Instruction-Level Parallelism (ILP)

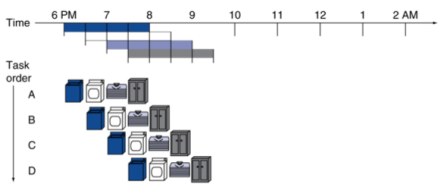
- Instruction-level parallelism: parallelism among instructions
  - Pipelining is one type of ILP: because pipeline executes multiple instructions in parallel
- To increase ILP
  - Deeper pipeline (more number of stages)
    - Less work per stage ⇒ shorter clock cycle
  - Multiple issue (start multiple instructions in one clock)
    - 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 (billion instructions per second), peak CPI = 0.25, peak IPC = 4
      - But dependencies reduce this in practice

更多的stage数

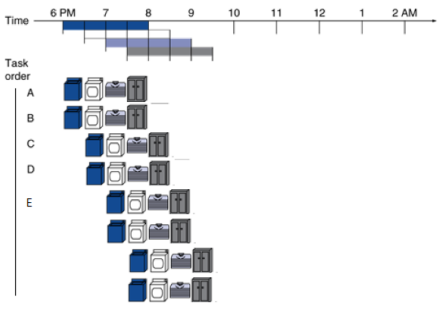
每个clock有更多指令

Pipeline vs. Multiple-issue 并行 vs. 多发射

- Pipeline:



- Multiple-issue:



Two Key Problems of Multiple Issue

- Packaging instructions into issue slots
  - How many instructions can be issued
  - Which instructions should be issued
- Dealing with data and control hazards

Static / Dynamic Multiple Issue

- Static multiple issue – decision made by compiler
  - Compiler groups instructions to be issued together
  - Packages them into "issue slots"
  - Compiler detects and avoids hazards
- Dynamic multiple issue – decision made by processor
  - 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

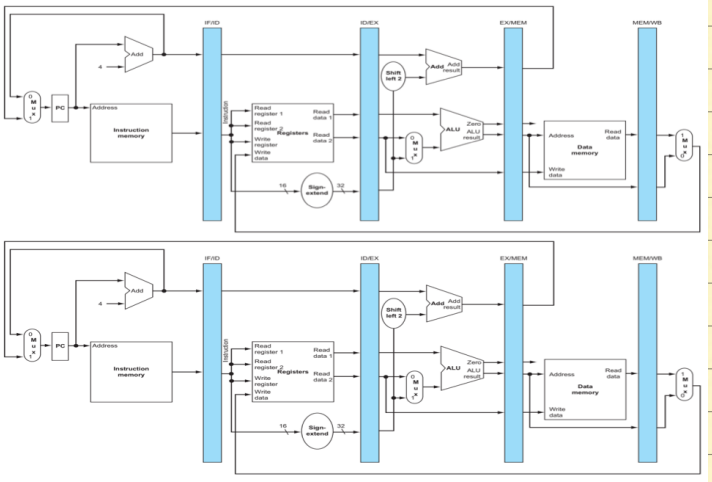
静态-多发射：由编译器控制

动态-多发射：由处理器控制

Static Multiple Issue

- Compiler groups instructions into "issue packet"
  - 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)

Naïve Static Dual Issue



相当于多了一个 pipeline

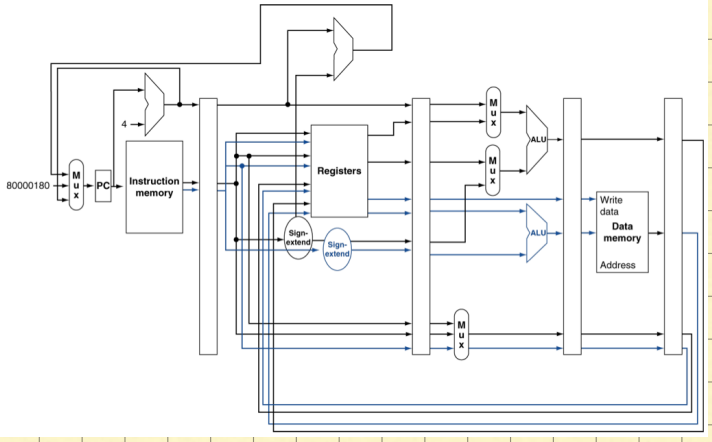
MIPS with Static Dual Issue

- Two-issue packets
  - Divide instructions into two types:
    - Type 1: ALU or branch instructions
    - Type 2: load or store instructions
  - In each cycle, execute a type1 and a type2 ins simultaneously
  - 64-bit aligned instructions

将指令分成两类.

每个 time slot 同时放两个指令.
这两个指令分别来自两类之中.

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



增加了一些硬件.

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`
      `lw $s2, 0($t0)`
    - Split into two packets, effectively a stall
- Load-use hazard
- Still one cycle use latency (number of clock cycles between load and use), but now two instructions
- More aggressive scheduling required

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

Scheduling Example

- Schedule this for dual-issue MIPS
  - IPC = 5/4 = 1.25 (c.f. peak IPC = 2)

```
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:		lw $t0, 0($s1)	1
			2
			3
			4

①

	ALU/branch	Load/store	cycle
Loop:		lw $t0, 0($s1)	1
			2
	addu $t0, $t0, $s2		3
			4

②

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

③

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

④

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

⑤

	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

⑥

Loop Unrolling 将loop展开，每次循环执行多条。

- "Name dependence" or "anti-dependence"
  - Repeated instance of
    ```
    lw   $t0, 0($s1)
    addu $t0, $t0, $s2
    sw   $t0, 0($s1)
    ```
  - The data are independent, no data flow between two sets
  - Dependence comes from the reuse of the register name
- We use "loop unrolling" to remove "name dependence"
  - Replicate loop body to expose more parallelism
  - Use different registers per replication (called "register renaming")
  - Reduces loop-control overhead

- IPC = 14/8 = 1.75
  - Closer to 2, but at cost of registers and code size
- How about we choose to execute 3 or 5 instructions in a loop, instead of 4?

可以用多个不同的寄存器减少依赖。

Loop Unrolling Example

- Repeat the code in the loop

```
lw   $t0, 0($s1)        lw   $t1, -4($s1)
addu $t0, $t0, $s2      addu $t1, $t1, $s2
sw   $t0, 0($s1)        sw   $t1, -4($s1)
```

	ALU/branch	Load/store	cycle
Loop:	addi $s1, $s1,-16	lw $t0, 0($s1)	1
			2
	addu $t0, $t0, $s2		3
		sw $t0, 16($s1)	4
			5
			6
			7
			8

①

	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		3
	addu $t1, $t1, $s2	sw $t0, 16($s1)	4
		sw $t1, 12($s1)	5
			6
			7
			8

②

	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, $t3, $s2	sw $t1, 12($s1)	6
	nop	sw $t2, 8($s1)	7
	bne $s1, $zero, Loop	sw $t3, 4($s1)	8

③

Dynamic Multiple Issue

- The decision is made by the processor during execution
- also called "Superscalar" processors
- CPU decides whether to issue 0, 1, 2, … each cycle
  - Avoiding structural and data hazards
- No need for compiler scheduling
  - Though it may still help
  - Code semantics ensured by the CPU

Dynamic Pipeline Scheduling

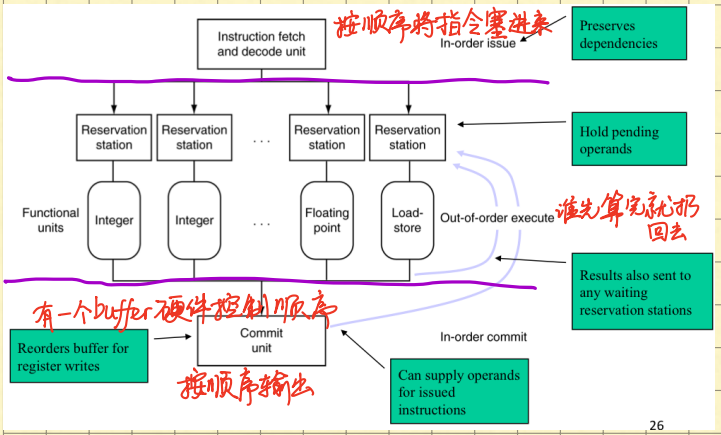
- Hardware support for reordering the order of instruction execution
- Allow the CPU to execute instructions out of order to avoid stalls
  - But commit result to registers in order
- Example

```
lw    $t0, 20($s2)
addu  $t1, $t0, $t2
sub   $s4, $s4, $t3
slti  $t5, $s4, 20
```

  - Can start sub while addu is waiting for lw

要增加硬件支持.
使得CPU能乱序地执行指令.

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

branch中 dynamic 效率比较好.

Summary

	Static multiple issue	Dynamic multiple issue
Decision made by	Compiler (software)	Processor (hardware)
Also called	Very long instruction word (VLIW)	Superscaler
Ways to remove hazard	Loop unrolling/ Register renaming	Out-of-order execution

sigle-cycle CPU + multiple issue pipeline CPU

Speculation 猜测

- "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
- Examples
  - Speculate on branch outcome
    - Roll back if path taken is different
  - Speculate on load
    - Roll back if location is updated

在branch和load时使用speculation

Compiler / Hardware Speculation

- Compiler can reorder instructions
  - e.g., change the sequence of load and other ins.
  - e.g., change the sequence of branch and other ins.
  - 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)

beq ....., B
A ..... exception ← A这里的exception一般不处理
B

静态预测用ISA

动态预测用buffer

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
  - Some parallelism is hard to expose
  - Memory delays and limited bandwidth
- Speculation can help if done well

不是issue越多越好.

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
Core i5 Nehalem	2010	3300MHz	14	4	Yes	1	87W
Core i5 Ivy Bridge	2012	3400MHz	14	4	Yes	8	77W

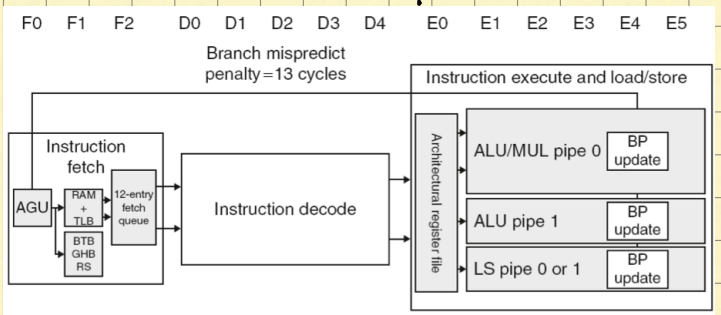
stage越多越耗电
clock rate越高越耗电

当前处理器大多是14个stage
issue越多越复杂

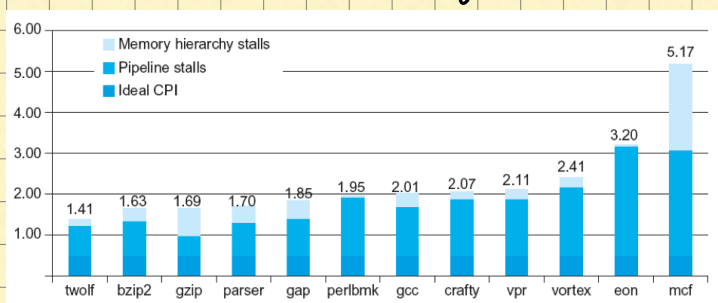
Cortex A8 & Intel i7

Processor	ARM A8	Intel Core i7 920
Market	Personal Mobile Device	Server, cloud
Thermal design power	2 Watts	130 Watts
Clock rate	1 GHz	2.66 GHz
Cores/Chip	1	4
Floating point?	No	Yes
Multiple issue?	Dynamic	Dynamic
Peak instructions/clock cycle	2	4
Pipeline stages	14	14
Pipeline schedule	Static in-order	Dynamic out-of-order with speculation
Branch prediction	2-level	2-level
1st level caches/core	32 KiB I, 32 KiB D	32 KiB I, 32 KiB D
2nd level caches/core	128-1024 KiB	256 KiB
3rd level caches (shared)	-	2- 8 MB

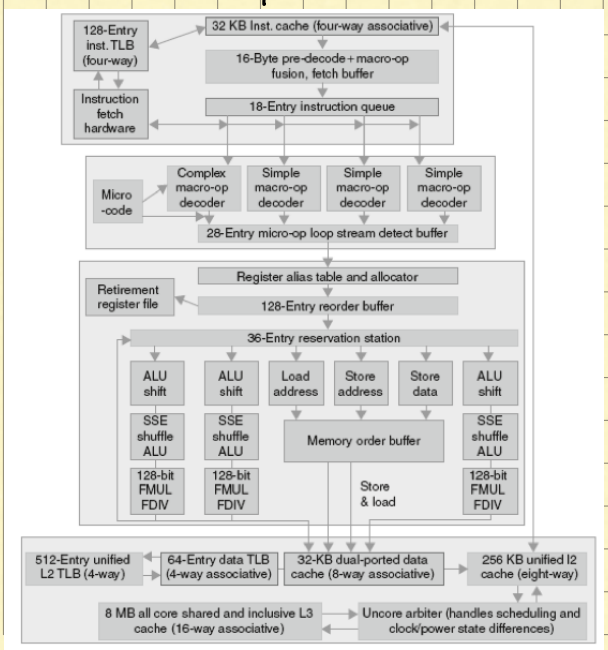
ARM Cortex-A8 Pipeline



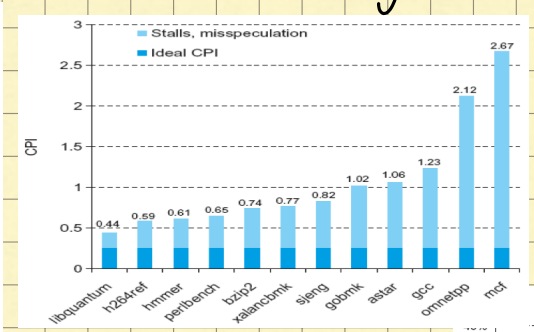
ARM Cortex-A8 Performance

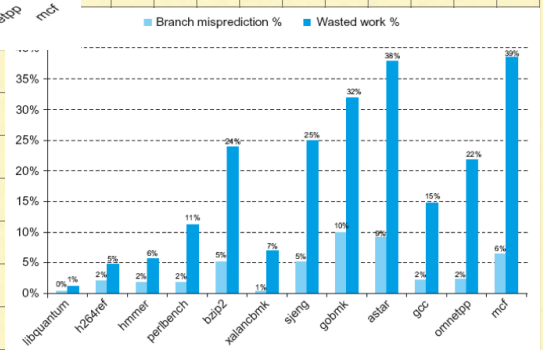


Core i7 Pipeline



Core i7 Performance





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 do pipelining?
  - More transistors make more advanced techniques feasible
  - Pipeline-related ISA design needs to take account of technology trends
    - e.g., predicated instructions

并行并不简单

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

A. Software B. Hardware C. Both

Branch prediction: C
Multiple Issue : C
Very long instruction word : A
Superscaler: B
Dynamic scheduling : B
Out-of-order execution: B
Speculation: C