Pipelining (III)

Multiple Issue Processsor

Instruction-Level Parallelism (ILP)

- Pipelining: executing multiple instructions in parallel
- To increase ILP
  - Deeper pipeline
    - Less work per stage ⇒ shorter clock cycle
  - Multiple issue
    - Replicate pipeline stages ⇒ multiple pipelines
    - Start multiple instructions per clock cycle
    - CPI < 1, so use Instructions Per Cycle (IPC)
    - E.g., 4GHz 4-way multiple-issue
      - 16 BIPS, peak CPI = 0.25, peak IPC = 4
    - But dependencies reduce this in practice

Multiple Issue

- Static multiple issue
  - Compiler groups instructions to be issued together
  - Packages them into “issue slots”
  - Compiler detects and avoids hazards
- Dynamic multiple issue
  - CPU examines instruction stream and chooses instructions to issue each cycle
  - Compiler can help by reordering instructions
  - CPU resolves hazards using advanced techniques at runtime

IF	ID	EX	MEM	WB		
IF	ID	EX	MEM	WB		
	IF	ID	EX	MEM	WB	
	IF	ID	EX	MEM	WB	
		IF	ID	EX	MEM	WB
		IF	ID	EX	MEM	WB

Static Multiple Issue

- Compiler groups instructions into “issue packets”
  - Group of instructions that can be issued on a single cycle
  - Determined by pipeline resources required
- Think of an issue packet as a very long instruction
  - Specifies multiple concurrent operations
  - ⇒ Very Long Instruction Word (VLIW)

Scheduling Static Multiple Issue

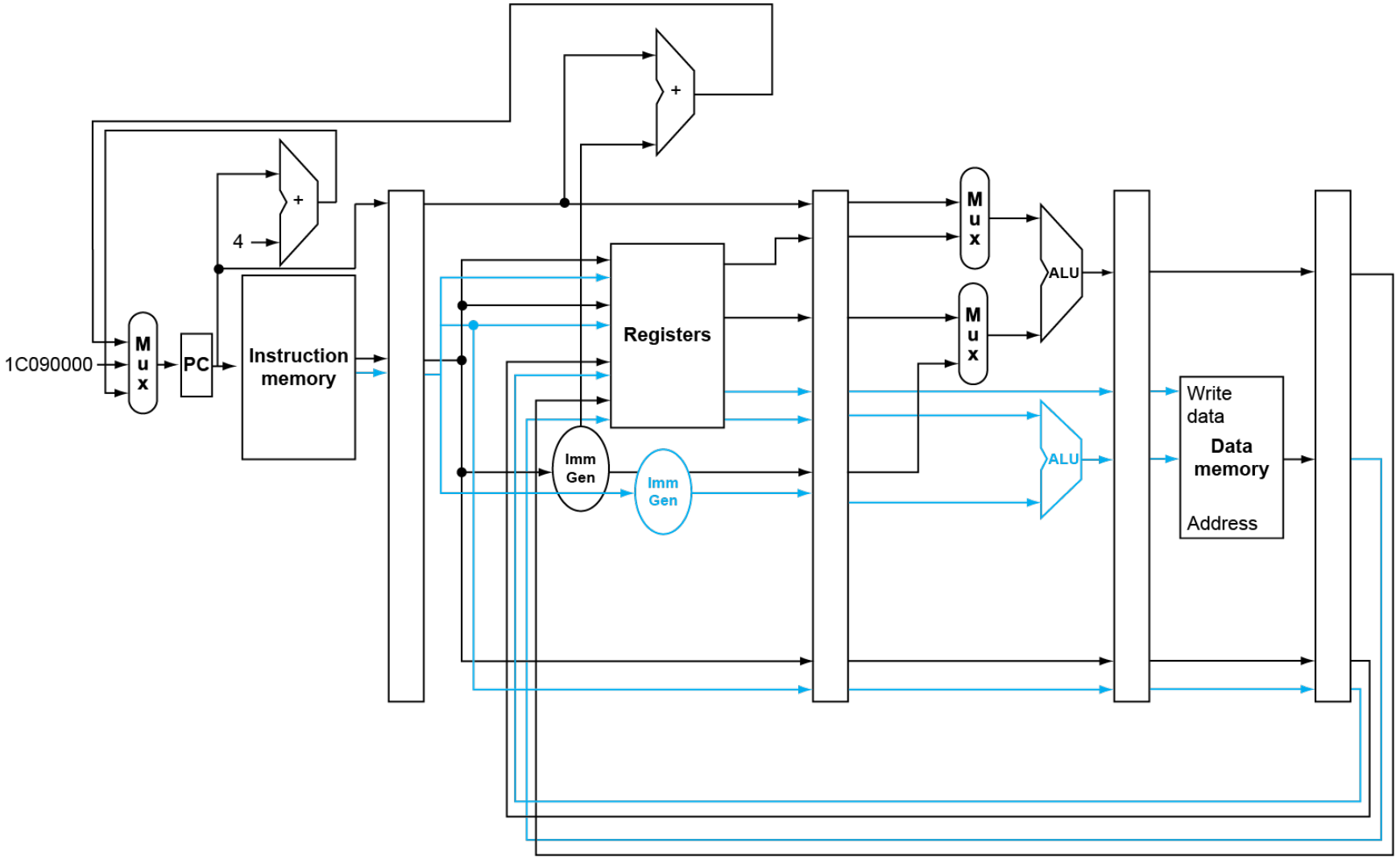
- Compiler must remove some/all hazards
  - Reorder instructions into issue packets
  - No dependencies within a packet
  - Possibly some dependencies between packets
    - Varies between ISAs; compiler must know!
  - Pad with nop if necessary

RISC-V with Static Dual Issue

- Two-issue packets
  - One ALU/branch instruction
  - One load/store instruction
  - 64-bit aligned
    - ALU/branch, then load/store
    - Pad an unused instruction with nop

Address	Instruction type	Pipeline Stages						
n	ALU/branch	IF	ID	EX	MEM	WB		
n + 4	Load/store	IF	ID	EX	MEM	WB		
n + 8	ALU/branch		IF	ID	EX	MEM	WB	
n + 12	Load/store		IF	ID	EX	MEM	WB	
n + 16	ALU/branch			IF	ID	EX	MEM	WB
n + 20	Load/store			IF	ID	EX	MEM	WB

RISC-V with Static Dual Issue



Hazards in the Dual-Issue RISC-V

- 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  x10, x0, x1
      ld   x2, 0(x10)
      ```
    - 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 RISC-V

```
Loop: ld   x31,0(x20)     // x31=array element
      add  x31,x31,x21    // add scalar in x21
      sd   x31,0(x20)     // store result
      addi x20,x20,-8     // decrement pointer
      blt  x22,x20,Loop   // branch if x22 < x20
```

	ALU/branch	Load/store	cycle
Loop:			1
			2
			3
			4

  - IPC =

Scheduling Example

- Schedule this for dual-issue RISC-V

```
Loop: ld   x31,0(x20)     // x31=array element
      add  x31,x31,x21    // add scalar in x21
      sd   x31,0(x20)     // store result
      addi x20,x20,-8     // decrement pointer
      blt  x22,x20,Loop   // branch if x22 < x20
```

	ALU/branch	Load/store	cycle
Loop:	nop	ld x31, 0(x20)	1
	addi x20, x20,-8	nop	2
	add x31, x31, x21	nop	3
	blt x22, x20, Loop	sd x31, 0(x20)	4

Scheduling Example

- Schedule this for dual-issue RISC-V

```
Loop: ld   x31,0(x20)     // x31=array element
      add  x31,x31,x21    // add scalar in x21
      sd   x31,0(x20)     // store result
      addi x20,x20,-8     // decrement pointer
      blt  x22,x20,Loop   // branch if x22 < x20
```

	ALU/branch	Load/store	cycle
Loop:	nop	ld x31, 0(x20)	1
	addi x20, x20,-8	nop	2
	add x31, x31, x21	nop	3
	blt x22, x20, Loop	sd x31, 8(x20)	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: ld   x31,0(x20)      // x31=array element
      add  x31,x31,x21     // add scalar in x21
      sd   x31,0(x20)      // store result
      addi x20,x20,-8      // decrement pointer
      blt  x22,x20,Loop    // branch if x22 < x20
```

Unrolled Loop That Minimizes Stalls

Renaming

```
Loop:   ld   x31, 0(x20)          Loop:   ld   x31, 0(x20)
        add  x31, x31, x21                add  x31, x31, x21
        sd   x31, 0(x20)                  sd   x31, 0(x20)

        ld   x31, -8(x20)                 ld   x30, -8(x20)
        add  x31, x31, x21                add  x30, x30, x21
        sd   x31, -8(x20)                 sd   x30, -8(x20)

        ld   x31, -16(x20)                ld   x29, -16(x20)
        add  x31, x31, x21                add  x29, x29, x21
        sd   x31, -16(x20)                sd   x29, -16(x20)

        ld   x31, -24(x20)                ld   x28, -24(x20)
        add  x31, x31, x21                add  x28, x28, x21
        sd   x31, -24(x20)                sd   x28, -24(x20)

        addi x20, x20,-32                 addi x20, x20,-32
        blt  x22,x20,Loop                 blt  x22,x20,Loop
```

Loop Unrolling Example

	ALU/branch	Load/store	cycle
Loop:			1
			2
			3
			4
			5
			6
			7
			8
			9
			10

```
ld   x31, 0(x20)
add  x31, x31, x21
sd   x31, 0(x20)

ld   x30, -8(x20)
add  x30, x30, x21
sd   x30, -8(x20)

ld   x29, -16(x20)
add  x29, x29, x21
sd   x29, -16(x20)

ld   x28, -24(x20)
add  x28, x28, x21
sd   x28, -24(x20)

addi x20, x20,-32
blt  x22, x20, Loop
```

Loop Unrolling Example

	ALU/branch	Load/store	cycle
Loop:	addi x20, x20,-32	ld x31, 0(x20)	1
		ld x30, -8(x20)	2
	add x31, x31, x21	ld x29, -16(x20)	3
	add x30, x30, x21	sd x31, 0(x20)	4
	add x29, x29, x21	sd x30, -8(x20)	5
		sd x29, -16(x20)	6
		ld x28, -24(x20)	7
			8
	add x28, x28, x21		9
		sd x28, -24(x20)	10

```
ld   x31, 0(x20)
add  x31, x31, x21
sd   x31, 0(x20)

ld   x30, -8(x20)
add  x30, x30, x21
sd   x30, -8(x20)

ld   x29, -16(x20)
add  x29, x29, x21
sd   x29, -16(x20)

ld   x28, -24(x20)
add  x28, x28, x21
sd   x28, -24(x20)

addi x20, x20,-32
blt  x22, x20, Loop
```

- Original version

Loop Unrolling Example

	ALU/branch	Load/store	cycle
Loop:	addi x20, x20,-32	ld x31, 0(x20)	1
		ld x30, 24(x20)	2
	add x31, x31, x21	ld x29, 16(x20)	3
	add x30, x30, x21	sd x31, 32(x20)	4
	add x29, x29, x21	sd x30, 24(x20)	5
		sd x29, 16(x20)	6
		ld x28, 8(x20)	7
			8
	add x28, x28, x21		9
	blt x22, x20, Loop	sd x28, 8(x20)	10

```
ld   x31, 0(x20)
add  x31, x31, x21
sd   x31, 0(x20)

ld   x30, -8(x20)
add  x30, x30, x21
sd   x30, -8(x20)

ld   x29, -16(x20)
add  x29, x29, x21
sd   x29, -16(x20)

ld   x28, -24(x20)
add  x28, x28, x21
sd   x28, -24(x20)

addi x20, x20,-32
blt  x22, x20, Loop
```

- Original version

Loop Unrolling Example

	ALU/branch	Load/store	cycle
Loop:	addi x20, x20,-32	ld x31, 0(x20)	1
		ld x30, 24(x20)	2
	add x31, x31, x21	ld x29, 16(x20)	3
	add x30, x30, x21	sd x31, 32(x20)	4
	add x29, x29, x21	sd x30, 24(x20)	5
		sd x29, 16(x20)	6
		ld x28, 8(x20)	7
			8
	add x28, x28, x21		9
	blt x22, x20, Loop	sd x28, 8(x20)	10

```
ld   x31, 0(x20)
add  x31, x31, x21
sd   x31, 0(x20)

ld   x30, -8(x20)
add  x30, x30, x21
sd   x30, -8(x20)

ld   x29, -16(x20)
add  x29, x29, x21
sd   x29, -16(x20)

ld   x28, -24(x20)
add  x28, x28, x21
sd   x28, -24(x20)

addi x20, x20,-32
blt  x22, x20, Loop
```

- Another optimization opportunity!!

Loop Unrolling Example

	ALU/branch	Load/store	cycle
Loop:	addi x20, x20,-32	ld x31, 0(x20)	1
	nop	ld x30, 24(x20)	2
	add x31, x31, x21	ld x29, 16(x20)	3
	add x30, x30, x21	ld x28, 8(x20)	4
	add x29, x29, x21	sd x31, 32(x20)	5
	add x28, x28, x21	sd x30, 24(x20)	6
	nop	sd x29, 16(x20)	7
	blt x22, x20, Loop	sd x28, 8(x20)	8
			9
			10

```
ld   x31, 0(x20)
add  x31, x31, x21
sd   x31, 0(x20)

ld   x30, -8(x20)
add  x30, x30, x21
sd   x30, -8(x20)

ld   x29, -16(x20)
add  x29, x29, x21
sd   x29, -16(x20)

ld   x28, -24(x20)
add  x28, x28, x21
sd   x28, -24(x20)

addi x20, x20,-32
blt  x22, x20, Loop
```

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

Dynamic Multiple Issue

- “Superscalar” processors
- CPU decides whether to issue 0, 1, 2, … each cycle
  - Avoiding structural and data hazards
- Avoids the need for compiler scheduling
  - Though it may still help
  - Code semantics ensured by the CPU

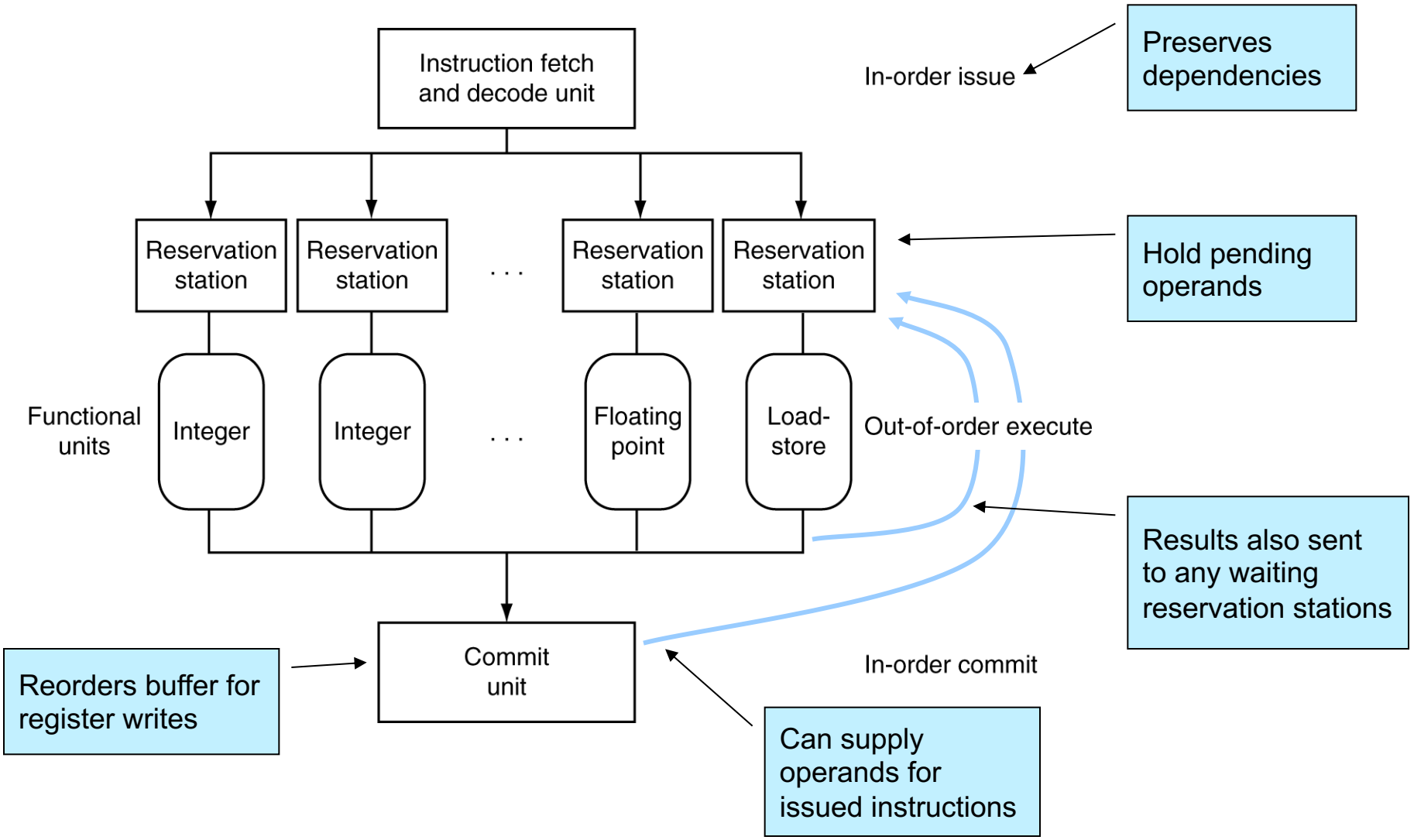
Dynamic Pipeline Scheduling

- Allow the CPU to execute instructions out of order to avoid stalls
  - But commit result to registers in order
- Example

```
ld    x31, 20(x21)
add   x1, x31, x2
sub   x23, x23, x3
andi  x5, x23, 20
```

  - Can start `sub` while `add` is waiting for `ld`

Dynamically Scheduled CPU



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?

The BIG Picture

- 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	2000MHz	22	3	Yes	1	75W
P4 Prescott	2004	3600MHz	31	3	Yes	1	103W
Intel Core	2006	2930MHz	14	4	Yes	2	75W
UltraSparc III	2003	1950MHz	14	4	No	1	90W
UltraSparc T1	2005	1200MHz	6	1	No	8	70W

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