Instruction-level parallelism

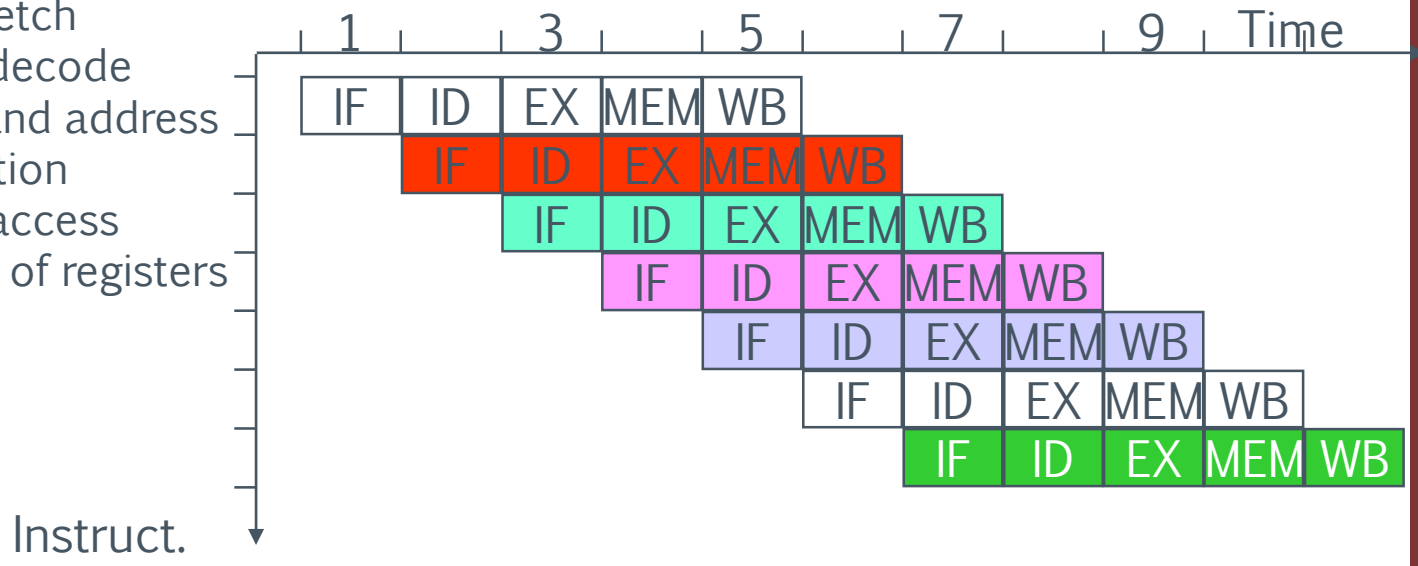
Paolo Burgio
paolo.burgio@unimore.it

Instruction Level Parallelism

- Pipelining overlaps various stages of instruction execution to achieve performance
- At a high level of abstraction, an instruction can be executed while the next one is being decoded and the next one is being fetched.
- This is akin to an assembly line for manufacture of cars.

A simple pipeline

IF: Instruction fetch
ID: Instruction decode
EX: Execution and address calculation
MEM: Memory access
WB: Write back of registers



Instruction Level Parallelism

- When exploiting pipeline ILP, goal is to minimize CPI (Cycles per Instruction)
  - Pipeline CPI =
    - Ideal pipeline CPI +
    - Structural stalls +
    - Data hazard stalls +
    - Control stalls

- Independent instructions can be executed in parallel and also in pipeline

- Unfortunately parallelism with *basic block* is limited
  - Typical size of basic block = 5-6 instructions
  - Must optimize across branches: speculative parallelization

Instruction Level Parallelism

- Pipelining, however, has several limitations.

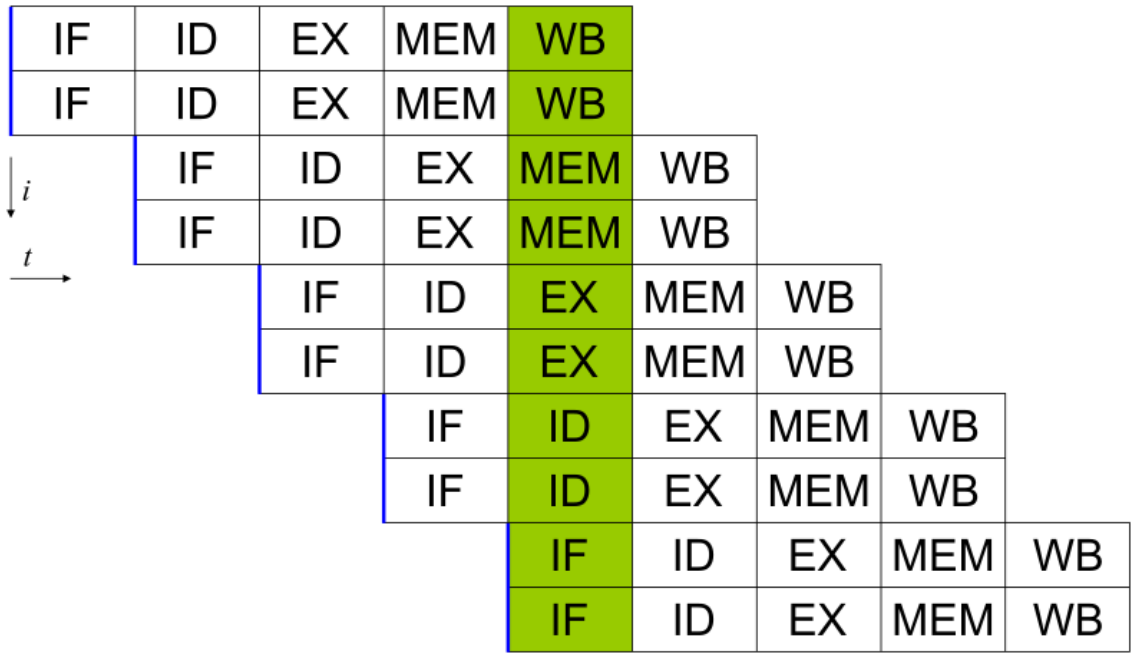
- The speed of a pipeline is eventually limited by the slowest stage.
- For this reason, conventional processors rely on very deep pipelines (20 stage pipelines in some Pentium processors).

- However, in typical program traces, every 5-6 instructions, there is a conditional jump! This requires very accurate branch prediction.
  - The penalty of a misprediction grows with the depth of the pipeline, since a larger number of instructions will have to be flushed.

- Data and resource dependency may stall the pipeline

ILP: Pipelining and Superscalar Execution

- We can use multiple pipelines
- Superscalar processors simultaneously dispatch multiple instructions to redundant functional units on the processor
  - Instructions are still issued from a sequential instruction stream
  - CPU hardware dynamically checks for data dependencies between instructions at run time (versus software checking at compile time)
  - The CPU issues multiple instructions per clock cycle
- The question then becomes how to select from the stream the instructions to be issued on these pipelines
- Compilers can help in generation code that can be easily parallelized



Superscalar Execution: An Example of two-way execution

```
load R1, @1000    # R1 = *(1000)
add R1, @1004     # R1 = R1 + *(1004)
add R1, @1008     # R1 = R1 *(1008)
add R1, @100C     # R1 = R1 + *(100C)
store R1, @2000   # *(2000) = R1
```

This code does not favor ILP exploitation (**no superscalar**)
Every instruction (data) depends on the previous one

In the next slide, we add another register and perform a partial sum between registers

- Sum a list of 4 integer numbers
- Stored in consecutive memory locations

  16 B (size)

  [0x1000 – 0x100F]
- Result stored to:

  [0x2000 – 0x2003]

› Instructions with only two operands
  – The 1st operand is both source and destination

› Mixed addressing
  – Both register and memory operands

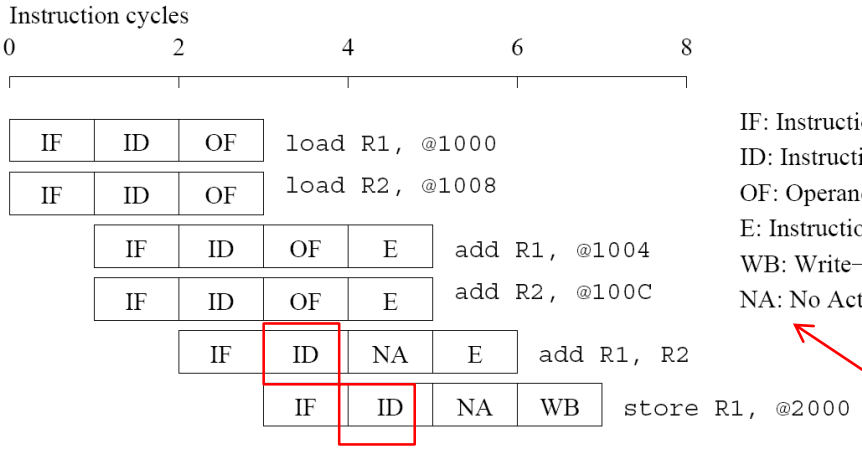
Superscalar Execution: An Example of two-way execution

```
1. load R1, @1000      1. load R1, @1000      1. load R1, @1000
2. load R2, @1008      2. add R1, @1004       2. add R1, @1004
3. add R1, @1004       3. add R1, @1008       3. load R2, @1008
4. add R2, @100C       4. add R1, @100C       4. add R2, @100C
5. add R1, R2          5. store R1, @2000     5. add R1, R2
6. store R1, @2000                            6. store R1, @2000
```

(i) (ii) (iii)

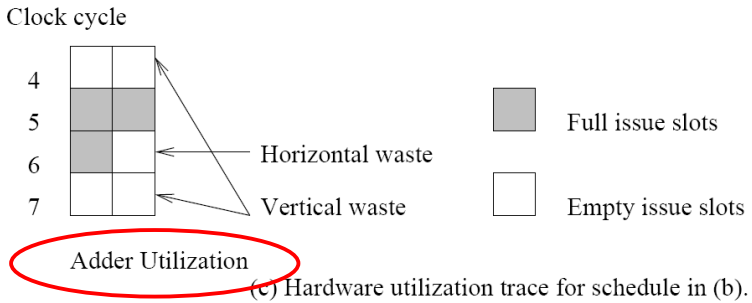
(a) Three different code fragments for adding a list of four numbers.

Pipeline of 4 stages



IF: Instruction Fetch
ID: Instruction Decode
OF: Operand Fetch
E: Instruction Execute
WB: Write–back
NA: No Action

(b) Execution schedule for code fragment (i) above.



(c) Hardware utilization trace for schedule in (b).

Pipeline of 4 stages

NA means that the pipeline stalls, waiting for the availability of data (**data dependency** due to registers **R1** and **R2**)

Superscalar Execution: An Example

- In the above example, there is some wastage of resources due to data dependencies.
- The example also illustrates that different instruction mixes with identical semantics can take significantly different execution time.

Superscalar Execution

- Scheduling of instructions is determined by a number of factors:
  - Data Dependency: The result of one operation is an input to the next.
  - Resource Dependency: Two operations require the same resource.
  - Branch Dependency: Scheduling instructions across conditional branch statements cannot be done deterministically a-priori.
  - The Dynamic scheduler, a piece of hardware, looks at a large number of instructions in an instruction queue and selects appropriate number of instructions to execute concurrently (if any!) based on these factors.
  - The complexity of this hardware is an important constraint on superscalar processors.

Issue Mechanisms

- In the simpler model, instructions can be issued only in the order in which they are encountered.
  - That is, if the second instruction cannot be issued because it has a data dependency with the first, only one instruction is issued in the cycle.
  - This is called *in-order issue*.
  - The compiler may statically reschedule instructions to increase the chance of parallel execution
- In a more aggressive model, instructions can be issued *out-of-order*.
  - In this case, if the second instruction has data dependencies with the first, but the third instruction does not, the first and third instructions can be co-scheduled. This is also called dynamic issue/scheduling.
- Performance of in-order issue is generally limited

Efficiency Considerations

- Due to limited parallelism in typical instruction traces, dependencies, or the inability of the scheduler to extract parallelism, the performance of superscalar processors is eventually limited
  - Not all functional units can be kept busy at all times
  - A lot of waste of resources, i.e., functional units not utilized

- Conventional microprocessors typically support up to four-way superscalar execution.

Very Long Instruction Word (VLIW) Processors

- The hardware cost and complexity of the superscalar scheduler is a major consideration in processor design.
- To address this issues, VLIW processors rely on compile time analysis to identify and bundle together instructions that can be executed concurrently
- These instructions are packed and dispatched together, and thus the name very long instruction word
  - An VLIW instruction hosts many op codes and associated operands
- This concept was used with some commercial success in the Multiflow Trace machine (1984).
- Variants of this concept are employed in the Intel Itanium IA64 processors.

VLIW Processors: Considerations

- Issue hardware is simpler.
- Compiler has a bigger context (than Superscalar CPUs) from which to select instructions to co-schedule
  - to pack in a VLIW instruction
- Compilers, however, do not have runtime information such as cache misses or branch taken.
- Scheduling decision must be based on sophisticated static predictions to pack independent instructions in each VLIW
  - may result inherently conservative.
- VLIW performance is highly dependent on the compiler. A number of techniques such as loop unrolling, speculative execution, branch prediction are critical.
- Typical VLIW processors are limited to 4-way to 8-way parallelism.
- Note: statically scheduled superscalars are actually closer in concept to VLIWs

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

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)

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
  - $\Rightarrow$ Very Long Instruction Word (VLIW)

Scheduling Static Multiple Issue

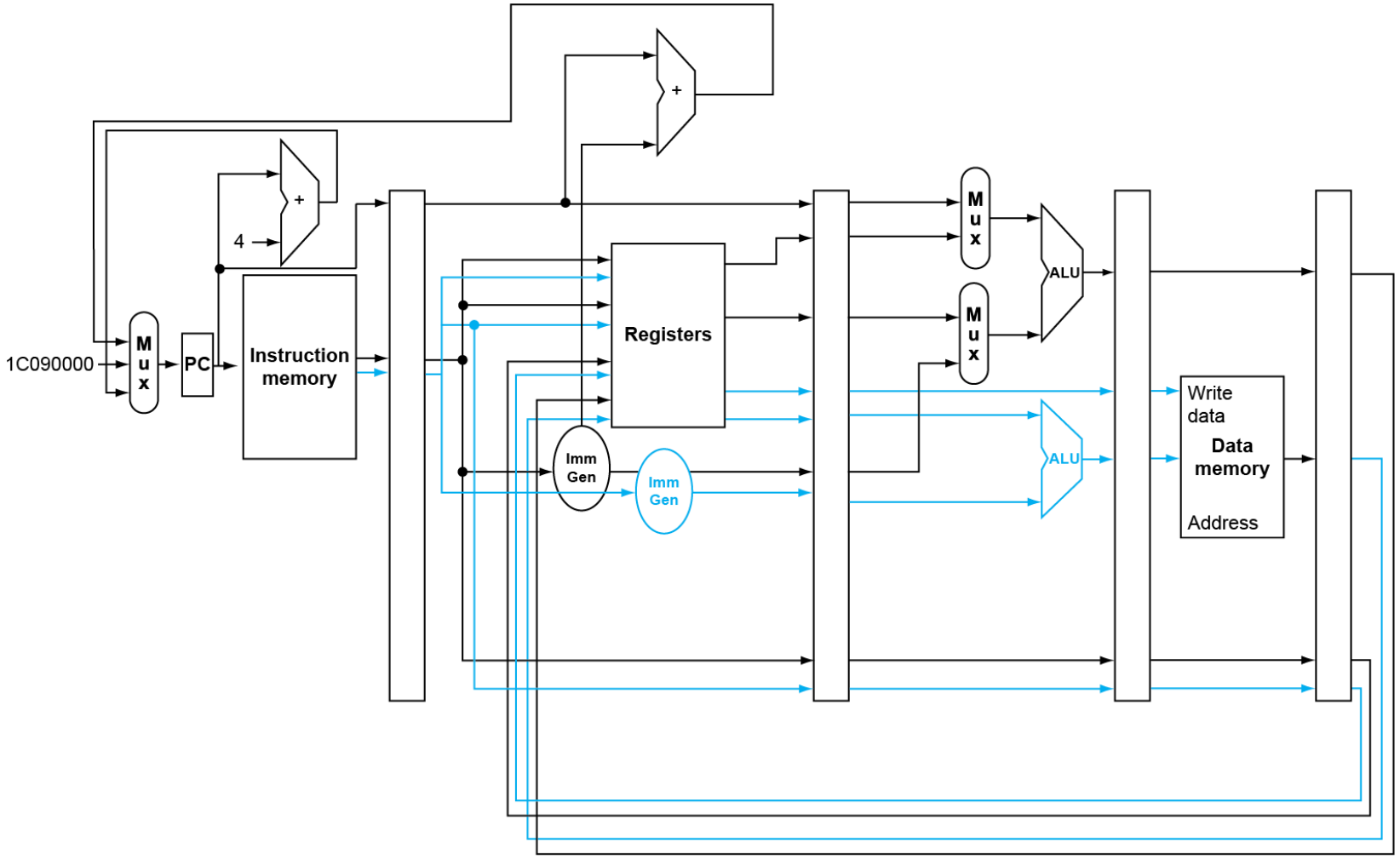
- Compiler must remove some/all hazards
  - Reorder instructions into issue packets
  - No dependencies with a packet
  - Possibly some dependencies between packets
    - Varies between ISAs; compiler must know!
  - Pad with nop if necessary

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:	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 Unrolling Example

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

- 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

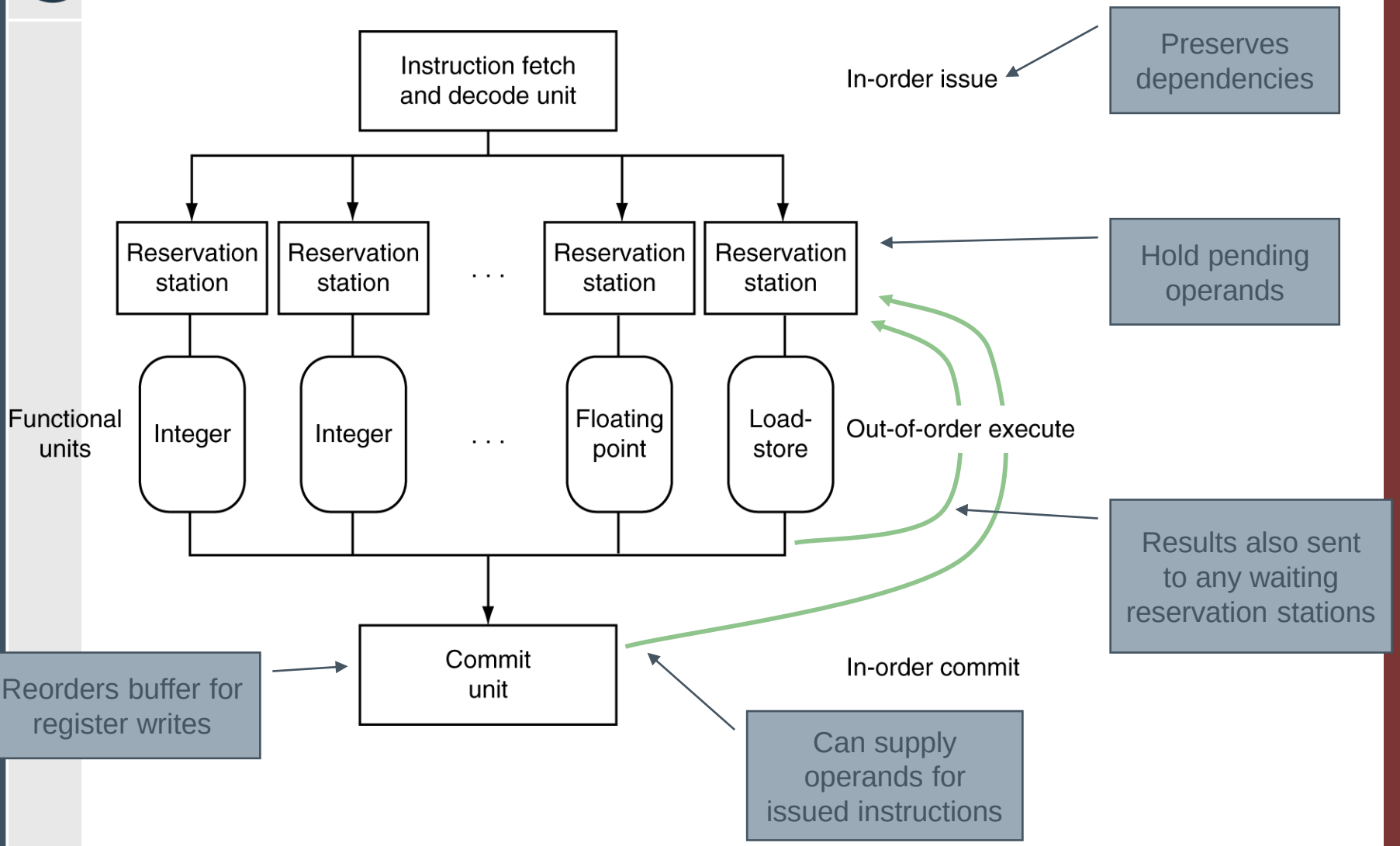
Dynamic Pipeline Scheduling

- 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



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

- 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 predictable
  - 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

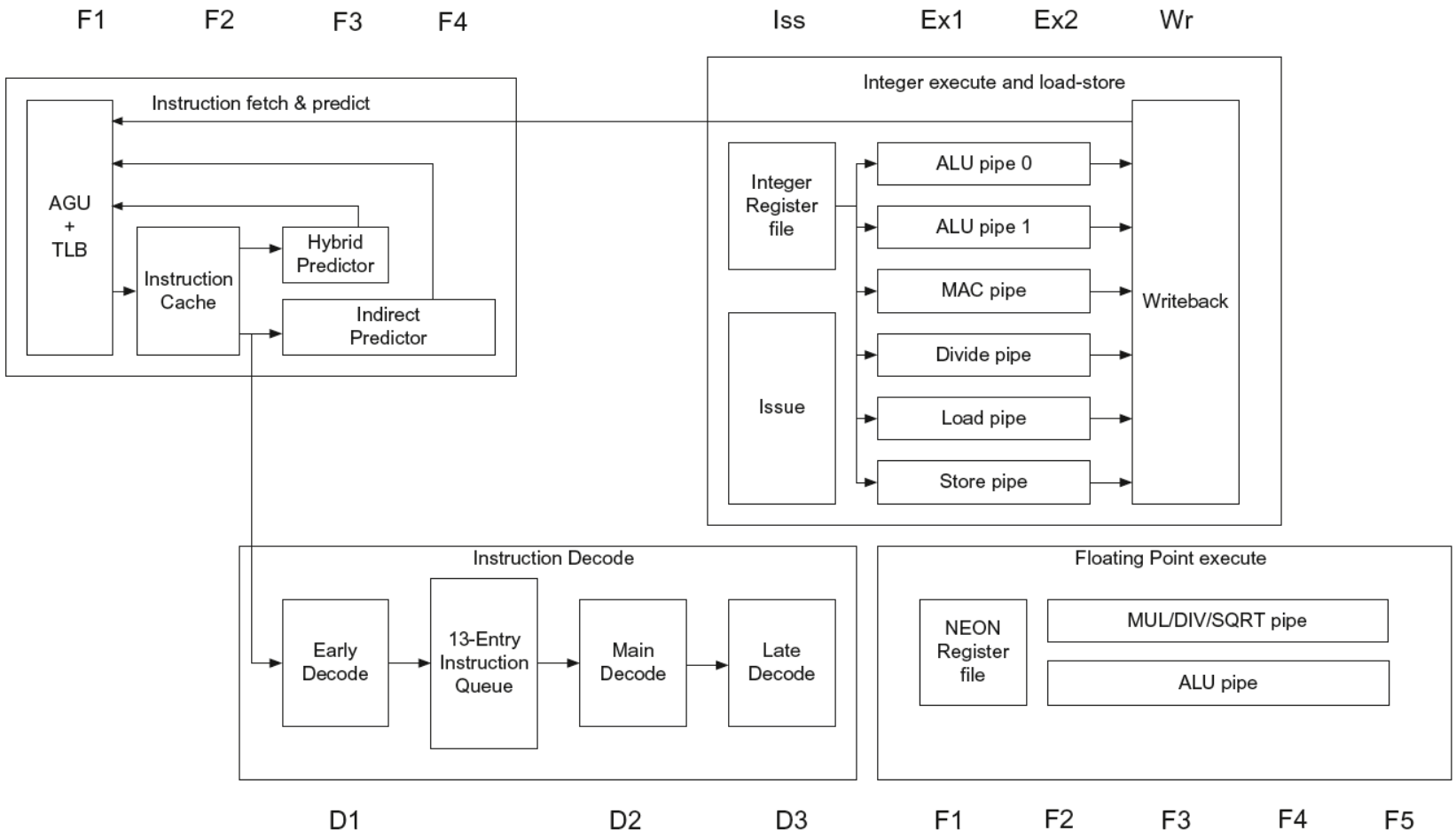
- Complexity of dynamic scheduling and speculations requires power
- Multiple simpler cores may be better

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	2000MHz	22	3	Yes	1	75W
P4 Prescott	2004	3600MHz	31	3	Yes	1	103W
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

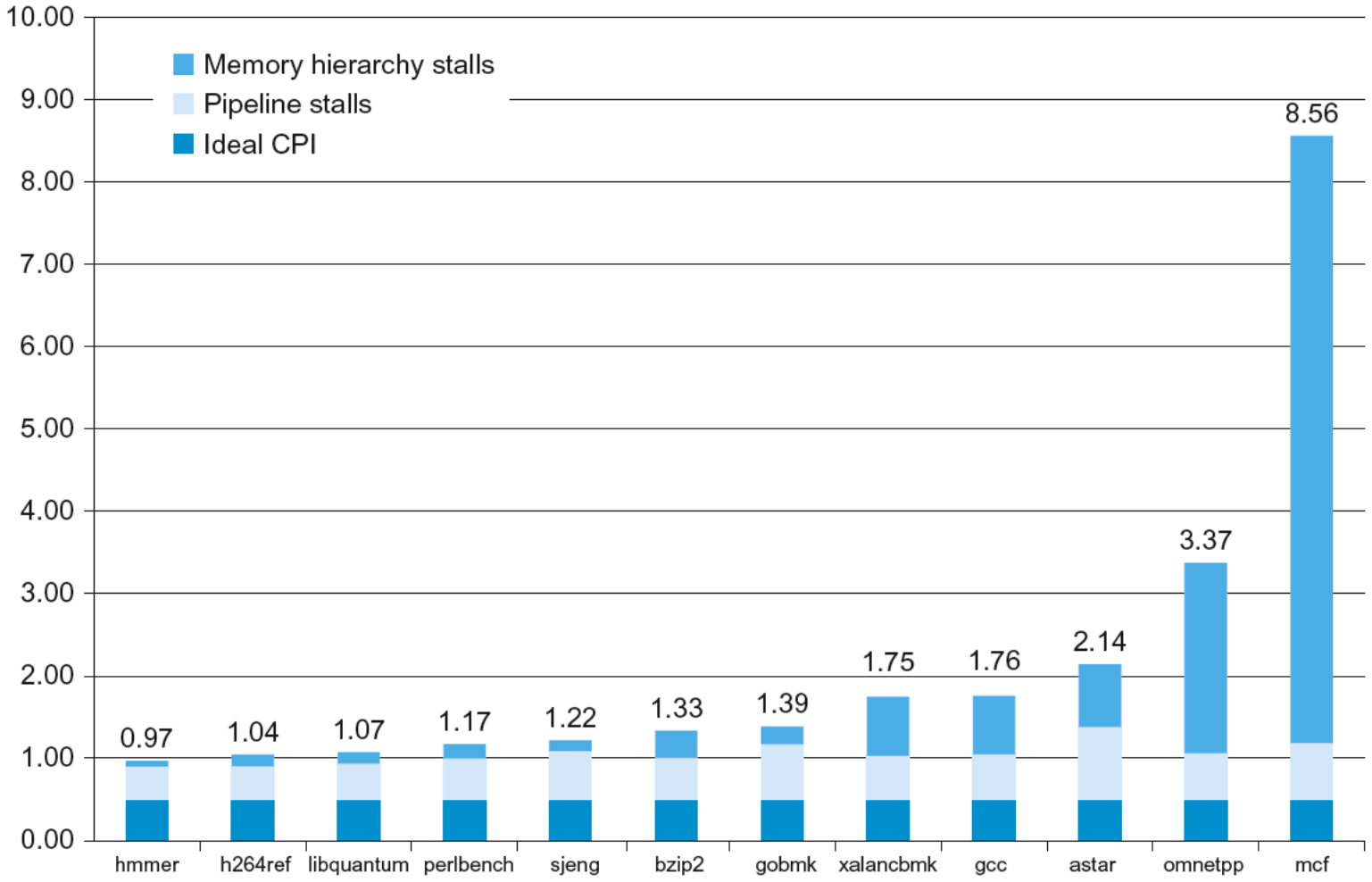
Cortex A53 and Intel i7

Processor	ARM A53	Intel Core i7 920
Market	Personal Mobile Device	Server, cloud
Thermal design power	100 milliWatts (1 core @ 1 GHz)	130 Watts
Clock rate	1.5 GHz	2.66 GHz
Cores/Chip	4 (configurable)	4
Floating point?	Yes	Yes
Multiple issue?	Dynamic	Dynamic
Peak instructions/clock cycle	2	4
Pipeline stages	8	14
Pipeline schedule	Static in-order	Dynamic out-of-order with speculation
Branch prediction	Hybrid	2-level
1st level caches/core	16-64 KiB I, 16-64 KiB D	32 KiB I, 32 KiB D
2nd level caches/core	128-2048 KiB	256 KiB (per core)
3rd level caches (shared)	(platform dependent)	2-8 MB

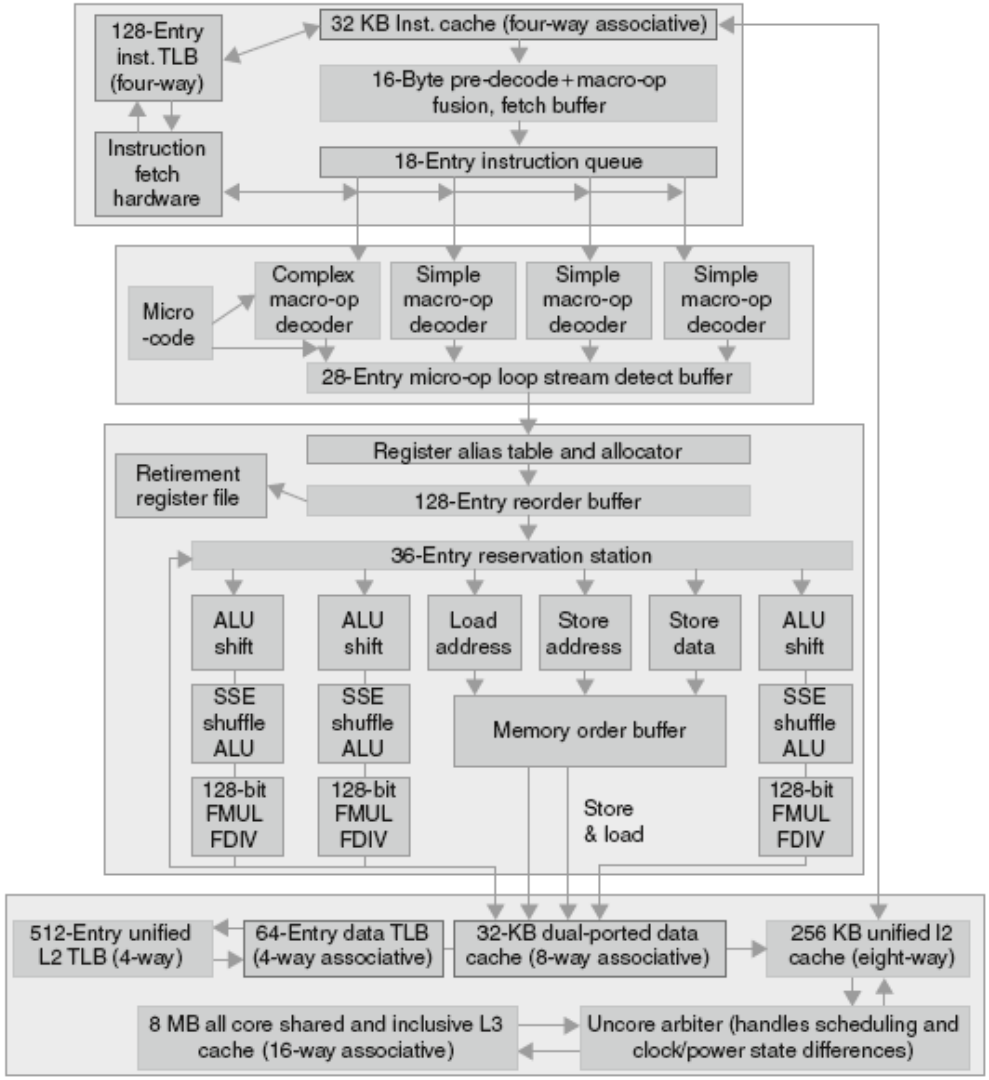
ARM Cortex-A53 Pipeline



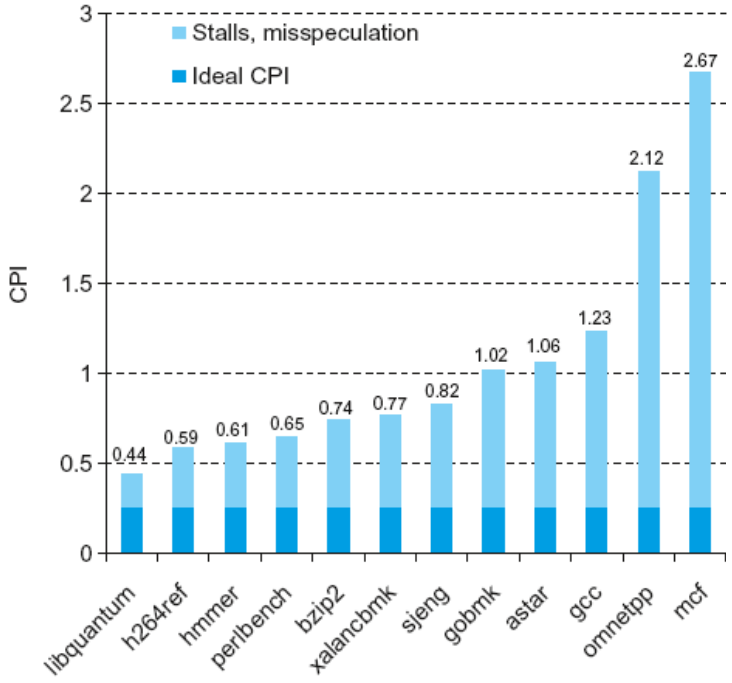
ARM Cortex-A53 Performance

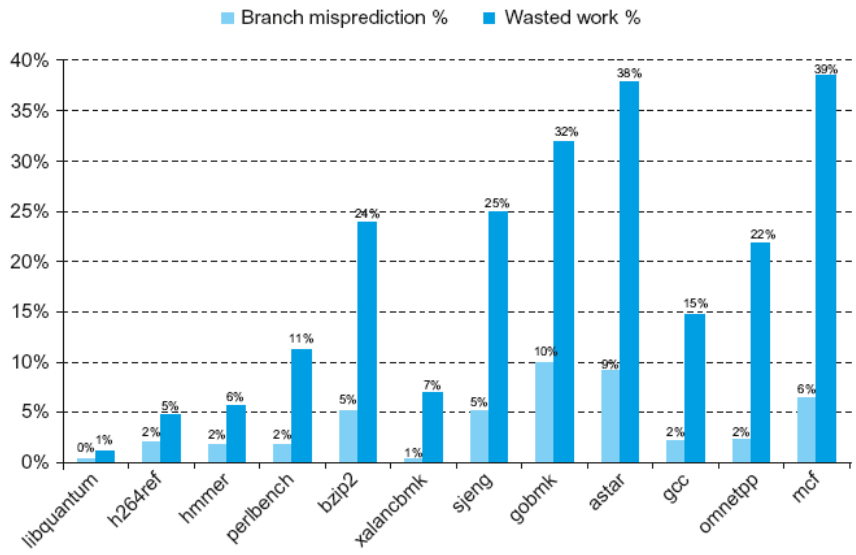


Core i7 Pipeline



Core i7 Performance





Matrix Multiply

- Unrolled C code

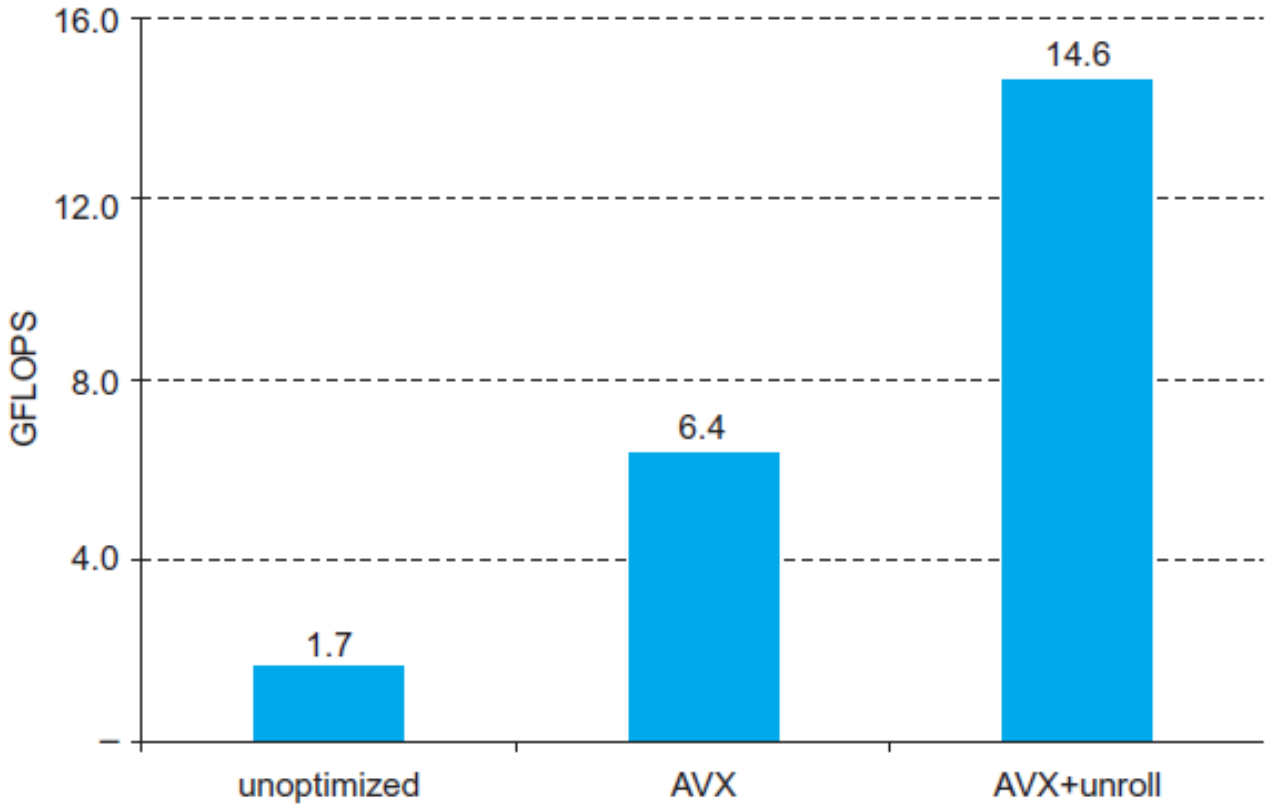
```
1 #include <x86intrin.h>
2 #define UNROLL (4)
3
4 void dgemm (int n, double* A, double* B, double* C)
5 {
6  for ( int i = 0; i < n; i+=UNROLL*4 )
7   for ( int j = 0; j < n; j++ ) {
8    __m256d c[4];
9    for ( int x = 0; x < UNROLL; x++ )
10    c[x] = _mm256_load_pd(C+i+x*4+j*n);
11
12   for( int k = 0; k < n; k++ )
13   {
14    __m256d b = _mm256_broadcast_sd(B+k+j*n);
15    for (int x = 0; x < UNROLL; x++)
16    c[x] = _mm256_add_pd(c[x],
17                         _mm256_mul_pd(_mm256_load_pd(A+n*k+x*4+i), b));
18   }
19
20    for ( int x = 0; x < UNROLL; x++ )
21     _mm256_store_pd(C+i+x*4+j*n, c[x]);
22  }
23 }
```

Matrix Multiply

- Assembly code:

```
1 vmovapd (%r11),%ymm4                  # Load 4 elements of C into %ymm4
2 mov %rbx,%rax                         # register %rax = %rbx
3 xor %ecx,%ecx                         # register %ecx = 0
4 vmovapd 0x20(%r11),%ymm3              # Load 4 elements of C into %ymm3
5 vmovapd 0x40(%r11),%ymm2              # Load 4 elements of C into %ymm2
6 vmovapd 0x60(%r11),%ymm1              # Load 4 elements of C into %ymm1
7 vbroadcastsd (%rcx,%r9,1),%ymm0       # Make 4 copies of B element
8 add $0x8,%rcx # register %rcx = %rcx + 8
9 vmulpd (%rax),%ymm0,%ymm5             # Parallel mul %ymm1,4 A elements
10 vaddpd %ymm5,%ymm4,%ymm4             # Parallel add %ymm5, %ymm4
11 vmulpd 0x20(%rax),%ymm0,%ymm5        # Parallel mul %ymm1,4 A elements
12 vaddpd %ymm5,%ymm3,%ymm3             # Parallel add %ymm5, %ymm3
13 vmulpd 0x40(%rax),%ymm0,%ymm5        # Parallel mul %ymm1,4 A elements
14 vmulpd 0x60(%rax),%ymm0,%ymm0        # Parallel mul %ymm1,4 A elements
15 add %r8,%rax                         # register %rax = %rax + %r8
16 cmp %r10,%rcx                        # compare %r8 to %rax
17 vaddpd %ymm5,%ymm2,%ymm2             # Parallel add %ymm5, %ymm2
18 vaddpd %ymm0,%ymm1,%ymm1             # Parallel add %ymm0, %ymm1
19 jne 68 <dgemm+0x68>                  # jump if not %r8 != %rax
20 add $0x1,%esi                        # register % esi = % esi + 1
21 vmovapd %ymm4,(%r11)                 # Store %ymm4 into 4 C elements
22 vmovapd %ymm3,0x20(%r11)             # Store %ymm3 into 4 C elements
23 vmovapd %ymm2,0x40(%r11)             # Store %ymm2 into 4 C elements
24 vmovapd %ymm1,0x60(%r11)             # Store %ymm1 into 4 C elements
```

Performance Impact



Fallacies

- Pipelining is easy (!)
  - The basic idea is easy
  - The devil is in the details
    - e.g., detecting data hazards
- Pipelining is independent of technology
  - So why haven't we always done pipelining?
  - More transistors make more advanced techniques feasible
  - Pipeline-related ISA design needs to take account of technology trends
    - e.g., predicated instructions

Pitfalls

- Poor ISA design can make pipelining harder
  - e.g., complex instruction sets (VAX, IA-32)
    - Significant overhead to make pipelining work
    - IA-32 micro-op approach
  - e.g., complex addressing modes
    - Register update side effects, memory indirection
  - e.g., delayed branches
    - Advanced pipelines have long delay slots

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