Computer Architecture

计算机体系结构

第3讲：ISA & ILP（2）

张献伟

xianweiz.github.io

DCS3013, 9/21/2022

Review Questions

- Guidelines and principles in design and analysis of arch?
  Parallelism, locality, common case
- Amdahl's Law?
  Speedup is limited by the fraction.
- If 80% of a program can be parallelized to run 4x faster, what's the overall speedup? The theoretical max?
  1 / (20% + 80%/4) = 2.5x   max: 1 / 20% = 5
- CPI vs. IPC?
  Cycles per instruction   instructions per cycle
- ISA vs. u-arch?
  u-arch is the specific implementation of ISA. Arch = ISA + u-arch
- What affect CPU performance?
  Program, compiler, ISA, organization, technology

Existing ISAs

- RISC: reduced-instruction set computer[精简指令集]
  - Coined by Patterson in early 80’s
  - RISC-I (Patterson), MIPS (Hennessy), IBM 801 (Cocke)
  - Examples: PowerPC, ARM, SPARC, Alpha, PA-RISC
- CISC: complex-instruction set computer[复杂指令集]
  - Term didn’t exist before “RISC”
  - Examples: x86, VAX, Motorola 68000, etc.

CISC RISC

国产CPU架构

- x86
  - 曙光/海光
- ARM
  - 华为、飞腾
- 自主
  - 龙芯、申威

* CPU及指令集演进 (漫画 | 20多年了，为什么国产CPU还是不行？)

国产GPU

国内GPU企业大盘点

公司	地区	融资/市值	股东	创始人等	技术背景
景嘉微电子	长沙	341亿	300474	曾万辉 胡亚华 饶先宏	国防科大、自研
芯动科技	珠海	估值300亿	芯动创始人团队、珠海国资、三星		中国一站式IP和芯片定制领军企业。"风华1号"GPU，刷新国产高性能GPU纪录。
芯原微电子	上海	297亿	大基金、张江火炬、浦新投、嘉兴基金	戴伟民	收购图芯美国，获得了GPU IP。
航锦科技	辽宁/长沙	241亿	000818		并购的长沙韶光
兆芯	上海		上汽集团 上海国资委	叶峻	上海国资和中国台湾威盛，世界上第三家拥有X86授权的微处理器公司
昆仑芯	北京	估值130亿	百度、CPE、IDG、君联	欧阳剑（技术背景 百度）	昆仑芯前身是百度智能芯片及架构部。
凌久微电子	武汉		中国船舶重工集团公司第七〇九研究所控股		GP101实现了我国通用3D显卡
天数智芯	上海/南京	约20亿	各路基金	刁石京	中国第一家通用GPU高端芯片及超级算力提供商，唯一实现通用GPU量产的企业
壁仞科技	上海	47亿	中芯聚源、上海、珠海、南通、IDG、格力、松禾等等	张文，商汤总裁 CEO李新荣 AMD系	GPGPU/图形GPU
沐曦集成电路	上海	10亿	国调基金、经纬中国、和利资本、红杉中国、联想、招商、复星、东方富海	AMD等	高性能GPU领域
登临科技	上海		高通、中电、元禾璞华	高通、华为等	GPU+为核心云端AI计算平台公司
摩尔线程	北京	30亿	深创投、红杉资本中国基金、GGV纪源、腾讯、字节	Nvdia张建中"隐居幕后"，冶金部、南京理工	首颗国产全功能GPU研制成功
瀚博半导体	上海	24亿	互联网基金、经纬、红点创投、五源资本、赛富	钱军、张磊 AMD系	GPU专家做DSA架构设计云端AI推理芯片
燧原科技	上海	31.4亿	科创投、腾讯、武岳峰、真格基金	赵立东 清华85级、AMD、紫光	GPGPU AI芯片
芯瞳半导体	西安	估值20亿	烟台基金 西安基金 盈富泰克	黄虎才（AMD、英特尔、华为等任职）	党政八大行业、云游戏。
龙芯	北京		中科院、北京市政府	LoongArch龙芯架构	集成GPU 集成显卡
中微电	深圳		CEC中电系	梅思行（英伟达）	自主可控
中船重工716研究所	连云港				JARI G12是2018年性能最强的国产通用图形处理器
华为海思	深圳		华为		自研GPU、嵌入式GPU

https://t.cj.sina.com.cn/articles/view/1874424022/6fb970d600101asua

Performance Argument[性能的争论]

- Performance equation:
  - (instructions/program) * (cycles/instruction) * (seconds/cycle)
- CISC
  - Reduce "instructions/program" with "complex" instructions
    - But tends to increase CPI or clock period
  - Easy for assembly-level programmers, good code density
  - Idea: give programmers powerful insts, fewer insts to complete the work
- RISC
  - Improve "cycles/instruction" with many single-cycle instructions
  - Increases "instruction/program", but hopefully not as much
    - Help from smart compiler
  - Idea: compose simple insts to get complex results

CISC Assembly:	RISC Assembly:
IMUL X, Y	LOAD A, X LOAD B, Y PROD A, B STORE X, A

CISC vs. RISC

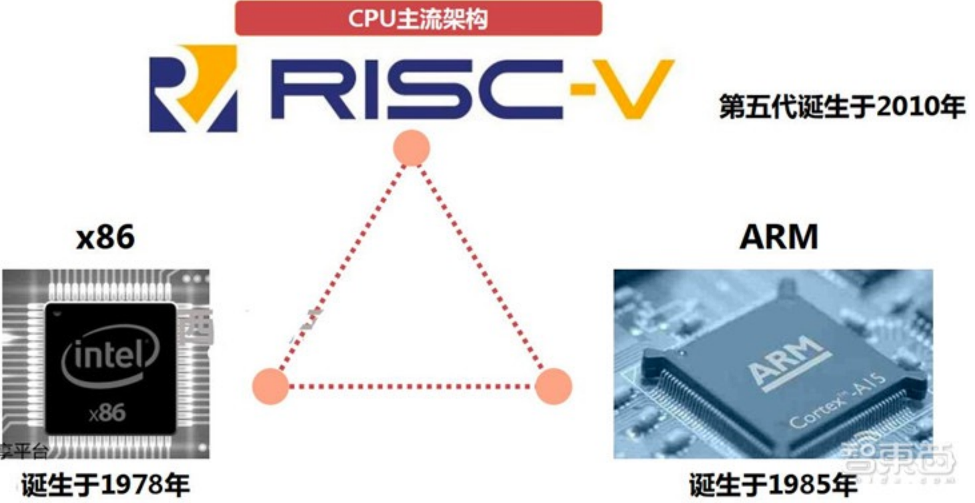
- **Instructions**[指令]: multi-cycle complex vs. single-cycle reduced
- **Addressing modes**[寻址模式]: many vs. few
- **Encoding**[编码]: many formats and lengths vs. fixed-length instruction format
- **Performance**[性能]: hand assemble to get good performance vs. reliance on compiler optimizations
- **Registers**[寄存器]: few vs. many (compilers are better at using them)
- **Code size**[代码大小]: small vs. large

CISC vs. RISC (cont.)

- The war started in mid 1980’s
  - CISC won the high-end commercial war (1990s to today)
    - Compatibility a stronger force than anyone (but Intel) thought
  - RISC won the embedded computing war
- CISC: winner on revenue[赢在收益]
  - X86 was the first 16-bit microprocessor
    - No competing choices → historical inertia and “financial feedback”
  - Moore’s law was the helper
    - Most engineering problems can be solved with more transistors
- RISC: winner on volume[赢在数量]
  - First ARM chip in mid-1980s → 150 billion chips
  - Low-power and embedded devices (e.g., cellphones)

x86 → ARM → RISC-V[进行中的变革]

- But now, things are changing ...
  - Fugaku: ARM-based supercomputer (Top2)
  - Apple: ARM-based M1/2 chip
  - Amazon: AWS Graviton processor
- RISC-V: a freely licensed open standard (Linux in hw)
  - Builds on 30 years of experience with RISC architecture, "cleans up" most of the short-term inclusions and omissions
    - Leading to an arch that is easier and more efficient to implement



What is RISC-V?

- Fifth generation of RISC design from UC Berkeley[第五代]
- A high-quality, license-free, royalty-free RISC ISA[自由]
- Experiencing rapid uptake in both industry and academia[快速发展]
- Supported by growing shared software ecosystem[生态]
- Appropriate for all levels of computing system, from microcontrollers to supercomputers[普适]
  - 32-bit, 64-bit, and 128-bit variants
- Standard maintained by non-profit RISC-V Foundation

https://riscv.org/

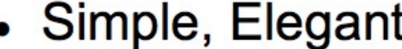
RISC-V (cont.)

- The free and open RISC instruction set architecture
  - Enabling a new era of processor innovation through open standard collaboration[彻底开放]
  - RISC-V ISA delivers a new level of open, extensible software and hardware freedom on architecture, paving the way for the next 50 years of computing design and innovation

What's Different About RISC-V?

("RISC Five", fifth UC Berkeley RISC)

- Free and Open
  - Anyone can use
  - More competition ⇒ More innovation
  - Pick ISA, then vendor
- For Cloud & Edge
  - From large to tiny computers
- Secure/Trustworthy
  - Design own secure core
  - Open cores ⇒ no secrets



- Simple, Elegant
  - 25 years later, learn from 1st gen RISCs*
  - Far simpler than ARM and x86
  - **Can add custom instructions**
  - **Input from software/architecture experts BEFORE finalize ISA**
- Community designed
  - RISC-V Foundation owns RISC-V ISA

The RISC-V Architecture[架构]

- 32, 64-bit general purpose registers (GPRs)
  - called x0, … , x31 (x0 is hardwired to the value 0)
- 32, 64-bit floating point registers - FPRs (each can hold a 32-bit single precision or a 64-bit double precision value)
  - called f0, f1, … , f31
- A few special purpose registers (example: floating point status)
- Byte addressable memories with 64-bit addresses
- 32-bit instructions
- Only immediate and displacement addressing modes (12-bit field)

Data transfer operations: ld, lw, lb, lh, flw, sd, sw, sb, sh, fsw, …
Arithmetic/logical operations: add, addi, sub, subi, slt, and, andi, xor, mul, div, …
Control operations: beq, bne, blt, jal, jalr, …
Floating point operations: fadd, fsub, fmult, fsqrt, …

μ-ops[微操作]

- x86: RISC inside
  - Maintains x86 ISA externally for compatibility
  - But executes RISC μISA internally for implementability
    - x86 code is becoming more "RISC-like"
  - Different μ-ops for different designs
    - Not part of the ISA specification, not publicly disclosed
- Example:

  push $eax

  becomes (we think, μ-ops are proprietary)

  store $eax, -4($esp)

  addi $esp,$esp,-4

Translation and Virtual ISAs[翻译和虚拟]

- New compatibility interface: ISA + translation software
  - Binary-translation: transform static image, run native
  - Emulation: unmodified image, interpret each dynamic inst
  - Typically optimized with just-in-time (JIT) compilation
  - Example: x86 → LoongArch[龙芯]
  - Performance overheads reasonable (many recent advances)

- Virtual ISAs: designed for translation, not direct execution
  - Target for high-level compiler (one per language)
  - Source for low-level translator (one per ISA)
  - Goals: Portability (abstract hardware nastiness), flexibility over time
  - Examples: Java Bytecodes, NVIDIA's "PTX"

Example

鲲鹏DevKit：全流程开发效率提升

一站式迁移、编译、测试、分析

x86

代码迁移	编译调试	性能调优	异常诊断
代码迁移工具 & 加速库插件	编译调试插件	性能分析工具	诊断调试工具
高效迁移 函数加速	鲲鹏亲和 性能更高	全面分析 精准调优	快速诊断 准确定位

有源码迁移 工具化插件与IDE开发环境集成

无源码迁移 平滑迁移、业务无损 华为动态二进制翻译工具（ExaGear）

https://max.book118.com/html/2021/0820/7105126054003163.shtm

RISC-V Instructions[指令]

- All RISC-V instructions are 32 bits long, have 6 formats
  - R-type: instructions using **r**egister-register
  - I-type: instructions with **i**mmediates, loads
  - S-type: **s**tore instructions
  - B-type: **b**ranch instructions (beq, bge)
  - U-type: instructions with **u**pper immediates
  - J-type: **j**ump instructions (jal)

31 … 25	24 … 20	19 … 15	14 … 12	11 … 7	6 … 0	
funct7	rs2	rs1	funct3	rd	opcode	R
imm[11:0]		rs1	funct3	rd	opcode	I
imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode	S
imm[12|10:5]	rs2	rs1	funct3	imm[4:1|11]	opcode	B
imm[31:12]				rd	opcode	U
imm[20|10:1|11]		imm[19:12]		rd	opcode	J

https://riscv.org/wp-content/uploads/2018/05/13.15-13-50-Talk-riscv-base-isa-20180507.pdf

Example

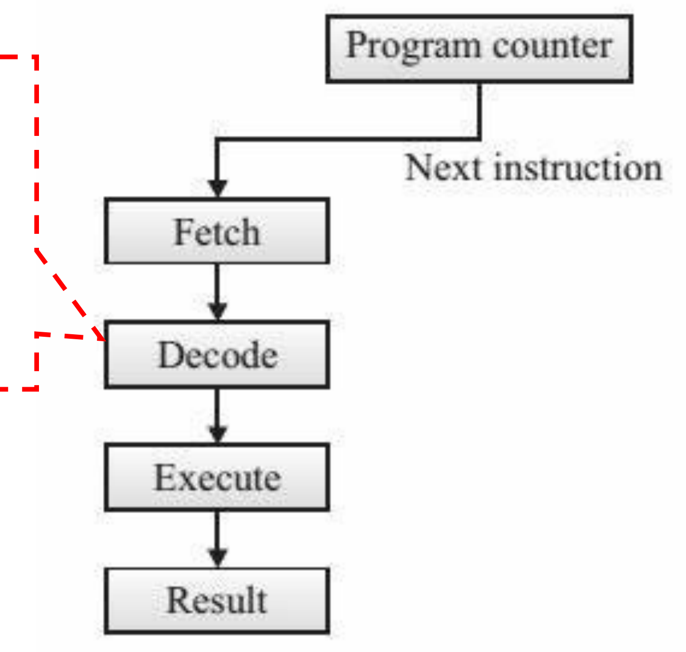
31 25	24 20	19 15	14 12	11 7	6 0
funct7	rs2	rs1	funct3	rd	opcode

- Fields of R-type
  - **opcode**: partially specifies what instruction it is
  - **funct7+funct3**: combined with opcode, these two fields describe what operation to perform
  - **rs1** (source register #1): specifies register of first operand
  - **rs2**: specifies second register operand
  - **rd** (destination register): specifies register which will receive result of computation
    - Each register field holds a 5-bit unsigned integer (0-31) corresponding to a register number (x0-x31)
- add x18,x19,x10

0000000	rs2	rs1	000	rd	0110011
add	rs2=10	rs1=19	add	rd=18	Reg-Reg OP

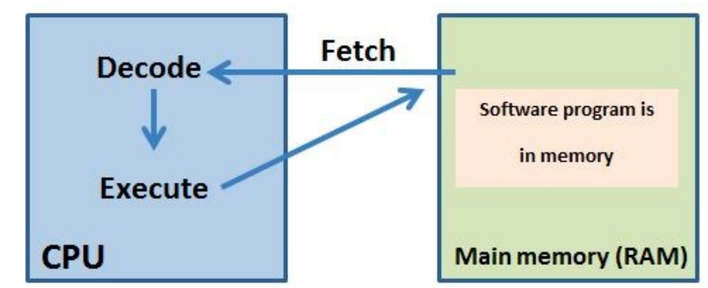
Executing an Instruction[执行指令]

- Very generally, what steps do you take to figure out the effect/result of the next RISC-V instruction?
  - Get the instruction[获取指令]
    - add x18,x19,x10
  - What instruction is it?[操作符]
    - add
  - Gather data read[操作数]
    - R[x19], R[x10]
  - Perform operation[操作]
    - calc R[x19]+R[x10]
  - Store result[结果]
    - save into x18



Five-Stage Execution(§C.1)[5阶段执行]

- **Instruction fetch** (IF)[取指令]/(IM: instruction memory)
  - Fetch the next instruction from memory (and update PC to the next sequential instruction)
- **Instruction decode** (ID)[解码]/(REG: register fetch)
  - Decode the inst and read the registers corresponding to register source specifiers
- **Execution**/effective address (EX)[执行]/(ALU)
  - Operate on the operands prepared in the prior cycle
- **Memory access** (MEM)[访存]/(DM: data memory)
  - Load: read using the effective address
  - Store: write to memory
- **Write-back** (WB)[回写]/(REG)
  - Writes the result into the register



Examples

- Arithmetic/logic instructions: R-type rd, rs1, rs2
  - IF: fetch instruction
  - ID: read registers *rs1* and *rs2*
  - EX: compute result (use ALU)
  - WB: write to register *rd*
- Load instructions: lw rd, c(rs1)
  - IF: fetch instruction
  - ID: read register *rs1*
  - EX: use ALU to compute memory address = content of *rs1* + *c*
  - MEM: read from memory
  - WB: write to register *rd*
- Store instructions: sw rs2, c(rs1)
  - IF: fetch instruction
  - ID: read registers *rs1* and *rs2*
  - EX: use ALU to compute memory address = content of *rs1* + *c*
  - WB: write value of *rs2* to memory at address *rs1+c*

Why Five Stages?

- Could we have a different number of stages?
  - Yes, and other architectures do
- So why does RISC-V have five if instructions tend to idle for at least one stage?
  - The five stages are the union of all the operations needed by all the instructions
  - There is one instruction that uses all five stages: load (lw/lb)

R-type rd, rs1, rs2
IF: fetch instruction
ID: read registers rs1 and rs2
EX: compute result (use ALU)
WB: write to register rd

lw rd, c(rs1)
IF: fetch instruction
ID: read register rs1
EX: use ALU to compute address = rs1 + c
MEM: read from memory
WB: write to register rd

Pipelining[指令流水]

- Pipelining: an implementation technique whereby multiple instructions are overlapped in execution
  - Just like an assembly line
  - Takes advantage of parallelism that exists among the actions needed to execute an instruction
  - Pipelining is the key technique to make fast processors

Stage 1 | Stage 2 | Stage 3

Instr. No.	Pipeline Stage						
1	IF	ID	EX	MEM	WB		
2		IF	ID	EX	MEM	WB	
3			IF	ID	EX	MEM	WB
4				IF	ID	EX	MEM
5					IF	ID	EX
Clock Cycle	1	2	3	4	5	6	7

Visualize Pipelining[表示?]

Cycle 1 Cycle 2 Cycle 3 Cycle 4 Cycle 5 Cycle 6 Cycle 7 Cycle 8

add $4, $5, $6 — IM → REG → EX → DM → WB

and $1, $2, $3 — IM → REG → EX → DM → WB

lw $3, 300($0) — IM → REG → EX → DM → WB

sub $7, $8, $9 — IM → REG → EX → DM → WB

	IM (Fetch)	ID (Reg)	EX (ALU)	DM (memory)	WB
Cycle 1	add $4, $5, $6				
Cycle 2	and $1, $2, $3	add $4, $5, $6			
Cycle 3	lw $3, 300($0)	and $1, $2, $3	add $4, $5, $6		
Cycle 4	sub $7, $8, $9	lw $3, 300($0)	and $1, $2, $3	add $4, $5, $6	

Revisiting Pipeline

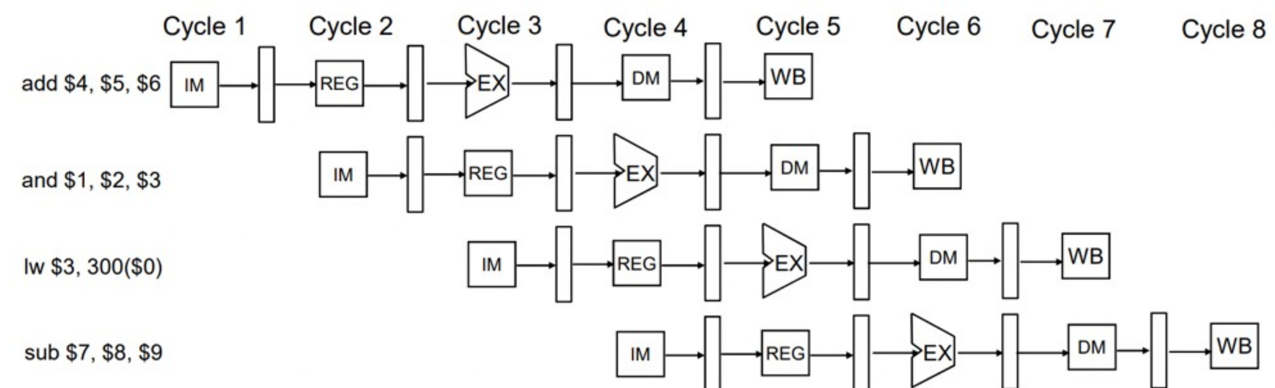
- Automobile assembly line[汽车流水线]
  - How often a completed car exits the assembly line (i.e., #cars/hour)[产能]
  - The longest step determines the time between advancing the line
- Instruction pipelining[指令流水线]
  - **Throughput**: how often an instruction exits the pipeline[吞吐]
  - **Processor cycle**: the time required between moving an inst one step down the pipeline
    - The length of a processor cycle is determined by the time required for the slowest pipe stage
    - Usually 1 clock cycle
- Pipeline designer should balance the length of each stage

Pipelining Effects[效果]

- If stages are perfectly balanced, then the time per inst on the pipelined processor (assuming ideal conditions)
  - $= \frac{\textit{Time per instruction on unpipelined machine}}{\textit{Number of pipe stages}}$
- Speedup from pipelining equals the number of stages
  - An assembly pipeline with *n* stages can ideally produce cars *n* times fast
  - Instruction exit: every *n* cycles vs. every single cycle
- Pipelining reduces the avg execution time per inst
  - Baseline of multi clock cycles/inst: pipelining reduces CPI
  - Baseline of single clock cycle/inst: pipelining decreases the clock cycle time

Pipelining Effects (cont.)

- Pipelining exploits parallelism among the insts[并行]
  - Not visible to the programmer
- Pipelining improves instruction throughput rather instruction latency[提高吞吐]
  - Goal is to make programs, not individual insts, go faster
  - Single instruction latency
    - Doesn't really matter, billions of insts in a program
    - Difficult to reduce anyway
  - In fact, pipelining usually slightly increases the execution time of each inst



中山大學 SUN YAT-SEN UNIVERSITY

NSCC GZ

Performance Issues in Pipelining[问题]

- Impossible to reach the ideal speedup (= *n* stages)
  - Usually, the stages will not be perfectly balanced[并不平衡]
    - The clock can run no faster than the time needed for the slowest pipeline stage
  - Furthermore, pipelining does involve some overhead[额外开销]
    - Pipeline register delay + clock skew[时钟漂移]

Example: one unpipelined processor has 1ns clock cycle and instructions are ALUs (4 cycles, 40%), branches (4 cycles, 20%), memory (5 cycles, 40%). Suppose that pipelining the processor adds 0.2ns overhead to the clock. How much pipelining speedup?

- Unpipelined processor, avg inst exe time = clock cycle x avg CPI = 1 ns x (40%x4 + 20%x4 + 40%x5) = 4.4ns
- Pipelined processor, avg inst exe time = 1 + 0.2 ns = 1.2 ns

Dependences and Hazards[依赖和冒险]

- **Dependence**[依赖]: relationship between two insts
  - Data: two insts use same storage location
  - Control: one inst affects whether another executes at all
  - Not a bad thing, programs would be boring without them
  - Enforced by making older inst go before younger one
    - Happens naturally in single-/multi-cycle designs
    - But not in a pipeline
- **Hazard**[冒险]: dependence & possibility of wrong inst order
  - Effects of wrong inst order cannot be externally visible
    - Stall: for order by keeping younger inst in same stage
  - Hazards are a bad thing: stalls reduce performance

Pipeline Hazards (§C.2)[流水冒险]

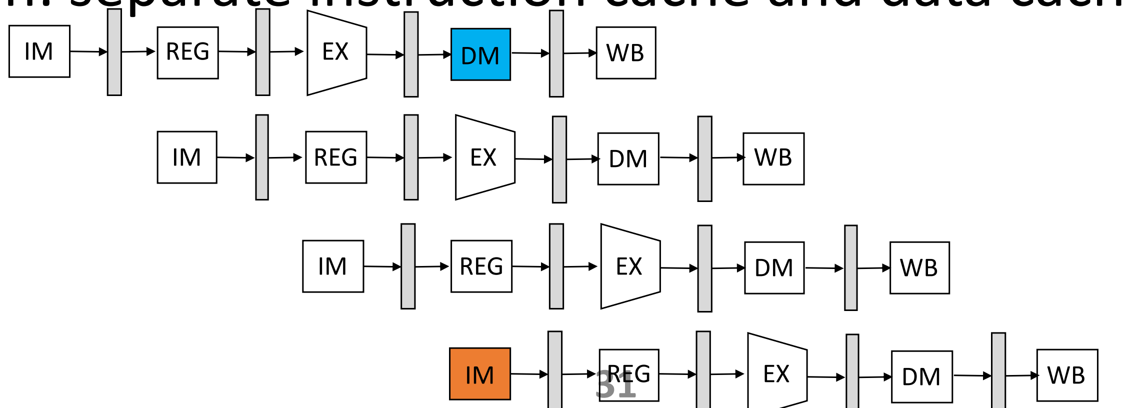
- Hazards prevent next instruction from executing during its designated clock cycle[妨碍执行]
  - Hazards reduce the performance from the ideal speedup gained by pipelining
- Three classes of hazards
  - **Structural hazards**[结构]: HW cannot support some combination of instructions
  - **Data hazards**[数据]: An instruction depends on result of prior instruction still in the pipeline
  - **Control hazards**[控制]: Pipelining of branches & other instructions stall the pipeline until the hazard bubbles in the pipeline

Structural Hazards[结构冒险]

- Different instructions are using the same resource at the same time[资源冲突]
  - Some functional unit is not fully pipelined
    - Then a sequence of insts using that unpipelined unit cannot proceed at the rate of one per clock cycle
  - Some resource has not been duplicated enough
- To fix structural hazards: proper ISA/pipeline design[解决]
  - Each inst uses every structure exactly one, for at most one cycle
  - Always at the same stage relative to Fetch
- Tolerate structural hazards[容忍]
  - Add stall logic to stall pipeline when hazards occur

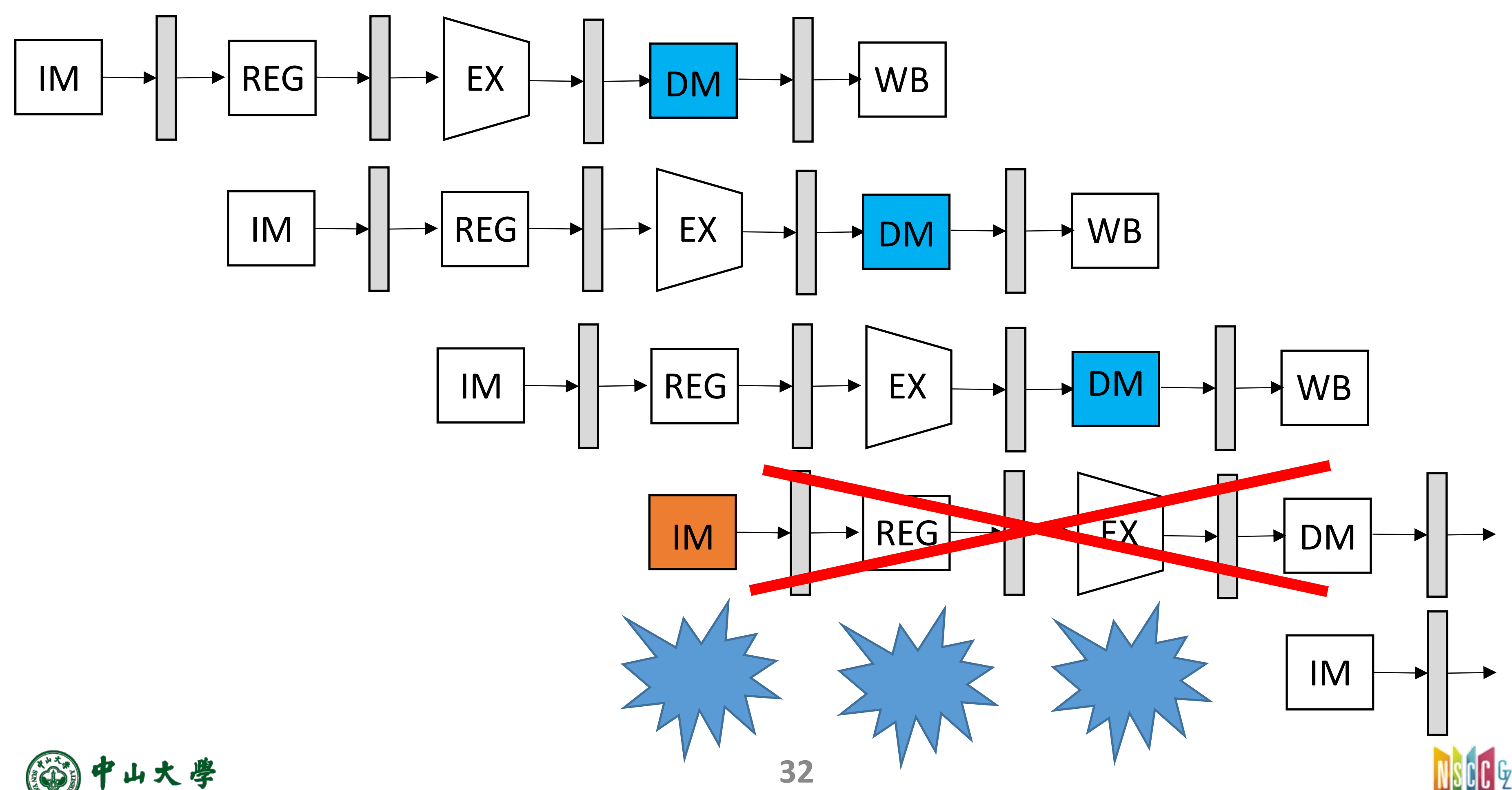
Structural Hazards (cont.)

- Register file[寄存器]
  - Accessed in two stages
    - Read during stage 2 (ID or REG)
    - Write during stage 5 (WB)
  - Solution: one read port, one write port
- Memory[内存]
  - Accessed in two stages
    - Instruction fetch during stage 1 (IF or IM)
    - Data read/write during stage 4 (MEM)
  - Solution: separate instruction cache and data cache



Structural Hazards (cont.)

- Bubbles are inserted
  - Wasted cycles → performance loss



SUN YAT-SEN UNIVERSITY
中山大學
NSCC GZ

Data Hazards[数据冒险]

- Pipeline changes the order of read/write accesses to operands
  - The order may differ from the order seen by sequentially executing insts on an unpipelined processor
- Example
  - All the instructions after the DADD use the result of the DADD instruction
    - What if the old result is being accessed?
      - DADD writes into R1, happening in stage 5 (WB)
      - DSUB reads from R1, happening in stage 2 (ID)

```
DADD    R1, R2, R3
DSUB    R4, R1, R5
AND     R6, R1, R7
OR      R8, R1, R9
XOR     R10, R1, R11
```

Data Hazards (cont.)

CC 1 CC 2 CC 3 CC 4 CC 5

DADD R1, R2, R3 — IM → REG → EX → DM → WB

DSUB R4, R1, R5 — IM → REG → EX → DM → WB

ADD R6, R1, R7 — IM → REG → EX → DM

OR R8, R1, R9 — IM → REG → EX

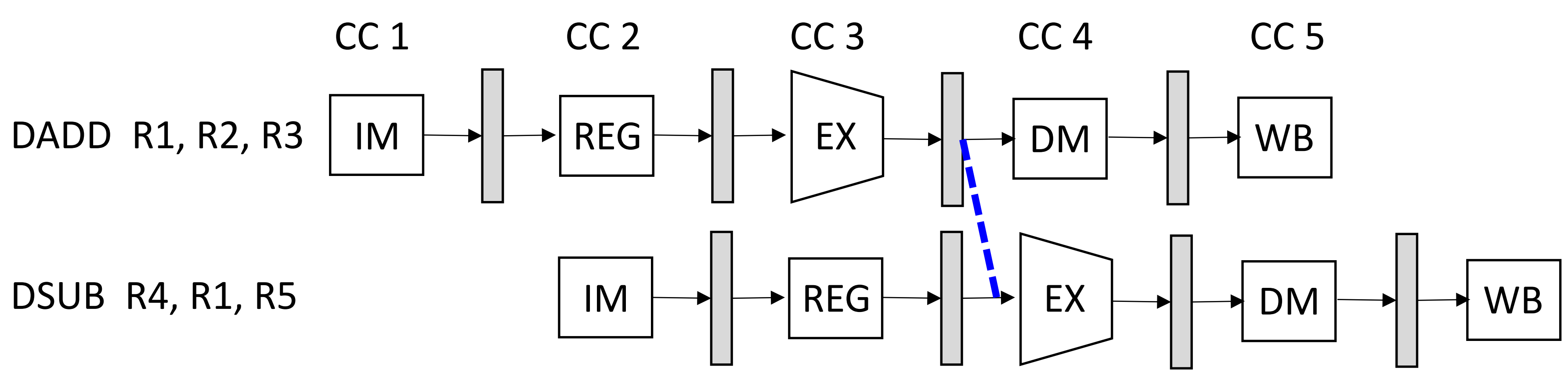
XOR R10, R1, R11 — IM → REG

Program order

- DSUB is not even deterministic
  - Right, if an interrupt occurs between DADD and DSUB
  - Wrong, otherwise

Forwarding[转发]

- Minimizing data hazards stalls by forwarding
  - a.k.a., bypassing, short-circuiting
  - The result is not really needed by the *DSUB* until after the *DADD* actually produces it
  - If the result can be moved from the pipeline register where the *DADD* stores it to where the *DSUB* needs it, then the need for a stall can be avoided



Forwarding (cont.)

	CC 1	CC 2	CC 3	CC 4	CC 5		
DADD R1, R2, R3	IM	REG	EX	DM	WB		
DSUB R4, R1, R5		IM	REG	EX	DM	WB	
ADD R6, R1, R7			IM	REG	EX	DM	
OR R8, R1, R9				IM	REG	EX	
XOR R10, R1, R11					IM	REG	

- ALU inputs could use forwarded inputs from either the same pipeline register or from different pipeline registers

Forwarding is Insufficient[仅转发不够]

	CC 1	CC 2	CC 3	CC 4	CC 5		
LD R1, 0(R2)	IM	REG	EX	DM	WB		
DSUB R4, R1, R5		IM	REG	EX	DM	WB	
AND R6, R1, R7			IM	REG	EX	DM	
OR R8, R1, R9				IM	REG	EX	

- LD can bypass its results to AND and OR instructions
- But not to the DSUB
  - Forwarding the result in “negative time” !