Department of Electrical Engineering, Feng-Chia University

Chapter 4
The Processor (Part 4)

王振傑 (Chen-Chieh Wang)
ccwang@mail.ee.ncku.edu.tw

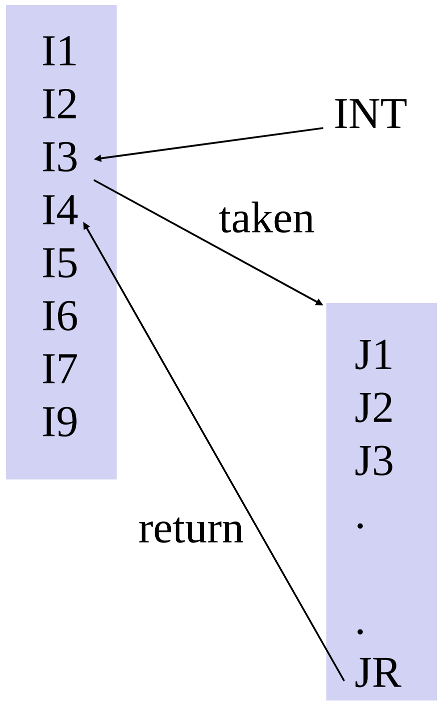
Department of Electrical Engineering, Feng-Chia University

Outline

4.9 Exceptions
4.10 Parallelism and Advanced ILP
4.14 Concluding Remarks

Exception

- What is an exception ?
  - Unexpected events
  - Change PC
- Interrupt
  - External, usually
  - I/O devices wish to communicate with CPU
- Exception
  - Internal or external



Vector interrupt vs. status register

- Use a status register to hold exception causes
  - Single entry point: 0x8000_0180
- Use vectored interrupt
  - The address to which the control is transferred is determined by the cause of the exception

Exception type	Exception vector address (in hex)
undefined instruction	0x8000_0000
arithmetic overflow	0x8000_0180

Additional registers

- Exception Program Counter (EPC):
  - a 32-bit register used to hold the address of the affected instruction
- Cause:
  - a register used to record the cause of the exception
    - 0: undefined instruction (can not recognize the opcode)
    - 1: arithmetic overflow

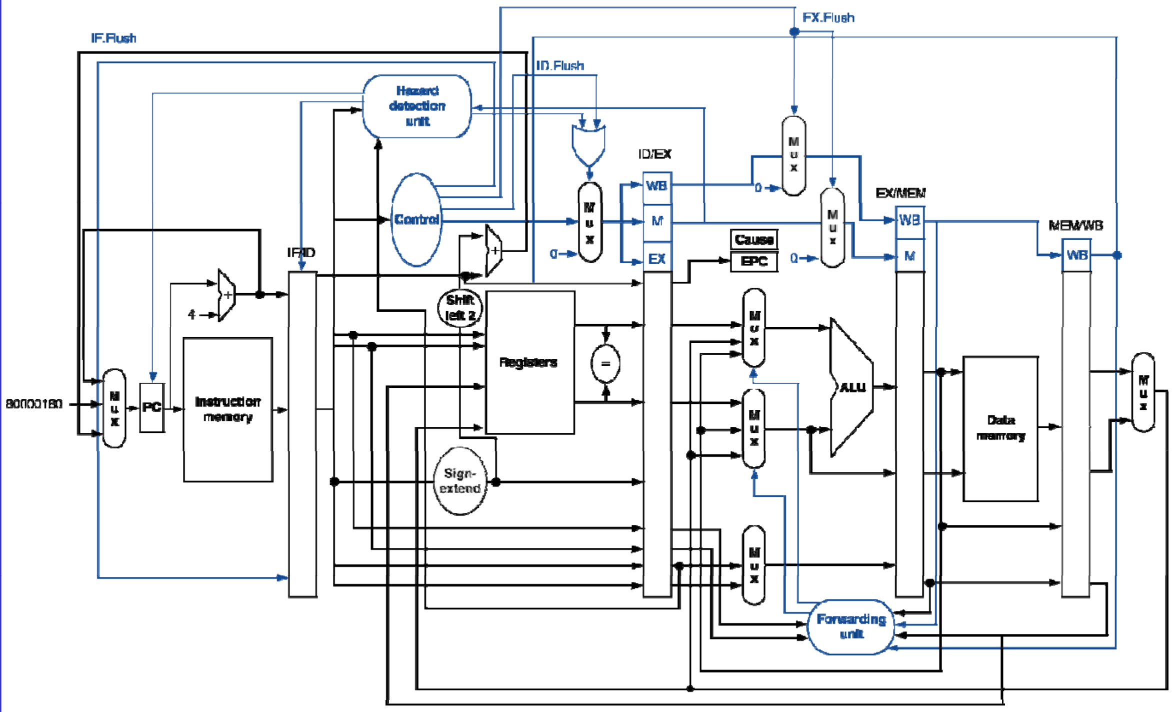
Exceptions in a Pipeline

- Another form of control hazard
- Consider overflow on add in EX stage
  - Prevent $1 from being clobbered
  - Complete previous instructions
  - Flush add and subsequent instructions
  - Set Cause and EPC register values
  - Transfer control to handler

```
40      sub  $11, $2, $4
44      and  $12, $2, $5
48      or   $13, $2, $6
4C      add  $1,  $2, $1
50      slt  $15, $6, $7
54      lw   $16, 50($7)
…
```

- Similar to mispredicted branch
  - Use much of the same hardware

Pipeline with Exceptions



Exception Properties

- Restartable exceptions
  - Pipeline can flush the instruction
  - Handler executes, then returns to the instruction
    - Refetched and executed from scratch
- PC saved in EPC register
  - Identifies causing instruction
  - Actually PC + 4 is saved
    - Handler must adjust

Exception Example

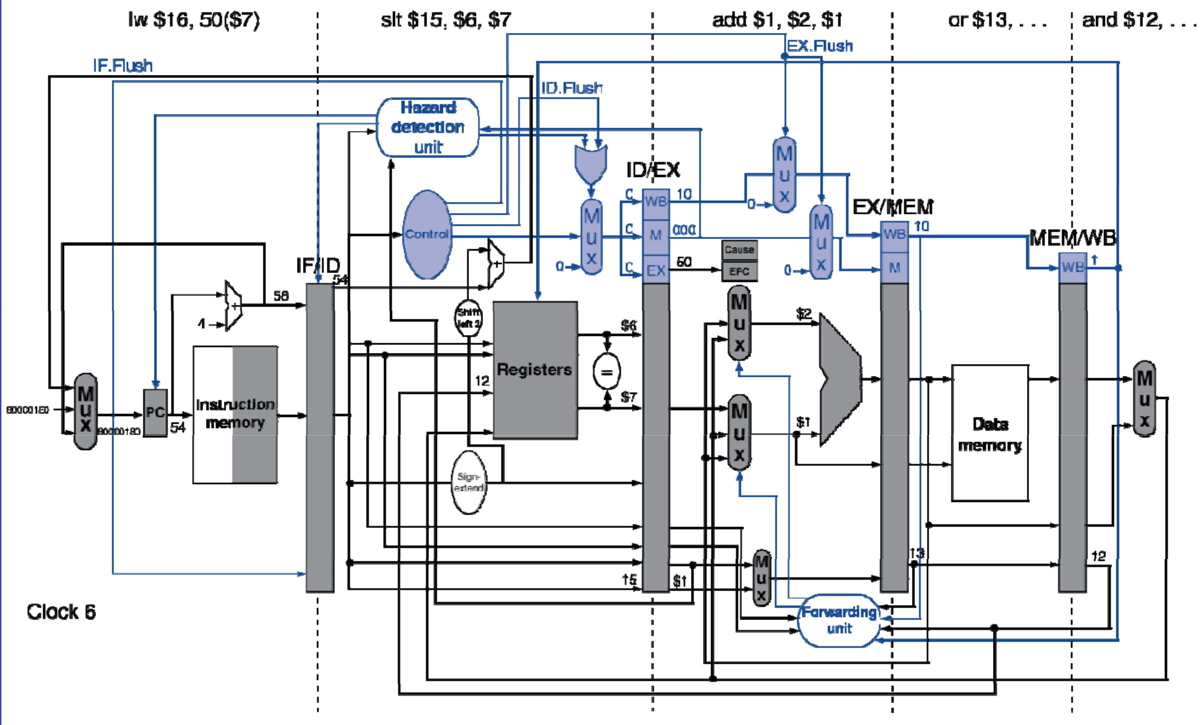
- Exception on add in

```
40     sub  $11, $2, $4
44     and  $12, $2, $5
48     or   $13, $2, $6
4C     add  $1,  $2, $1
50     slt  $15, $6, $7
54     lw   $16, 50($7)
…
```

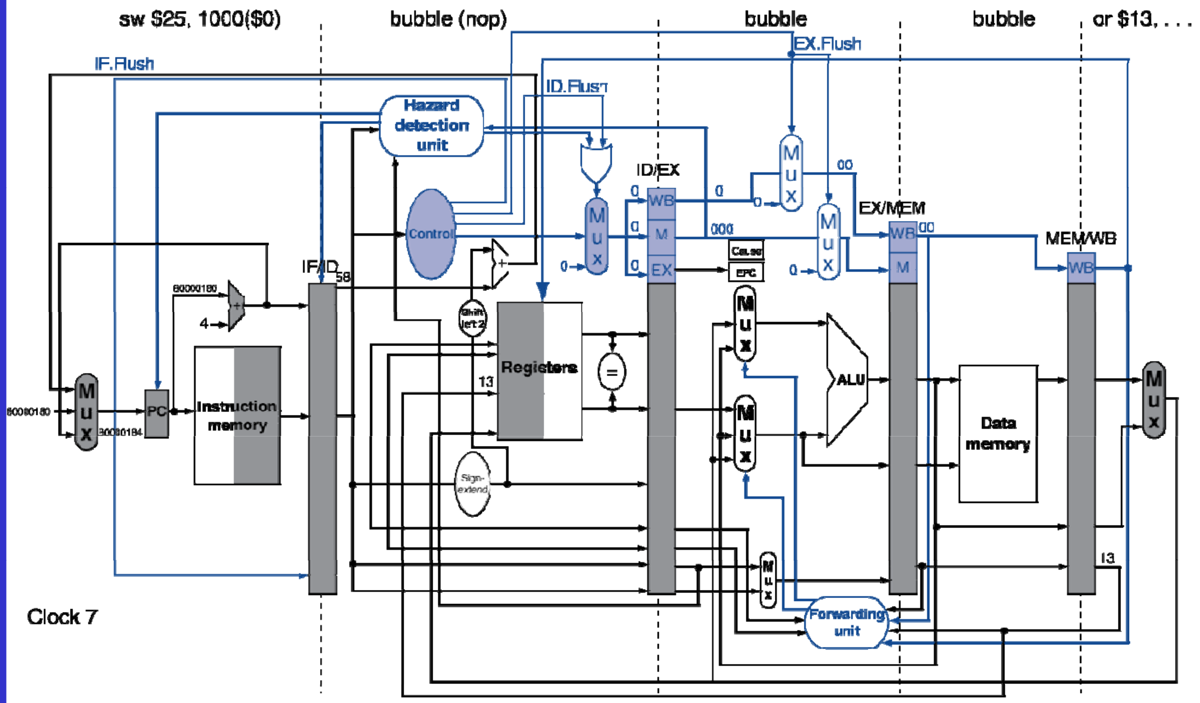
- Handler

```
80000180   sw   $25, 1000($0)
80000184   sw   $26, 1004($0)
…
```

Exception Example



Exception Example



Multiple Exceptions

- Pipelining overlaps multiple instructions
  - Could have multiple exceptions at once
- Simple approach: deal with exception from earliest instruction
  - Flush subsequent instructions
  - "Precise" exceptions
- In complex pipelines
  - Multiple instructions issued per cycle
  - Out-of-order completion
  - Maintaining precise exceptions is difficult!

Imprecise Exceptions

- Just stop pipeline and save state
  - Including exception cause(s)
- Let the handler work out
  - Which instruction(s) had exceptions
  - Which to complete or flush
    - May require “manual” completion
- Simplifies hardware, but more complex handler software
- Not feasible for complex multiple-issue out-of-order pipelines

Outline

4.9 Exceptions

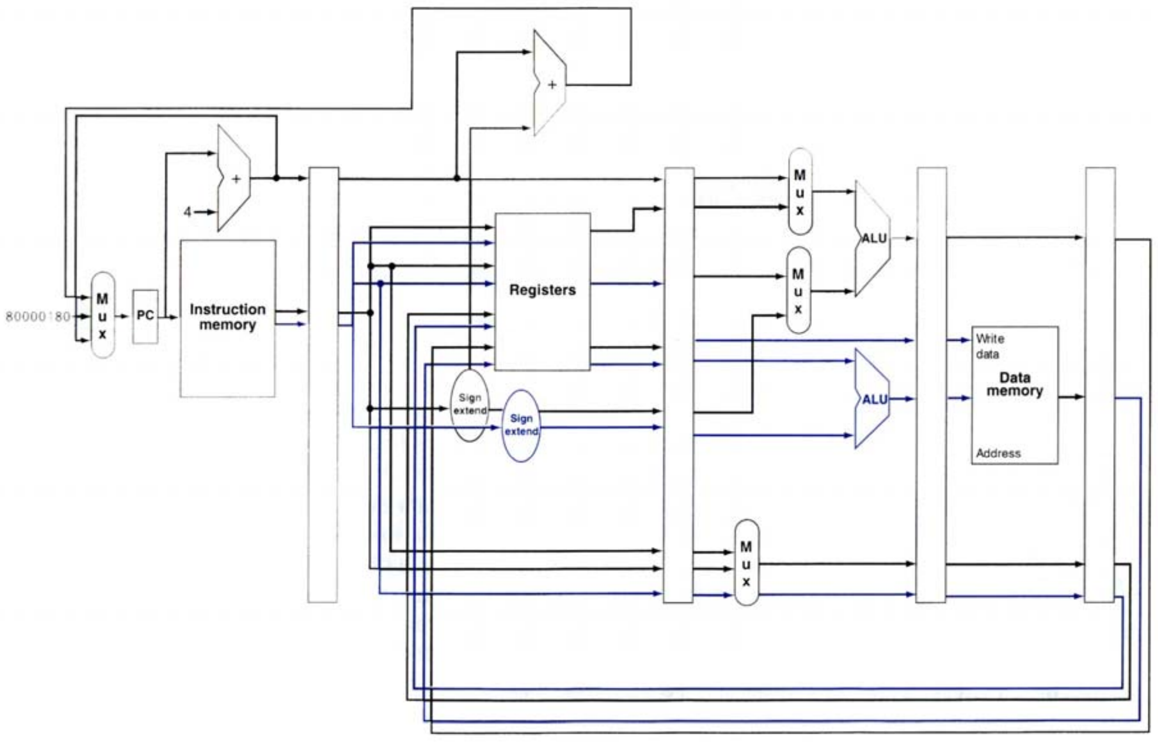
4.10 Parallelism and Advanced ILP

4.14 Concluding Remarks

Instruction Level Parallelism (ILP)

- Two methods to increase ILP
  - Increase the pipeline depth
    - More operations being overlapped
  - Multiple issue
    - Static multiple issue: determined at compile time
      - VLIW: Very Long Instruction Word (relies more on compiler technology)
      - EPIC: Explicitly Parallel Instruction Computer
    - Dynamic multiple issue: determined during execution
      - Superscalar
- All modern processors are superscalar and issue multiple instructions usually with some limitations

A static two-issue datapath



Hazards in the Dual-Issue MIPS

- More instructions executing in parallel
- EX data hazard
  - Forwarding avoided stalls with single-issue
  - Now can't use ALU result in load/store in same packet

    -
      ```
      add  $t0, $s0, $s1
      load $s2, 0($t0)
      ```
    - Split into two packets, effectively a stall
- Load-use hazard
  - Still one cycle use latency, but now two instructions
- More aggressive scheduling required

Scheduling Example

- Schedule this for dual-issue MIPS

```
Loop: lw   $t0, 0($s1)       # $t0=array element
      addu $t0, $t0, $s2     # add scalar in $s2
      sw   $t0, 0($s1)       # store result
      addi $s1, $s1,–4       # decrement pointer
      bne  $s1, $zero, Loop # branch $s1!=0
```

	ALU/branch	Load/store	cycle
Loop:	nop	lw $t0, 0($s1)	1
	addi $s1, $s1,–4	nop	2
	addu $t0, $t0, $s2	nop	3
	bne $s1, $zero, Loop	sw $t0, 4($s1)	4

IPC = 5/4 = 1.25 (c.f. peak IPC = 2)

Loop Unrolling

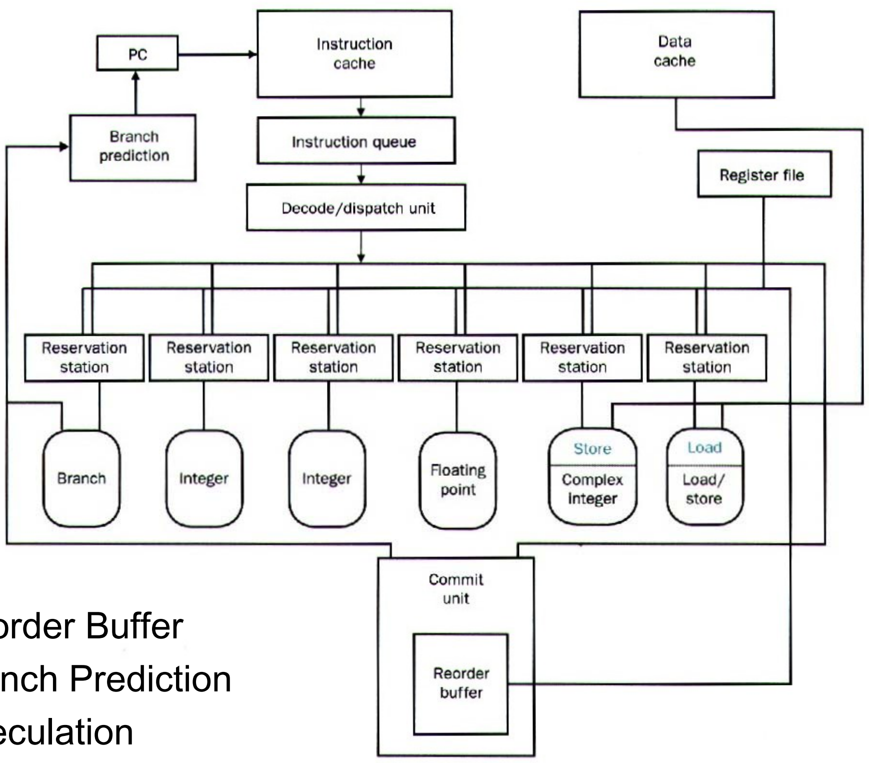
- Replicate loop body to expose more parallelism
  - Reduces loop-control overhead
- Use different registers per replication
  - Called "register renaming"
  - Avoid loop-carried "anti-dependencies"
    - Store followed by a load of the same register
    - Aka "name dependence"
      - Reuse of a register name

Loop Unrolling Example

	ALU/branch	Load/store	cycle
`Loop:`	`addi $s1, $s1,–16`	`lw  $t0, 0($s1)`	1
	`nop`	`lw  $t1, 12($s1)`	2
	`addu $t0, $t0, $s2`	`lw  $t2, 8($s1)`	3
	`addu $t1, $t1, $s2`	`lw  $t3, 4($s1)`	4
	`addu $t2, $t2, $s2`	`sw  $t0, 16($s1)`	5
	`addu $t3, $t4, $s2`	`sw  $t1, 12($s1)`	6
	`nop`	`sw  $t2, 8($s1)`	7
	`bne  $s1, $zero, Loop`	`sw  $t3, 4($s1)`	8

- IPC = 14/8 = 1.75
  - Closer to 2, but at cost of registers and code size

Out-of-order execution



- Reorder Buffer
- Branch Prediction
- Speculation

Register Renaming

- Reservation stations and reorder buffer effectively provide register renaming
- On instruction issue to reservation station
  - If operand is available in register file or reorder buffer
    - Copied to reservation station
    - No longer required in the register; can be overwritten
  - If operand is not yet available
    - It will be provided to the reservation station by a function unit
    - Register update may not be required

Speculation

- "Guess" what to do with an instruction
  - Start operation as soon as possible
  - Check whether guess was right
    - If so, complete the operation
    - If not, roll-back and do the right thing
- Common to static and dynamic multiple issue
- Examples
  - Speculate on branch outcome
    - Roll back if path taken is different
  - Speculate on load
    - Roll back if location is updated

Compiler/Hardware Speculation

- Compiler can reorder instructions
  - e.g., move load before branch
  - Can include "fix-up" instructions to recover from incorrect guess
- Hardware can look ahead for instructions to execute
  - Buffer results until it determines they are actually needed
  - Flush buffers on incorrect speculation

Speculation and Exceptions

- What if exception occurs on a speculatively executed instruction?
  - e.g., speculative load before null-pointer check
- Static speculation
  - Can add ISA support for deferring exceptions
- Dynamic speculation
  - Can buffer exceptions until instruction completion (which may not occur)

Speculation

- Predict branch and continue issuing
  - Don't commit until branch outcome determined
- Load speculation
  - Avoid load and cache miss delay
    - Predict the effective address
    - Predict loaded value
    - Load before completing outstanding stores
    - Bypass stored values to load unit
  - Don't commit load until speculation cleared

Why Do Dynamic Scheduling?

- Why not just let the compiler schedule code?
- Not all stalls are predicable
  - e.g., cache misses
- Can’t always schedule around branches
  - Branch outcome is dynamically determined
- Different implementations of an ISA have different latencies and hazards

Does Multiple Issue Work?

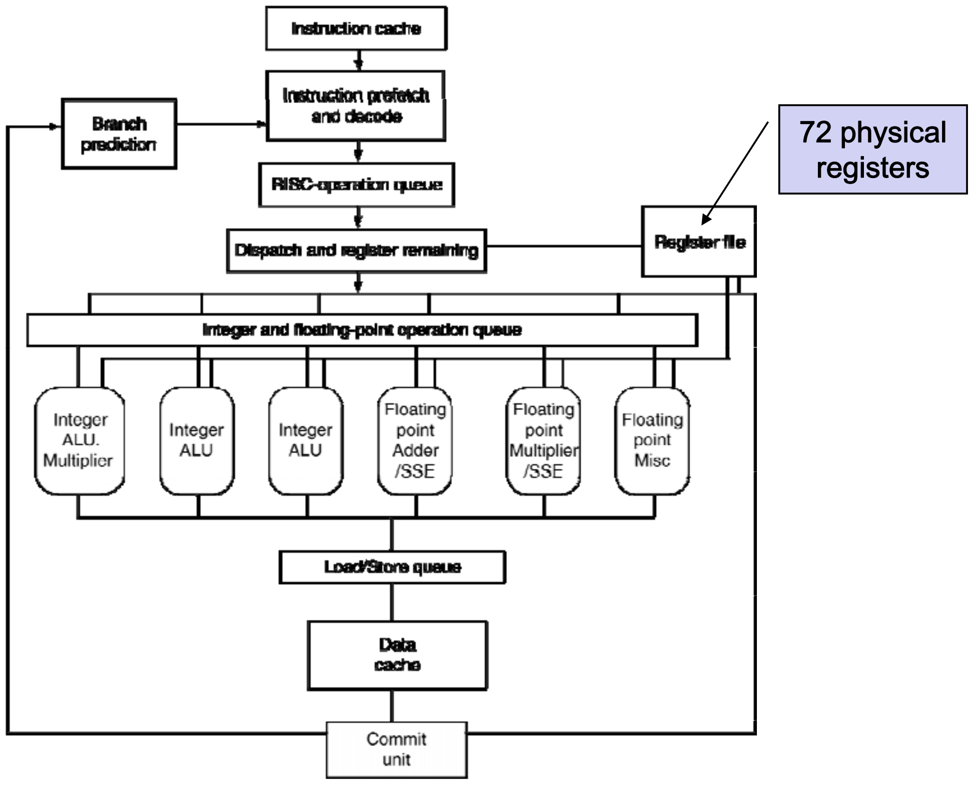
- Yes, but not as much as we’d like
- Programs have real dependencies that limit ILP
- Some dependencies are hard to eliminate
  - e.g., pointer aliasing
- Some parallelism is hard to expose
  - Limited window size during instruction issue
- Memory delays and limited bandwidth
  - Hard to keep pipelines full
- Speculation can help if done well

Power Efficiency

- Complexity of dynamic scheduling and speculations requires power
- Multiple simpler cores may be better

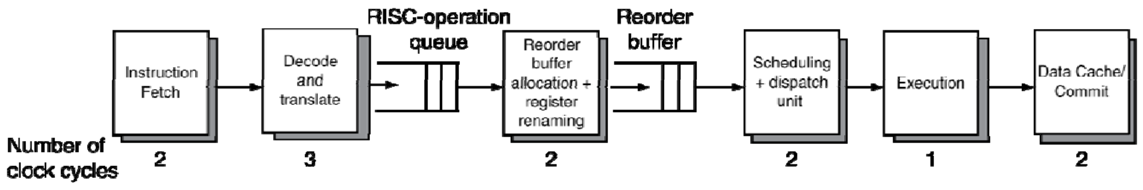
Microprocessor	Year	Clock Rate	Pipeline Stages	Issue width	Out-of-order/ Speculation	Cores	Power
i486	1989	25MHz	5	1	No	1	5W
Pentium	1993	66MHz	5	2	No	1	10W
Pentium Pro	1997	200MHz	10	3	Yes	1	29W
P4 Willamette	2001	2,000MHz	22	3	Yes	1	75W
P4 Prescott	2004	3,600MHz	31	3	Yes	1	103W
Core	2006	2,930MHz	14	4	Yes	2	75W
UltraSparc III	2003	1,950MHz	14	4	No	1	90W
UltraSparc T1	2005	1,200MHz	6	1	No	8	70W

The Opteron X4 Microarchitecture



The Opteron X4 Pipeline Flow

- For integer operations



  - FP is 5 stages longer
  - Up to 106 RISC-ops in progress
- Bottlenecks
  - Complex instructions with long dependencies
  - Branch mispredictions
  - Memory access delays

Outline

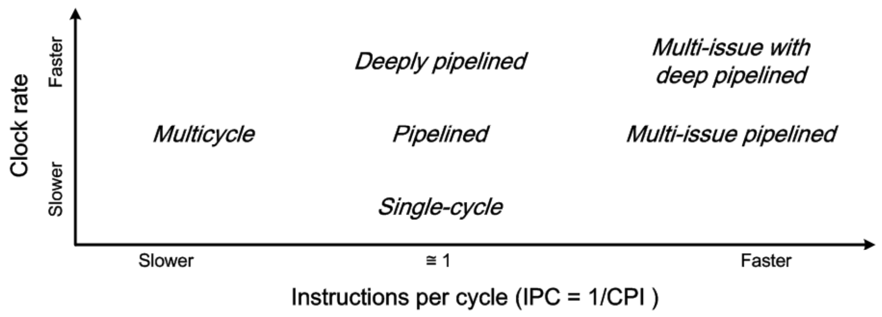
4.9 Exceptions
4.10 Parallelism and Advanced ILP
4.14 Concluding Remarks

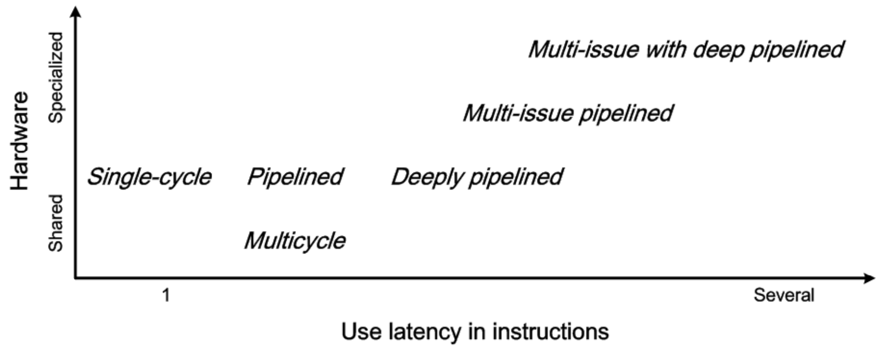
Concluding Remarks

- ISA influences design of datapath and control
- Datapath and control influence design of ISA
- Pipelining improves instruction throughput using parallelism
  - More instructions completed per second
  - Latency for each instruction not reduced
- Hazards: structural, data, control
- Multiple issue and dynamic scheduling (ILP)
  - Dependencies limit achievable parallelism
  - Complexity leads to the power wall

Summary

- Pipelining does not improve latency, but does improve throughput


Clock rate
Faster
Slower
Deeply pipelined
Multi-issue with deep pipelined
Multicycle
Pipelined
Multi-issue pipelined
Single-cycle
Slower
≅ 1
Faster
Instructions per cycle (IPC = 1/CPI)


Hardware
Specialized
Shared
Multi-issue with deep pipelined
Multi-issue pipelined
Single-cycle
Pipelined
Deeply pipelined
Multicycle
Several
Use latency in instructions