4
bne   $s1,$0,lp
```

$s1	p1
$s2	p2
$t0	p4

p3

Map Table

p5
p6
p7
p8

Free List

(25)

Renaming Example: `addu` Renaming

Over-written Reg

```
lw    $t0,0($s1)        lw    p4,0(p1)      [p3]
addu  $t0,$t0,$s2  ⟶    addu  p5,p4,p2      [p4]
sw    $t0,0($s1)
sub   $t0,$s1,$s2
addi  $s1,$s1,-4
bne   $s1,$0,lp
```

$s1	**p1**
$s2	**p2**
$t0	**p5**

p4 p3

Map Table

p6
p7
p8

Free List

(26)

Renaming Example: `sw` Renaming

Over-written Reg

```
lw    $t0,0($s1)        lw    p4,0(p1)      [p3]
addu  $t0,$t0,$s2       addu  p5,p4,p2      [p4]
sw    $t0,0($s1)  ⟶     sw    p5,0(p1)
sub   $t0,$s1,$s2
addi  $s1,$s1,-4
bne   $s1,$0,lp
```

$s1	**p1**
$s2	**p2**
$t0	**p5**

p4 p3

Map Table

p6
p7
p8

Free List

(27)

Renaming Example: `sub` Renaming

				Over-written Reg
`lw`	`$t0,0($s1)`	`lw`	`p4,0(p1)`	`[p3]`
`addu`	`$t0,$t0,$s2`	`addu`	`p5,p4,p2`	`[p4]`
`sw`	`$t0,0($s1)`	`sw`	`p5,0(p1)`	
`sub`	`$t0,$s1,$s2` →	`sub`	`p6,p1,p2`	
`addi`	`$s1,$s1,-4`			
`bne`	`$s1,$0,lp`			

$s1	**p1**
$s2	**p2**
$t0	**p5**

p4 p3

p6
p7
p8

Map Table **Free List**

(28)

Renaming Example: `sub` Renaming

				Over-written Reg
`lw`	`$t0,0($s1)`	`lw`	`p4,0(p1)`	`[p3]`
`addu`	`$t0,$t0,$s2`	`addu`	`p5,p4,p2`	`[p4]`
`sw`	`$t0,0($s1)`	`sw`	`p5,0(p1)`	
`sub`	`$t0,$s1,$s2` →	`sub`	`p6,p1,p2`	`[p5]`
`addi`	`$s1,$s1,-4`			
`bne`	`$s1,$0,lp`			

$s1	**p1**
$s2	**p2**
$t0	**p6**

p5 p4 p3

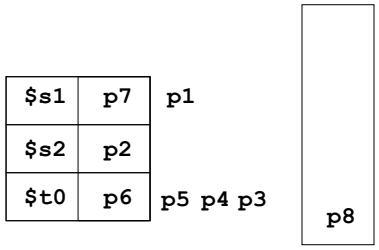
p7
p8

Map Table **Free List**

(29)

Renaming Example: `addi` Renaming

				Over-written Reg
`lw`	`$t0,0($s1)`	`lw`	`p4,0(p1)`	`[p3]`
`addu`	`$t0,$t0,$s2`	`addu`	`p5,p4,p2`	`[p4]`
`sw`	`$t0,0($s1)`	`sw`	`p5,0(p1)`	
`sub`	`$t0,$s1,$s2`	`sub`	`p6,p1,p2`	`[p5]`
`addi`	`$s1,$s1,-4` →	`addi`	`p7,p1,-4`	`[p1]`
`bne`	`$s1,$0,lp`			

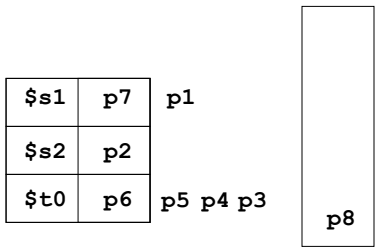


Map Table **Free List**

(30)

Renaming Example: `bne` Renaming

				Over-written Reg
`lw`	`$t0,0($s1)`	`lw`	`p4,0(p1)`	`[p3]`
`addu`	`$t0,$t0,$s2`	`addu`	`p5,p4,p2`	`[p4]`
`sw`	`$t0,0($s1)`	`sw`	`p5,0(p1)`	
`sub`	`$t0,$s1,$s2`	`sub`	`p6,p1,p2`	`[p5]`
`addi`	`$s1,$s1,-4`	`addi`	`p7,p1,-4`	`[p1]`
`bne`	`$s1,$0,lp` →	`bne`	`p7,p0,lp`	



Map Table **Free List**

(31)

Register renaming vs register allocation

- Register renaming
  - Architecture registers to physical registers (limited)
  - Resolve anti dependences
  - Map table to track the over-written history for potential restoring
  - Free list holding available physical registers.
  - **Hardware, in the ROB**
- Register allocation
  - Architecture registers (limited with a number)
  - Minimize the number of register spilling to memory
  - Liveness range analysis
  - Graph coloring algorithm
  - **Compiler back end, data flow analysis**

(32)

Loop carried dependences

```
For i=1,100
  a[i+1] = a[i]  + c[i]  ;
```

There is a loop carried dependence since the statement in an iteration depends on an earlier iteration.

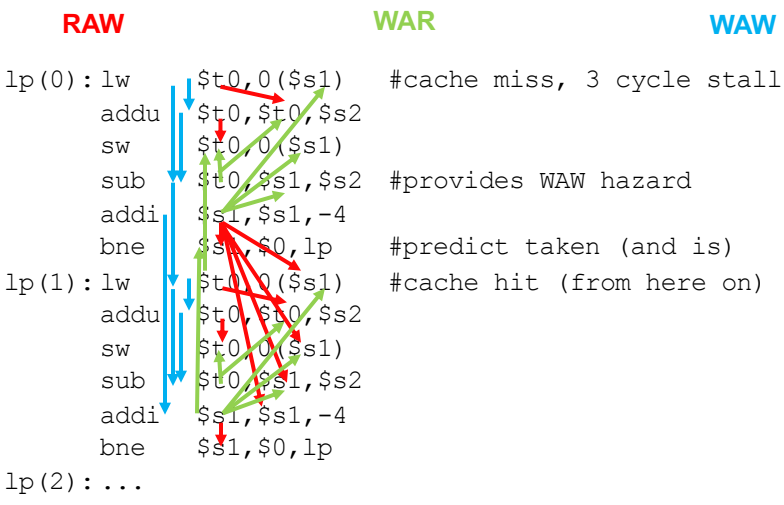
```
For i=1,100
  a[i] = a[i]  + s ;
```

There is no loop carried dependence

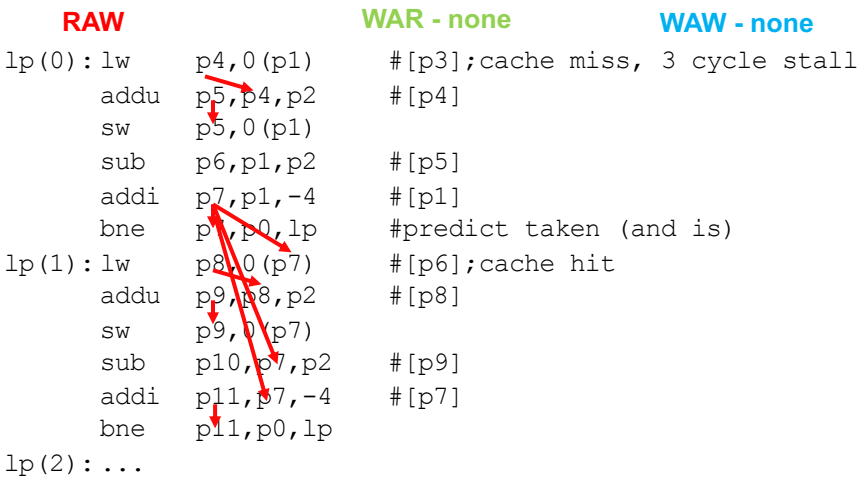
- The iterations of a loop can be executed in parallel if there is no *Loop carried dependences*.

(33)

Code Example



Code Example After Renaming



- As promised, renaming eliminated false data dependencies (WAW, WAR) and left true data dependencies (RAW) intact