中山大學 SUN YAT-SEN UNIVERSITY

国家超级计算广州中心 NATIONAL SUPERCOMPUTER CENTER IN GUANGZHOU

Advanced Computer Architecture

高级计算机体系结构

第11讲：概述、ISA&ILP (2)

张献伟

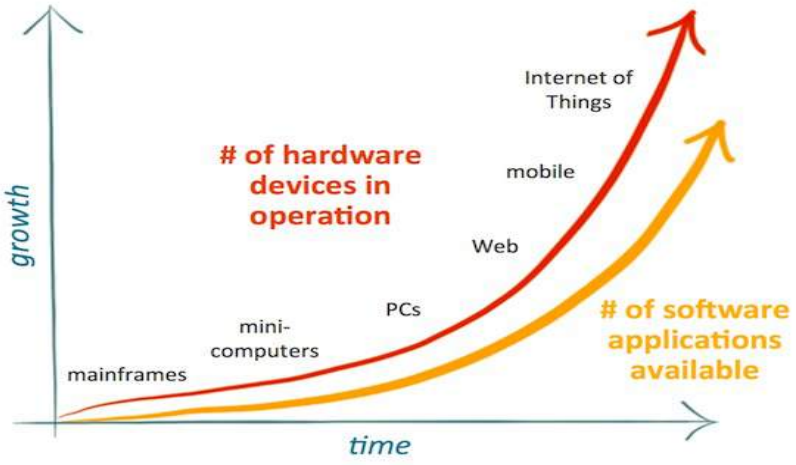
xianweiz.github.io

DCS5637, 11/9/2022

Part-III: ISA & ILP

The History

- For more than 50 years, we have enjoyed exponentially increasing compute power[算力急剧增长]
- The growth is based on a fundamental contract between HW and SW[得益于软硬件之间的协议]
  - HW may change radically “under the hood”
    - Old SW can still run on new HW (even faster)
  - HW looks the same to SW, always speaking the same language
    - The **ISA**, allows the decoupling of SW development from HW dev



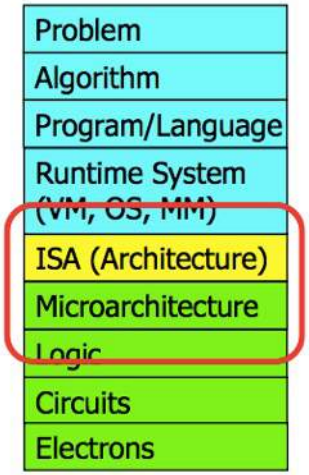
What is ISA?

- Instruction Set == A set of instructions
- The HW/SW contract[软硬件协议]
  - Compiler correctly translates source code to the ISA[编译器]
  - Assembler translates to relocatable binary[汇编器]
  - Linker solidifies relocatables into object code[连接器]
  - HW promises to do what the object code says[硬件执行]

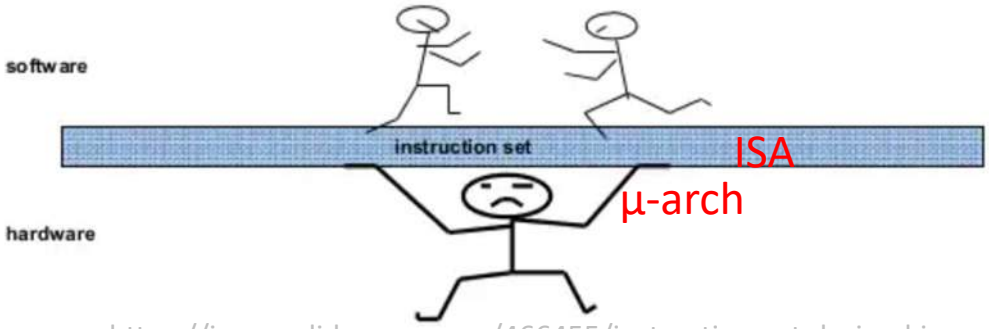
- Not in the “contract”: non-functional aspects[非协议]
  - How operations are implemented
  - Which operations are fast and which are slow and when
  - Which operations take more power and which take less

ISA + μ-arch = Arch

- "Architecture" = ISA + microarchitecture
- ISA[指令集架构]
  - Agreed upon interface between sw and hw
    - SW/compiler assumes, HW promises
  - What the software writer needs to know to write and debug system/user programs



- Microarchitecture (μ-arch)[微架构]
  - Specific implementation of an ISA
    - Implementation of the ISA under specific design constraints and goals
  - Not visible to the software

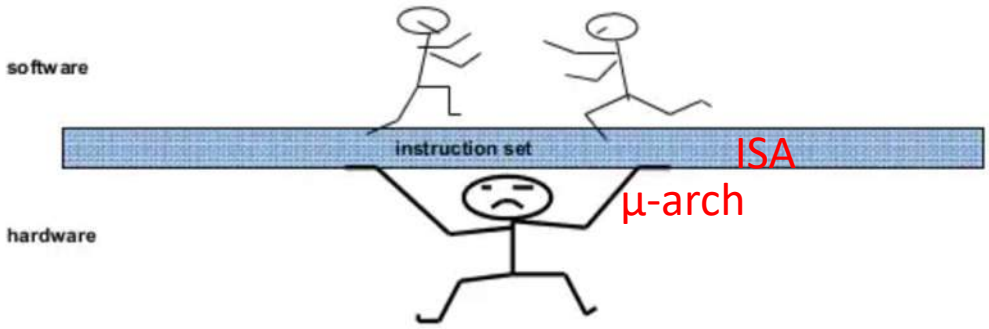


https://image.slideserve.com/466455/instruction-set-design-l.jpg

ISA vs. μ-arch (cont.)

- Implementation (μ-arch) can be various as long as it satisfies the specification (ISA)
  - Add instruction vs. Adder implementation
    - Bit serial, ripple carry, carry lookahead adders are all part of microarchitecture
  - x86 ISA has many implementations: 286, 386, 486, Pentium, Pentium Pro, Pentium 4, Core, …

- μ-arch usually changes faster than ISA
  - Few ISAs (x86, ARM, SPARC, MIPS, Alpha) but many μ-archs



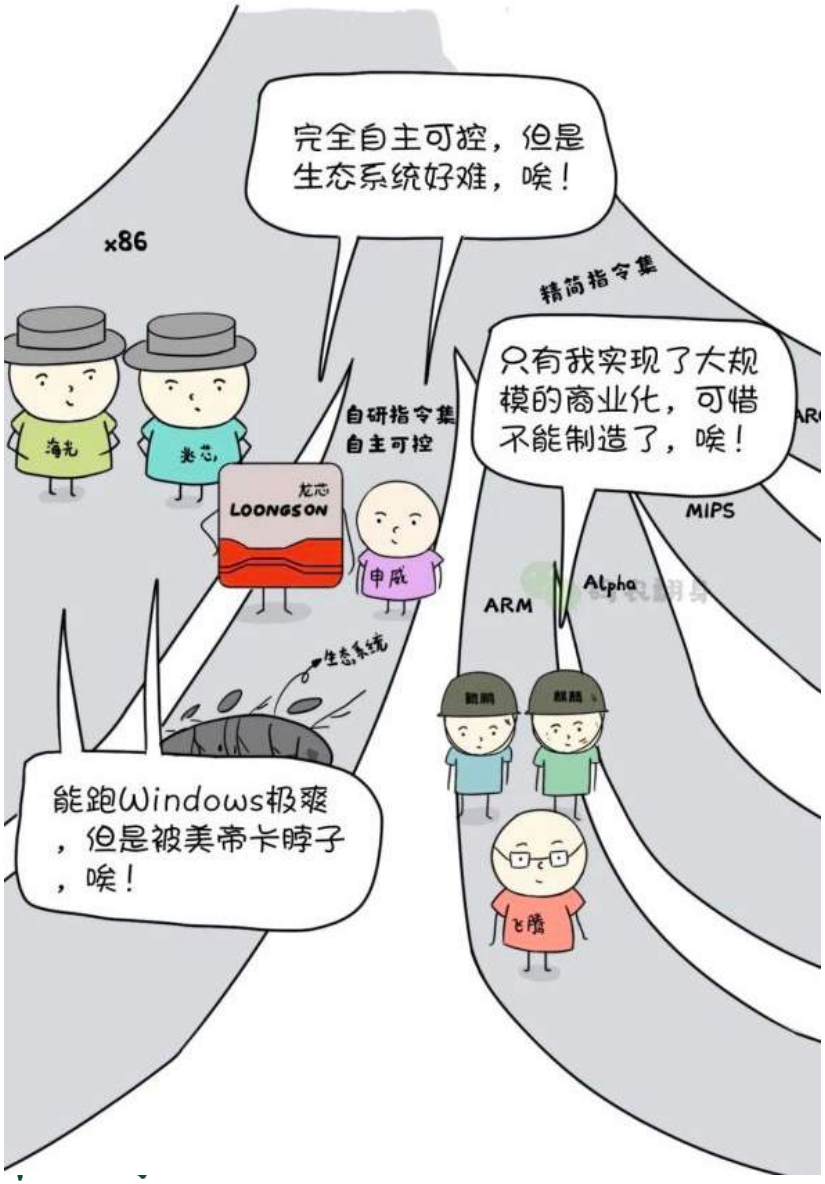
What Makes a Good ISA?

- Programmability[可编程性]
  - Easy to express programs efficiently?
- Implementability[可实现性]
  - Easy to design high-performance implementations?
  - More recently
    - Easy to design low-power implementations?
    - Easy to design high-reliability implementations?
    - Easy to design low-cost implementations?
- Compatibility[兼容性]
  - Easy to maintain programmability (implementability) as languages and programs (technology) evolves?
  - x86 (IA32) generations: 8086, 286, 386, 486, Pentium, PentiumII, PentiumIII, Pentium4, Core2…

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.

国产CPU



- 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

RISC-V®

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

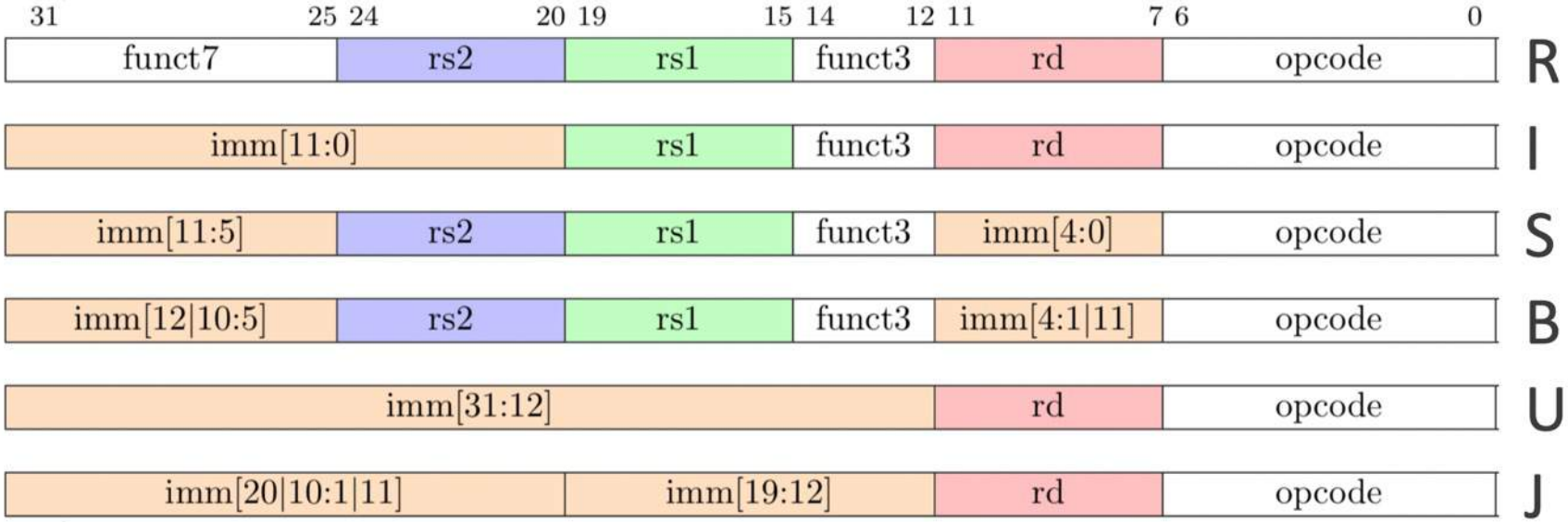
The RISC-V Architecture[架构]

- 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, …

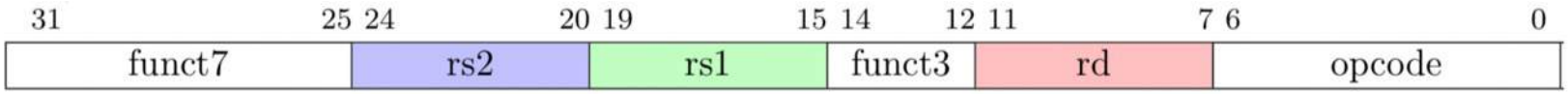
RISC-V Instructions[指令]

- 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)

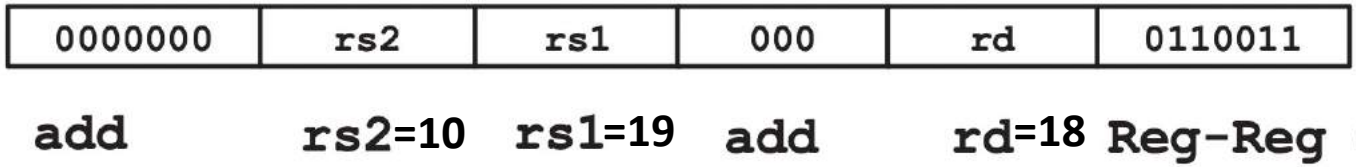


https://riscv.org/wp-content/uploads/2018/05/13.15-13-50-Talk-riscv-base-isa-20180507.pdf

Example

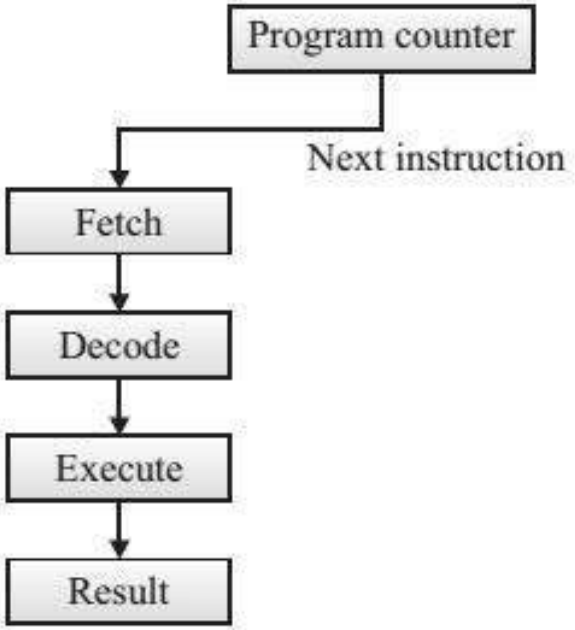


- Fields of R-type
  - **opcode**: partially specifies what instruction it is
  - **funct7**+**funct3**: combined with opcode, these two fields describe what operation to perform
  - **rs1** (source register #1): specifies register of first operand
  - **rs2**: specifies second register operand
  - **rd** (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



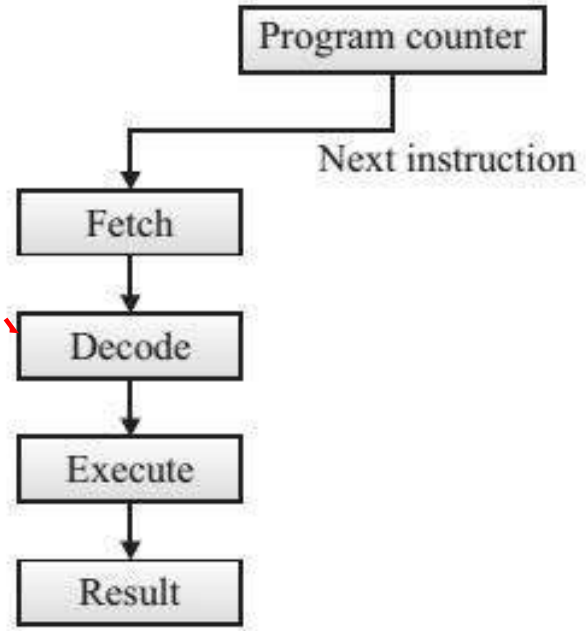
Executing an Instruction[执行指令]

- 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



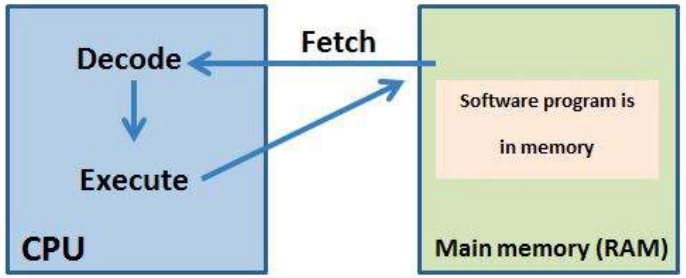
Executing an Instruction[执行指令]

- 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

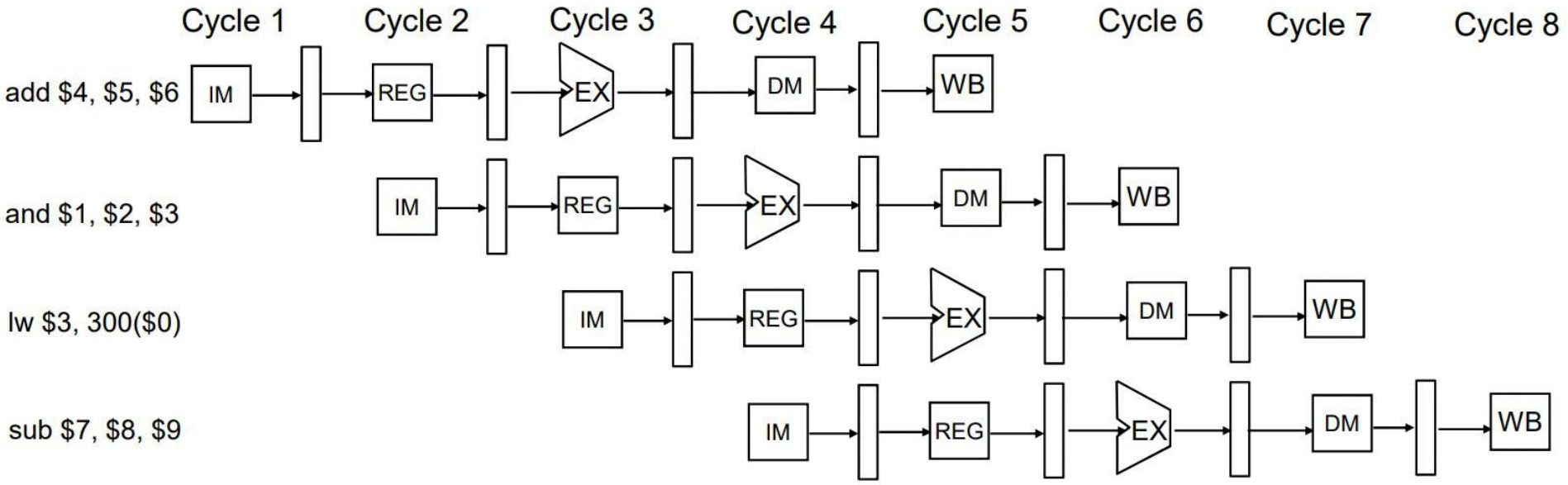
lw rd, c(rs1)
IF: fetch instruction
ID: read register rs1
EX: use ALU to compute address = rs1 + c
MEM: read from memory
WB: write to register rd

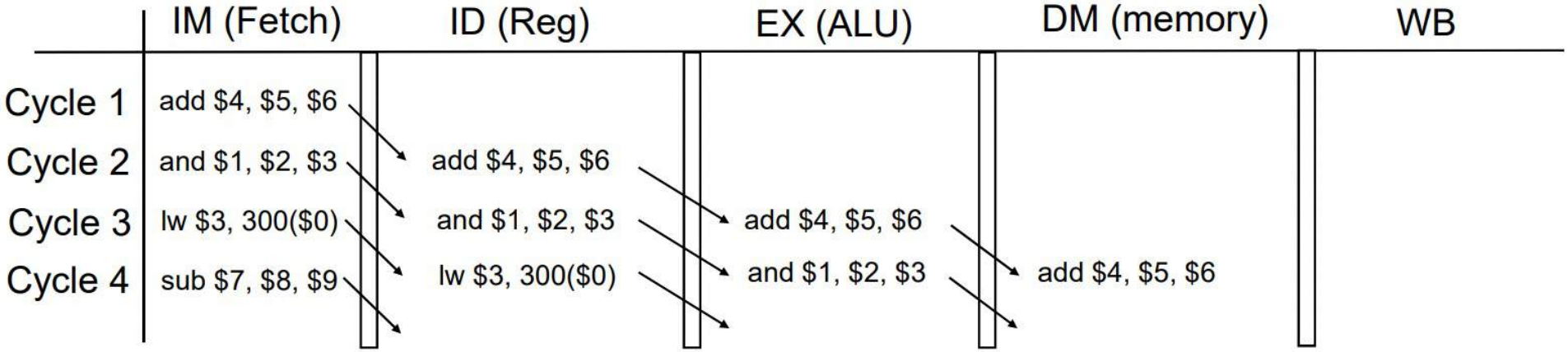
Pipelining[指令流水]

- 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

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?]



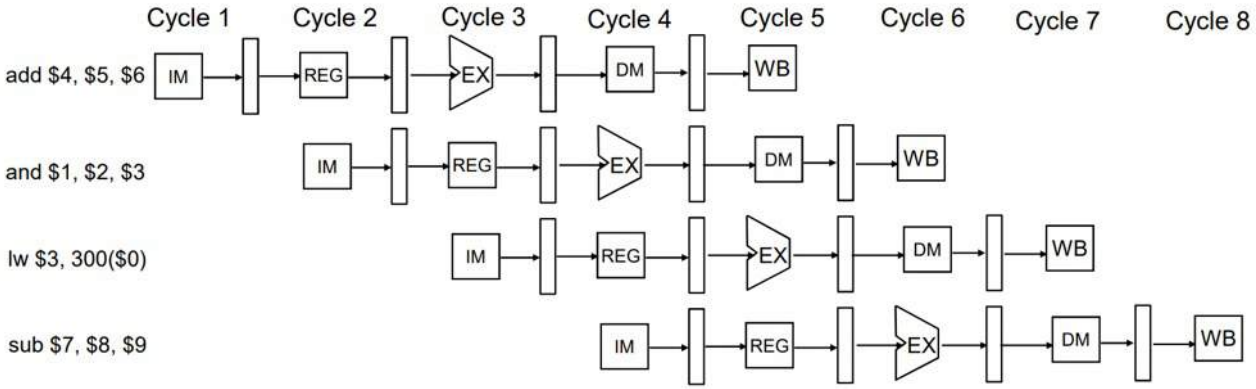


Pipelining Effects[效果]

- If stages are perfectly balanced, then the time per inst on the pipelined processor (assuming ideal conditions)
  - $\frac{Time\ per\ instruction\ on\ unpipelined\ machine}{Number\ of\ pipe\ stages}$
- 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



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

Instruction-Level Parallelism(§3.1)

- ILP: overlap execution of instructions[指令级并行]
  - Overlap among instructions[重叠]
    - Pipelining or multiple instruction execution
  - Fine-grained parallelism[细粒度]
    - In contrast to process-/task/thread-level parallelism (coarse-grained)
- Pipelining: exploits ILP by executing several instructions “in parallel”
  - Overlaps execution of different instructions
  - Execute all steps in the execution cycle simultaneously, but on different instructions
- Pipeline CPI = Ideal pipeline CPI + stalls due to hazards
  - Structural stalls + Data hazard stalls + Control stalls

Instruction-Level Parallelism(cont.)

- Approaches to exploit ILP[利用方法]
  - Rely on hardware to help discover and exploit the parallelism dynamically
  - Rely on software technology to find parallelism, statically at compile-time
- What determines the degree of ILP?[并行度]
  - Dependences: property of the program
  - Hazards: property of the pipeline (or the architecture)
- ILP challenge: overcoming data and control dependencies

Techniques to Improve ILP

Technique	Reduces	Section
Forwarding and bypassing	Potential data hazard stalls	C.2
Simple branch scheduling and prediction	Control hazard stalls	C.2
Basic compiler pipeline scheduling	Data hazard stalls	C.2, 3.2
Basic dynamic scheduling (scoreboarding)	Data hazard stalls from true dependences	C.7
Loop unrolling	Control hazard stalls	3.2
Advanced branch prediction	Control stalls	3.3
Dynamic scheduling with renaming	Stalls from data hazards, output dependences, and antidependences	3.4
Hardware speculation	Data hazard and control hazard stalls	3.6
Dynamic memory disambiguation	Data hazard stalls with memory	3.6
Issuing multiple instructions per cycle	Ideal CPI	3.7, 3.8
Compiler dependence analysis, software pipelining, trace scheduling	Ideal CPI, data hazard stalls	H.2, H.3
Hardware support for compiler speculation	Ideal CPI, data hazard stalls, branch hazard stalls	H.4, H.5

Types of Dependences[依赖类型]

- True **data dependences**: may cause RAW hazards[数据]
  - Instruction *Q* uses data produced by instruction *P* or by an instruction which is data dependent on *P*
  - Easy to determine for registers but hard to determine for memory locations since addresses are computed dynamically
    - Example: is 100(R1) the same location as 200(R2)?
- **Name dependences**: two instructions use the same name but do not exchange data (no data dependency)[名字]
  - **Anti-dependence**[反依赖]: instruction *P* reads from a register (or memory) followed by instruction *Q* writing to that register (or memory). May cause WAR hazards
  - **Output dependence**[输出依赖]: instructions *P* and *Q* write to the same location. May cause WAW hazards.

Example

```
Loop:   fld      f0, 0(x1)
        fadd.d   f4, f0, f2
        fsd      f4, 0(x1)
        fld      f0, -8(x1)
        fadd.d   f4, f0, f2
        fsd      f4, -8(x1)
        addi     x1, x1, #-16
        bne      x1, x0, loop
```

- Data dependence
  - RAW: read after write
- Anti-dependence
  - WAR: write after read
- Output dependence
  - WAW: write after write

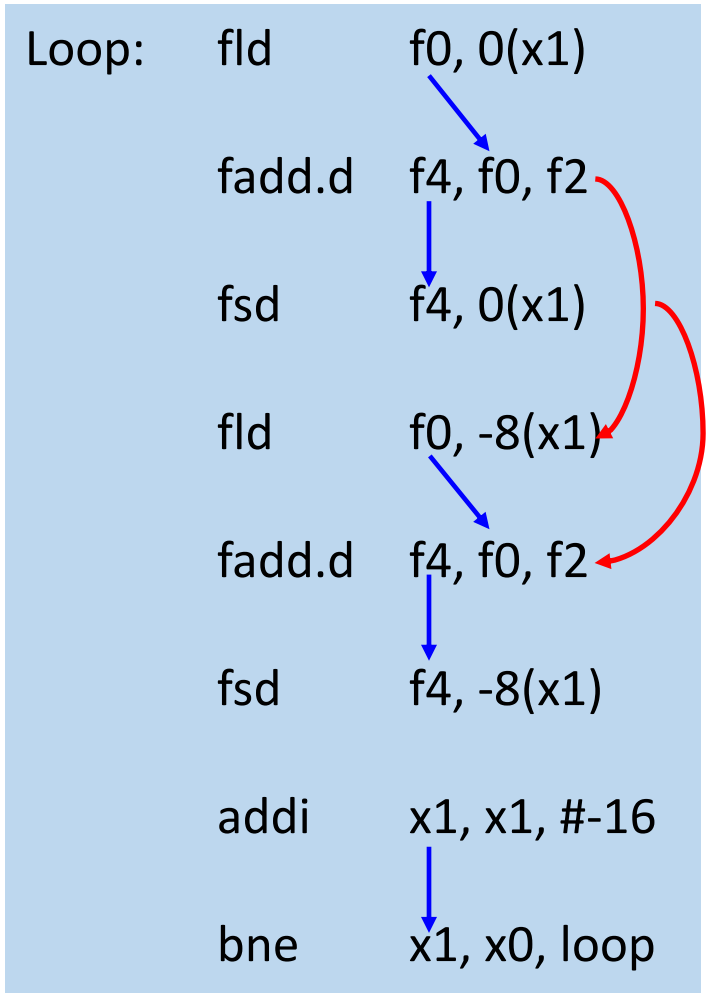
Example

```
Loop:   fld      f0, 0(x1)
        fadd.d   f4, f0, f2
        fsd      f4, 0(x1)
        fld      f0, -8(x1)
        fadd.d   f4, f0, f2
        fsd      f4, -8(x1)
        addi     x1, x1, #-16
        bne      x1, x0, loop
```

- Data dependence
  - RAW: read after write
- Anti-dependence
  - WAR: write after read
- Output dependence
  - WAW: write after write

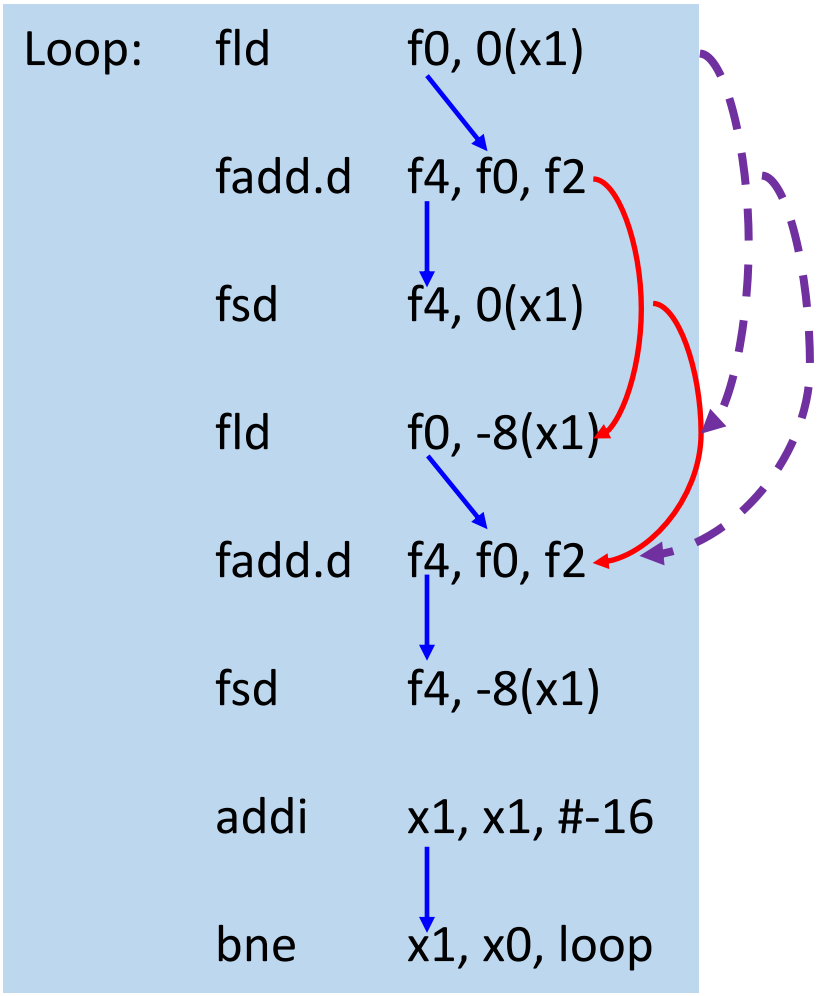
Example

```
Loop:   fld      f0, 0(x1)
        fadd.d   f4, f0, f2
        fsd      f4, 0(x1)
        fld      f0, -8(x1)
        fadd.d   f4, f0, f2
        fsd      f4, -8(x1)
        addi     x1, x1, #-16
        bne      x1, x0, loop
```



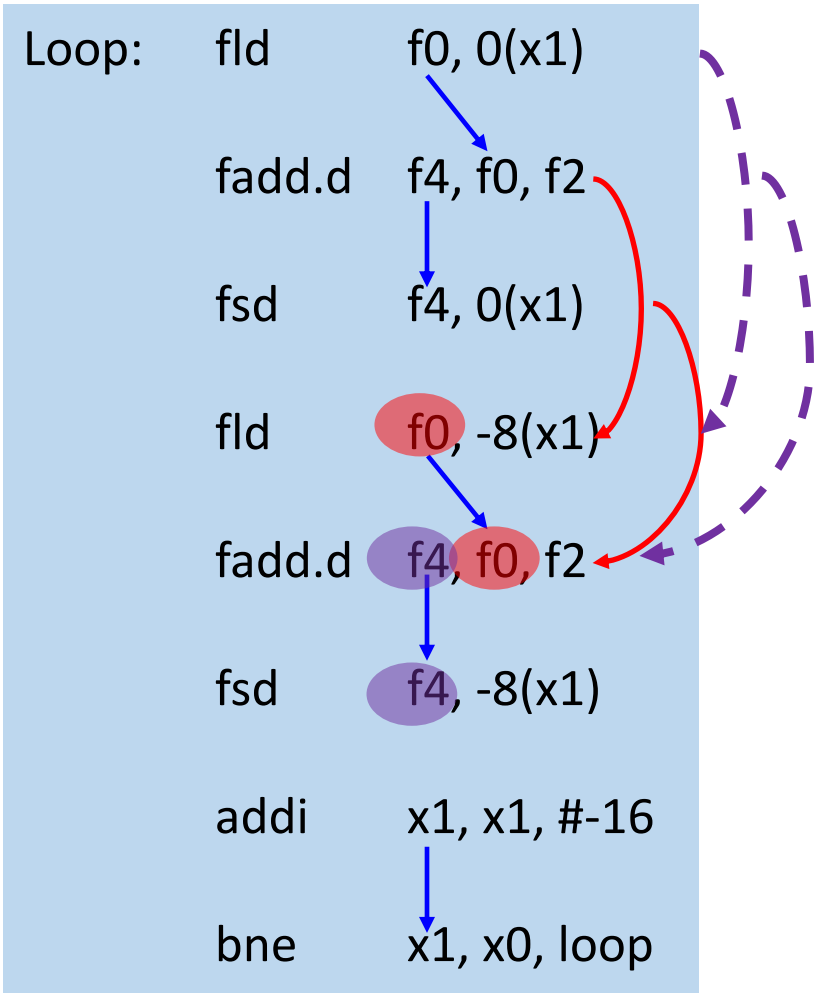
- Data dependence
  - RAW: read after write
- Anti-dependence
  - WAR: write after read
- Output dependence
  - WAW: write after write

Example



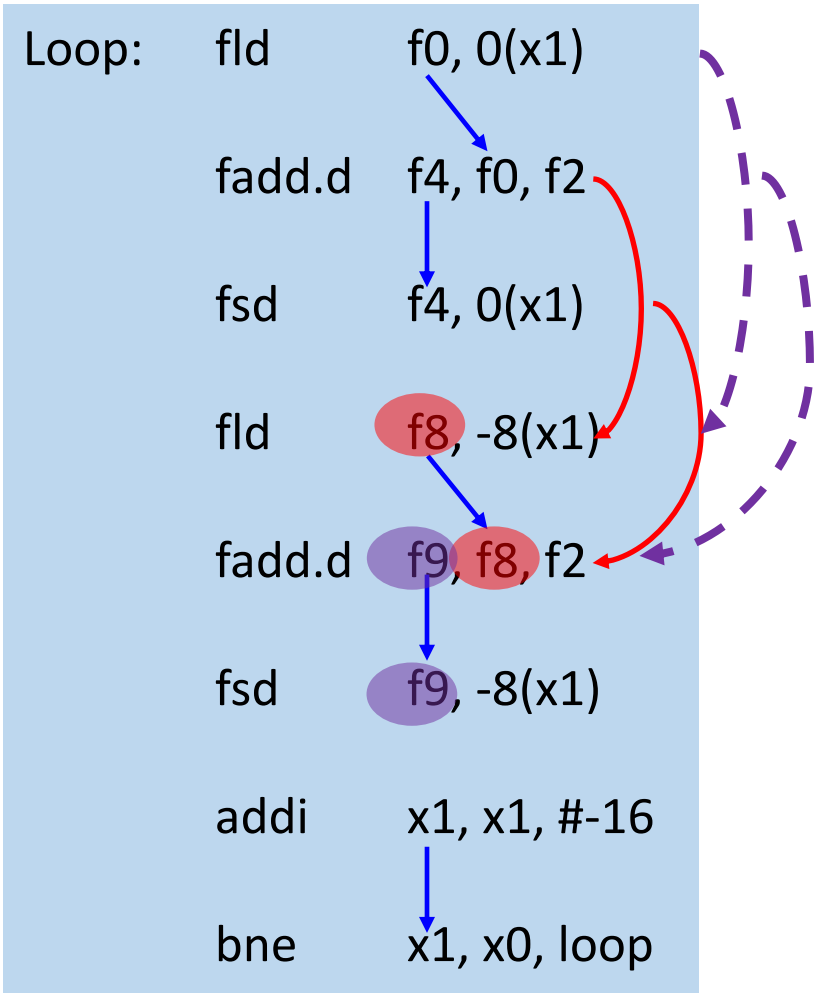
- Data dependence
  - RAW: read after write
- Anti-dependence
  - WAR: write after read
- Output dependence
  - WAW: write after write

Example



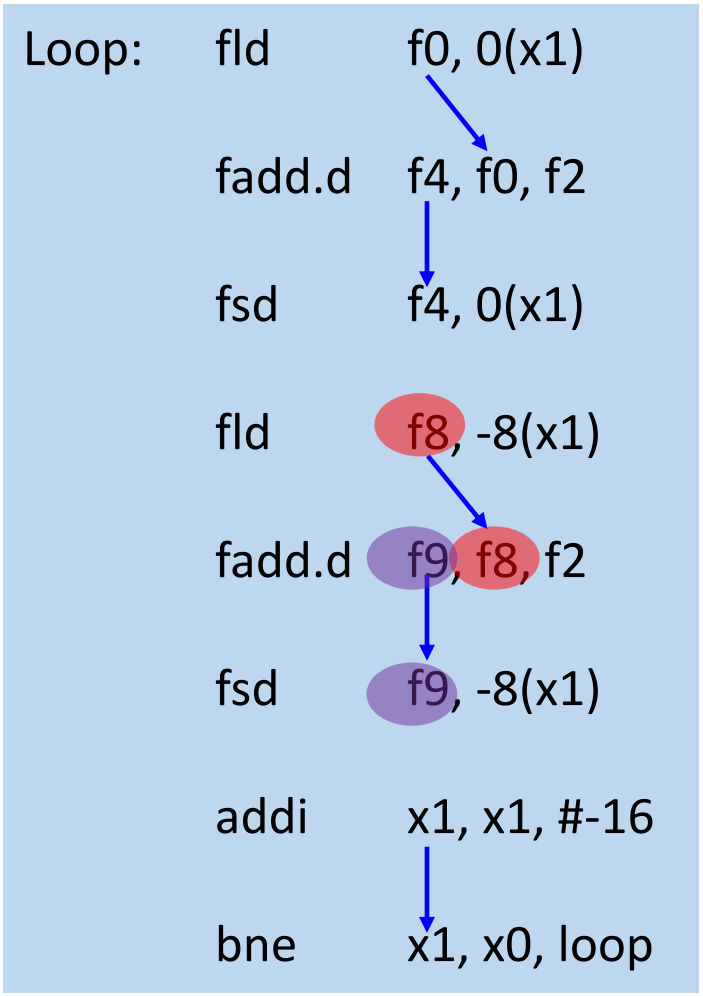
- Data dependence
  - RAW: read after write
- Anti-dependence
  - WAR: write after read
- Output dependence
  - WAW: write after write

Register Renaming[重命名]



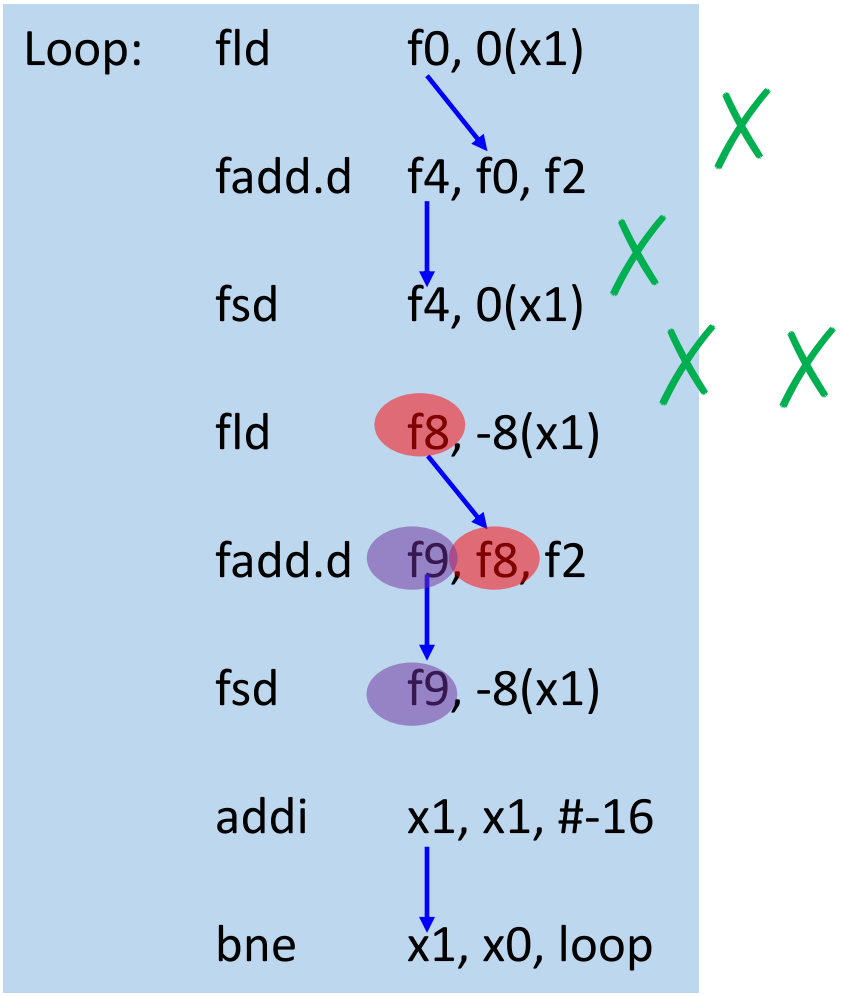
- How to remove name dependences?
  - Rename the dependent uses of f0 and f4

Register Renaming[重命名]



- How to remove name dependences?
  - Rename the dependent uses of f0 and f4

Register Renaming[重命名]



- How to remove name dependences?
  - Rename the dependent uses of f0 and f4

Control Dependences[控制依赖]

- Determine the order of instructions with respect to branches[相对分支的指令顺序]

  if *P1* then *S1* ;
  if *P2* then *S2* ;

  S1 is control dependent on *P1* and
  S2 is control dependent on *P2* (and P1 ??)

- An instruction that is control dependent on *P* cannot be moved to a place where it is no longer control dependent on *P*, and visa-versa[不可移动]

```
Example 1:
   add    x1, x2, x3
   beq    x4, x0, L
   sub    x1, x5, x6
L: …
    or    x7, x1, x8
```

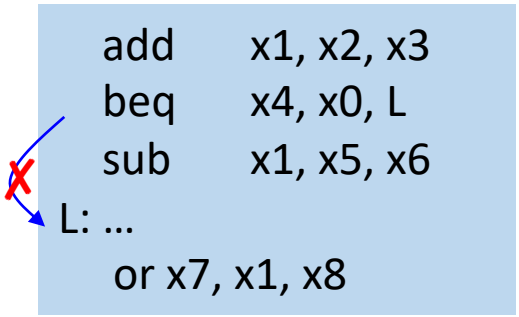
"or" depends on the execution flow

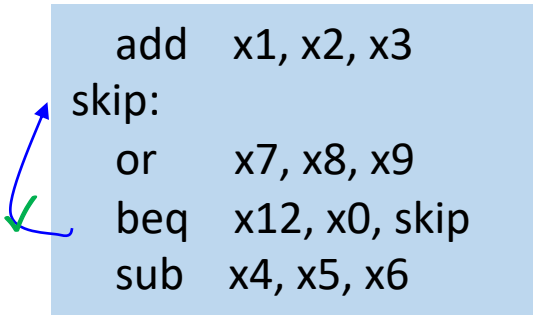
```
Example 2:
   add    x1, x2, x3
   beq    x12, x0, skip
   sub    x4, x5, x6
   add    x5, x4, x9
skip:
    or    x7, x8, x9
```

possible to move "sub" before "beq" (if x4 is not used after skip)

Branch Prediction(§3.3)[分支预测]

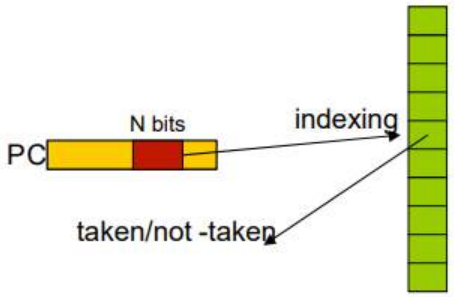
- Branches hurt pipeline performance
  - Branch hazards and stalls
- Static branch prediction[静态分支预测]
  - The default is to assume that branches are not taken
  - May have a design which predicts that branches are taken
- Reasonable to assume that[假设]
  - Forward branches are often not taken
  - Backward branches are often taken
- More predictors based on branch directions
  - Profiling is the standard technique for predicting the probability of branching
  - Dynamic predictors rely on the history to predict the future branch direction

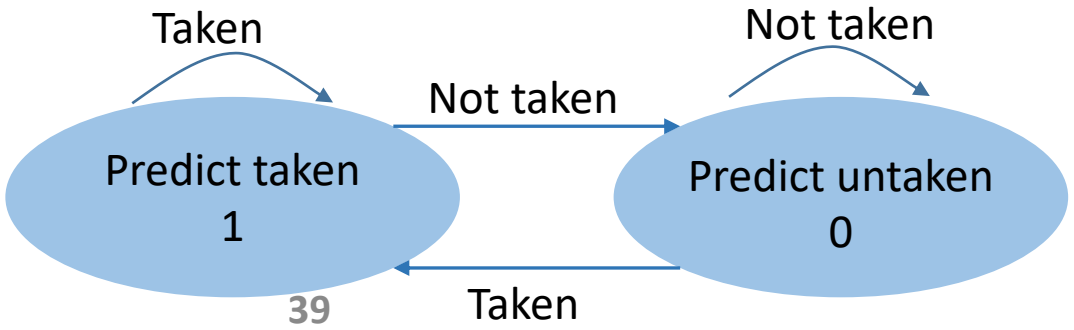




Dynamic Branch Prediction(§C2.7)[动态]

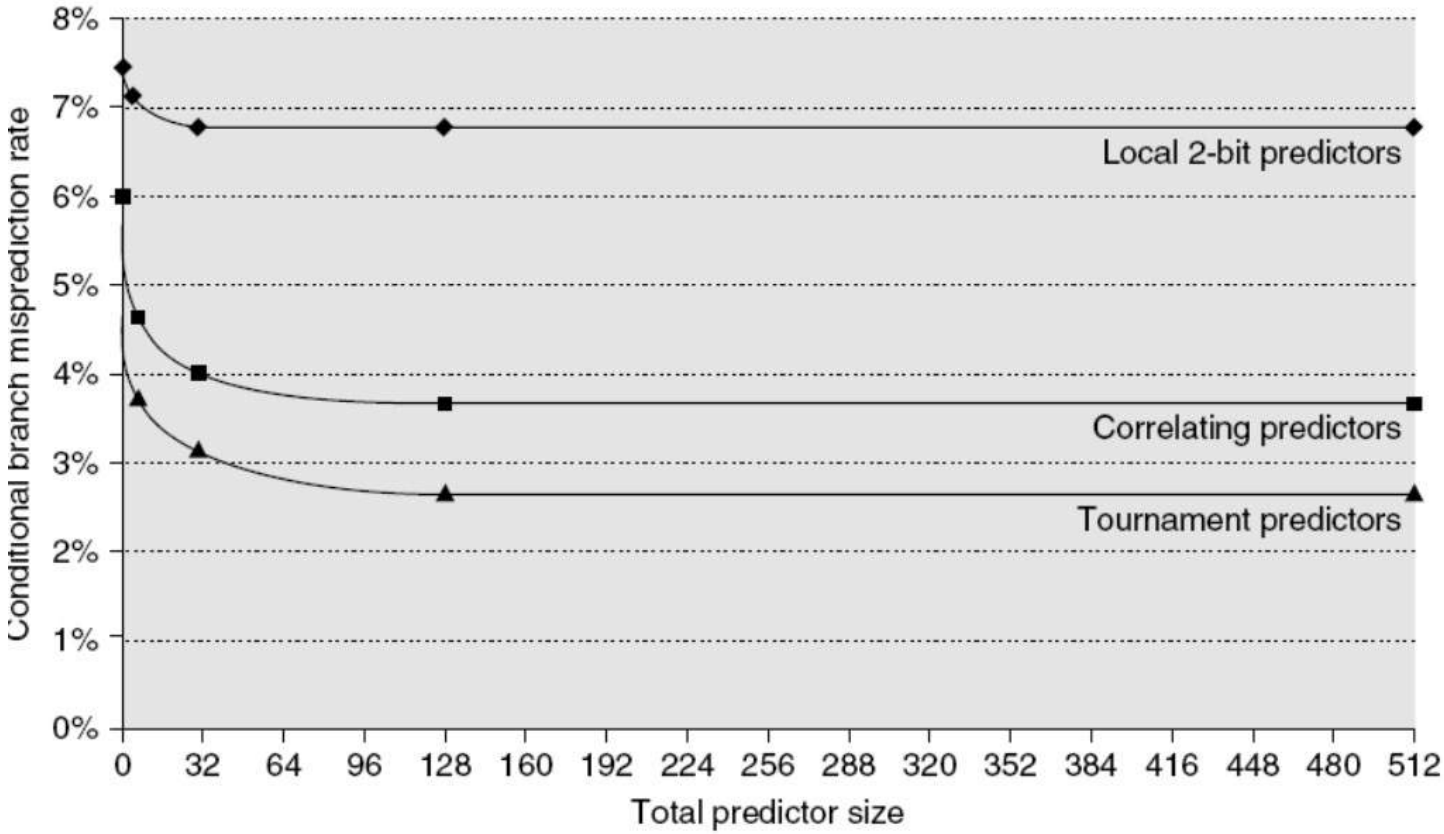
- Performance depends on the accuracy of prediction and the cost of miss-prediction[性能影响]
- The simplest branch prediction scheme: **Branch Prediction Buffer**[分支预测缓存]
  - 1-bit table (cache) indexed by some bits of the address of the branch instructions (can be accessed in decode stage) -> hashing[指令地址的低位作为索引]
  - Record whether or not the branch was taken last time – may have collision[冲突]
  - Will cause two miss-predictions in a loop (at start and end of loop)





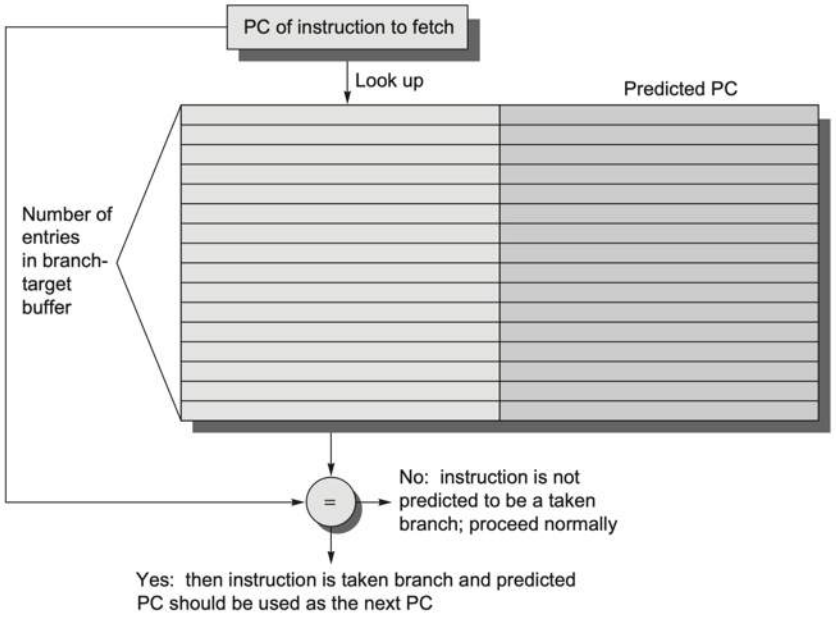
Performance[性能]

- Miss prediction rate for three different predictors



Branch Target Buffers(§3.9)[目标缓冲区]

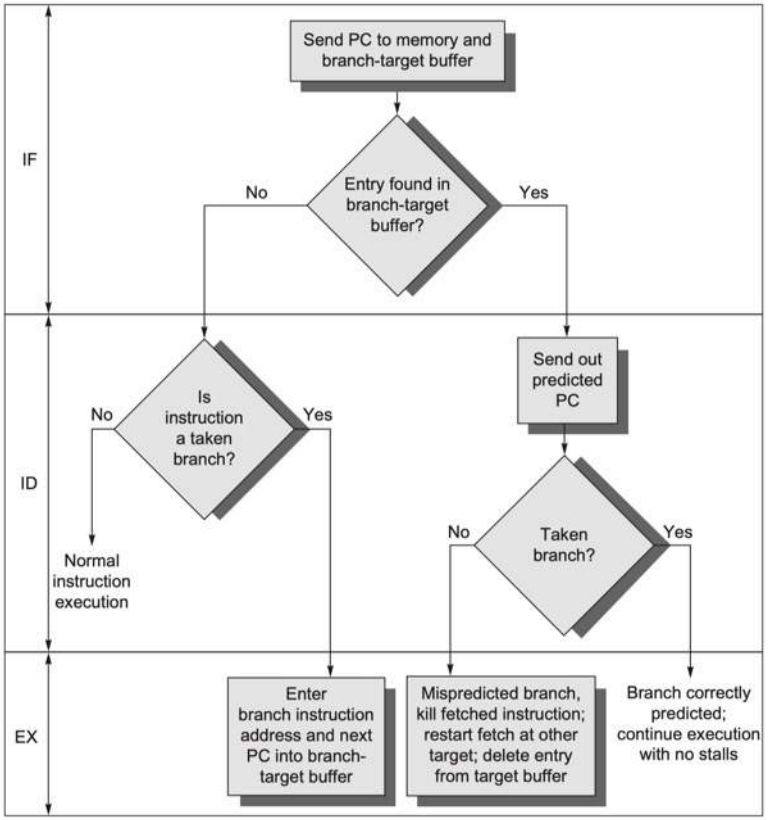
- To increase instruction fetch bandwidth
  - Store the *address* of the branch's target, in addition to the prediction



- Can determine the target address while fetching the branch instruction
  - How do you even know that the instruction is a branch?
  - Can't afford to use wrong branch address due to collision -- why?

Branch Prediction & Pipelining

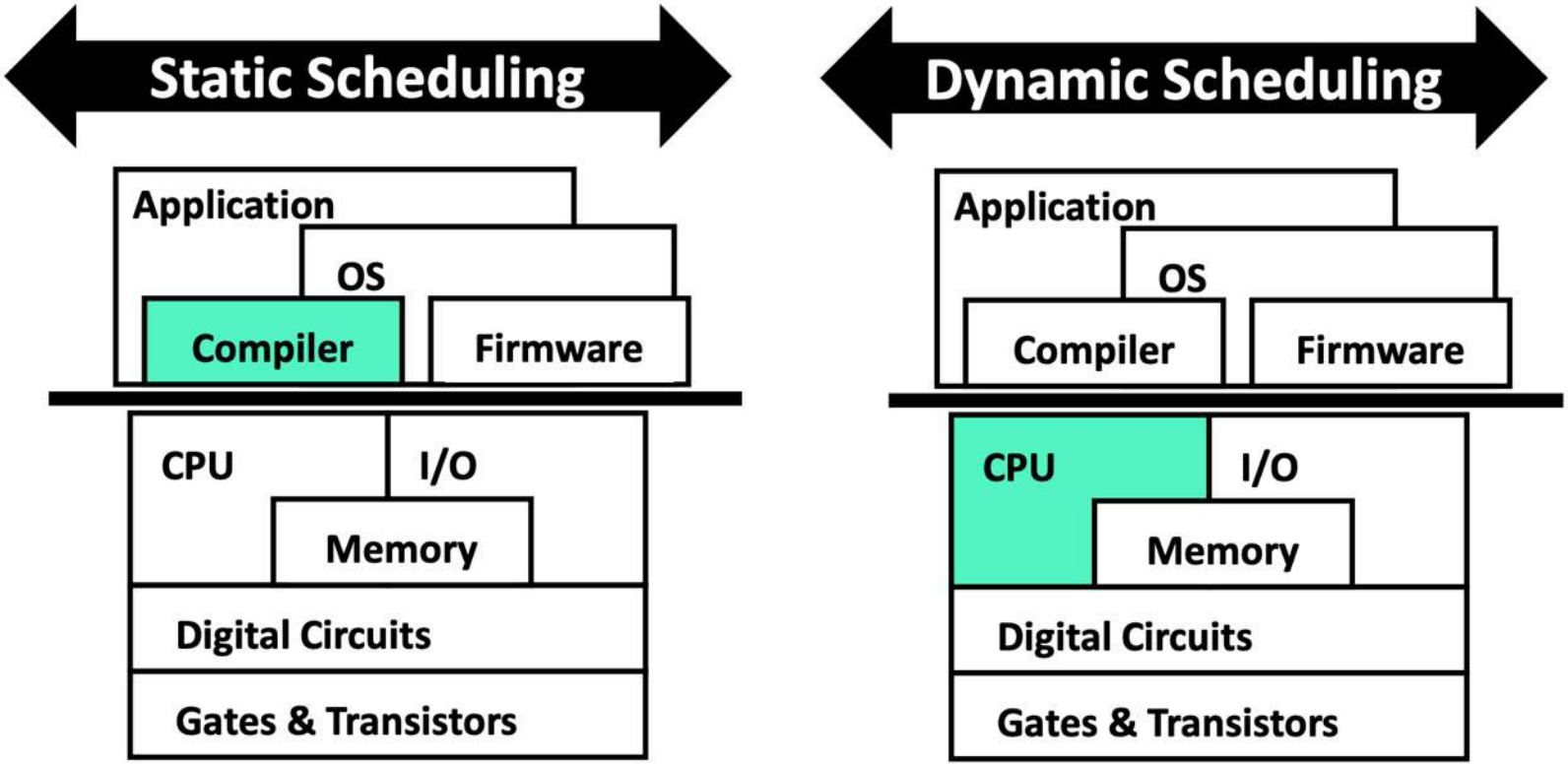
- Assuming that branch condition and target are resolved in *ID* stage



- A similar chart may be drawn if branch condition/target are resolved in *EX*

Instruction Scheduling[指令调度]

- Scheduling: act of finding independent instructions
  - Static: done at compile time by the compiler (sw)
  - Dynamic: done at runtime by the processor (hw)
    - Scoreboard, Tomasulo's algorithm, Reorder Buffer (ROB)



https://acg.cis.upenn.edu/milom/mini-course-March-2013/lectures/08_scheduling.pdf
http://camelab.org/uploads/Main/lecture08-scoreboard.pdf

Compiler Techniques to Expose ILP

- Scheduling[调度]
  - To keep a pipeline full, parallelism among insts must be exploited by finding sequences of unrelated insts that can be overlapped in the pipeline[重叠]
  - To avoid a pipeline stall, the execution of a dependent inst must be separated from the source insts by a distance in clock cycles equal to the pipeline latency of that source inst[分隔]
- A compiler's ability to perform the scheduling depends on
  - Amount of ILP in the program[程序特性]
  - Latencies of the functional units in the pipeline[硬件特性]
- Compiler can increase the amount of availablility of ILP by transforming loops[循环转换]

Loop Dependences(§3.2) [循环依赖]

```
for (i = 999; i >= 0; i = i-1)
    x[i+1] = x[i] + y[i];
```

- [有]There is a loop carried dependence since the statement in an iteration depends on an earlier iteration

```
for (i = 999; i >= 0; i = i-1)
    x[i] = x[i] + s;
```

- [无]There is no loop carried dependence

- The iterations of a loop can be executed in parallel if there is no loop carried dependence

Example: Loop Transformation[循环转换]

```
for (i = 999; i >= 0; i = i-1)
    x[i] = x[i] + s;
```

```
Loop:  fld      f0, 0(x1)      //f0=array element
       fadd.d   f4, f0, f2     //add scalar in f2
       fsd      f4, 0(x1)      //store result
       addi     x1, x1, -8     //decrement pointer
                               //8 bytes (per DW)
       bne      x1, x2, Loop   //branch x1 != x2
```

- Assume the latencies of FP operations
  - 3 cycles if an **FP ALU op** follows and depends on an **FP ALU op**
  - 2 cycles if an **FP store** follows and depends on an **FP ALU op**
  - 1 cycle is an **FP ALU op** follows and depends on an **FP load**
  - 1 cycle if a **branch** follows and depends on on **Integer ALU op**

Basic Scheduling[简单调度]

- Re-order the statements
  - Actual work: *load*, *add* and *store*
  - loop overhead: *addi*, *bne*, two *stalls*

```
                                    cycle
Loop:  fld      f0, 0(x1)           1
       stall                        2
       fadd.d   f4, f0, f2          3
       stall                        4
       stall                        5
       fsd      f4, 0(x1)           6
       addi     x1, x1, -8          7
       stall                        8
       bne      x1, x2, loop        9
```

9 clock cycles per iteration

```
                                    cycle
Loop:  fld      f0, 0(x1)           1
       addi     x1, x1, -8          2
       fadd.d   f4, f0, f2          3
       stall                        4
       stall                        5
       fsd      f4, 8(x1)           6
       bne      x1, x2, loop        7
```

7 clock cycles per iteration

Loop Unrolling[循环展开]

- Simply replicates the loop body multiple times, adjusting the loop termination code[复制->调整]
  - Increases the number of insts relative to the branch and overhead insts[增加有效指令数]
  - Eliminates branches, thus allowing insts from different iterations to be scheduled together[消除分支, 共同调度]

```
Loop: fld      f0, 0(x1)
      fadd.d   f4, f0, f2
      fsd      f4, 0(x1)
      fld      f6, -8(x1)
      fadd.d   f8, f6, f2
      fsd      f8, -8(x1)
      fld      f0, -16(x1)
      fadd.d   f12, f0, f2
      fsd      f12, -16(x1)
      fld      f14, -24(x1)
      fadd.d   f16, f14, f2
      fsd      f16, -24(x1)
      addi     x1, x1, -32
      bne      x1, x2, loop
```

```
Loop: fld      f0, 0(x1)
      fld      f6, -8(x1)
      fld      f0, -16(x1)
      fld      f14, -24(x1)
      fadd.d   f4, f0, f2
      fadd.d   f8, f6, f2
      fadd.d   f12, f0, f2
      fadd.d   f16, f14, f2
      fsd      f4, 0(x1)
      fsd      f8, -8(x1)
      fsd      f12, -16(x1)
      fsd      f16, -24(x1)
      addi     x1, x1, -32
      bne      x1, x2, loop
```

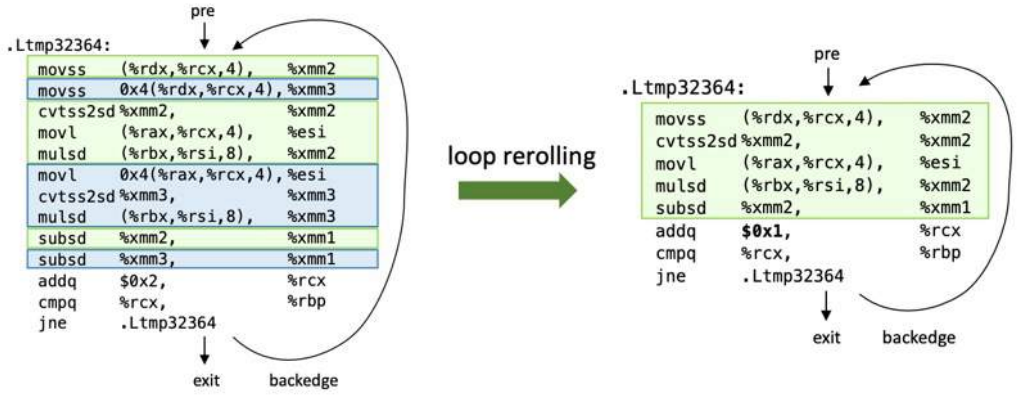
A total of 14 clock cycles
(3.5 cycles per iter)

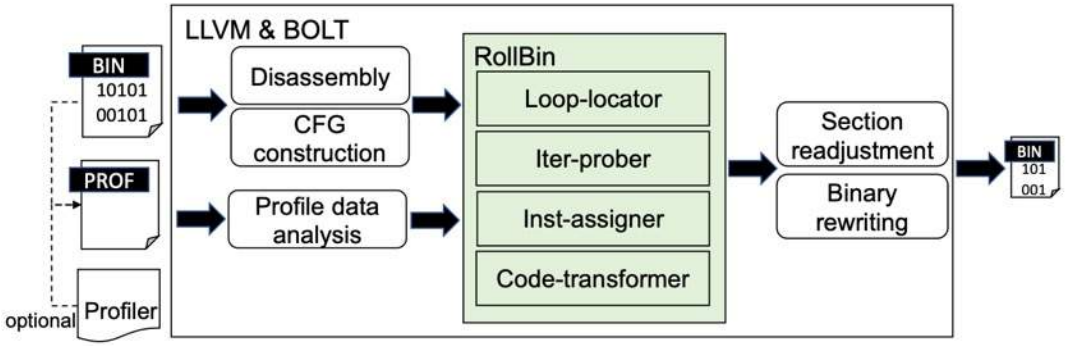
Unrolling Limitations[限制]

- The gains from loop unrolling are limited by
  - A decrease in the amount of overhead amortized with each unroll
    - Unrolled 4 times → 8 times: ½ cycle/iter → ¼ cycle/iter
  - Growth in code size caused by unrolling
    - May increase in the inst cache miss rate
    - May bring register pressure (more live values)
  - Compiler limitations
    - Sophisticated transformations increases the compiler complexity

```
Loop: fld     f0, 0(x1)
      fld     f6, -8(x1)
      fld     f0, -16(x1)
      fld     f14, -24(x1)
      fadd.d  f4, f0, f2
      fadd.d  f8, f6, f2
      fadd.d  f12, f0, f2
      fadd.d  f16, f14, f2
      fsd     f4, 0(x1)
      fsd     f8, -8(x1)
      fsd     f12, -16(x1)
      fsd     f16, -24(x1)
      addi    x1, x1, -32
      bne     x1, x2, loop
```

Paper: Loop Rerolling





RollBin: Reducing Code-Size via Loop Rerolling at Binary Level

Tianao Ge
Sun Yat-Sen University
China
getao3@mail2.sysu.edu.cn

Zewei Mo
Sun Yat-Sen University
China
mozw5@mail2.sysu.edu.cn

Kan Wu
Sun Yat-Sen University
China
wukan3@mail2.sysu.edu.cn

Xianwei Zhang
Sun Yat-Sen University
China
zhangxw79@mail.sysu.edu.cn

Yutong Lu
Sun Yat-Sen University
China
luyutong@mail.sysu.edu.cn

Abstract

Code size is an increasing concern on resource constrained systems, ranging from embedded devices to cloud servers. To address the issue, lowering memory occupancy has become a priority in developing and deploying applications, and accordingly compiler-based optimizations have been proposed to reduce program footprint. However, prior arts are generally dealing with source codes or intermediate representations, and thus are very limited in scope in real scenarios where only binary files are commonly provided. To fill the gap, this paper presents a novel code-size optimization `RollBin` to reroll loops at binary level. `RollBin` first locates the unrolled loops in binary files, and then probes to decide the unrolling factor by identifying regular memory address patterns. To reconstruct the iterations, we propose a customized data dependency analysis that tackles the challenges brought by shuffled instructions and loop-carry dependencies. Next, the recognized iterations are rolled up through instruction removal and update, which are generally reverting the normal unrolling procedure. The evaluations on standard SPEC2006/2017 and MiBench demonstrate that `RollBin` effectively shrinks code size by 1.7% and 2.2% on average (up to 7.8%), which respectively outperforms the state-of-the-arts by 31% and 38%. In addition, the use cases of representative realistic applications manifest that `RollBin` can be applicable in practices.

CCS Concepts: • **Software and its engineering → Compilers**.

LCTES '22, June 14, 2022, San Diego, CA, USA

ACM ISBN 978-1-4503-9266-2/22/06…$15.00
https://doi.org/10.1145/3519941.3535072

Keywords: Code-Size Reduction, Loop Rerolling, Binary Optimization

ACM Reference Format:
Tianao Ge, Zewei Mo, Kan Wu, Xianwei Zhang, and Yutong Lu. 2022. RollBin: Reducing Code-Size via Loop Rerolling at Binary Level. In *Proceedings of the 23rd ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES '22), June 14, 2022, San Diego, CA, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3519941.3535072

1 Introduction

In the past decades, computer programs have been continuously gaining new features and growing in size and complexity, which together drive the non-stop need for higher computing horsepower and larger memory capacity [2, 14]. As such, for smoothly executing programs and efficiently utilizing the precious resources, especially the memory space and bandwidth, reducing program footprint becomes essential on all computing platforms spanning from servers to embedded systems. For embedded and Internet-of-Things (IoT) devices, code volume is an overwhelming concern, as it directly impacts the chip area and cost, and further influences the overall performance and power [29, 42]. On larger machines, such as desktops, servers and supercomputers, whereas memory capacity is typically much less limited, code size is nonetheless critical for instruction cache (I-cache) performance [43]. Recently, there has been an increasing trend toward unifying libraries, tools, and frameworks to support cross-architecture executions [6, 20], including servers and edge devices, which thus further emphasizes the compacted code across platforms. TensorFlow Lite [40] and BLASFEO [13] are such representative examples actively expanding the machine learning and high-performance computing territories from powerful servers to constrained devices.

Classical techniques, including variable-length instruction encoding [16, 30], code compression [25, 44], and ISA modification [45], are designed to reduce the size of code. Program footprint can also be lessened by compiler-based similar code merging [34] and dead-code eliminating [21, 26].

Dynamically Scheduled Pipelines (§3.4)

- Key idea: allow instructions behind stall to proceed

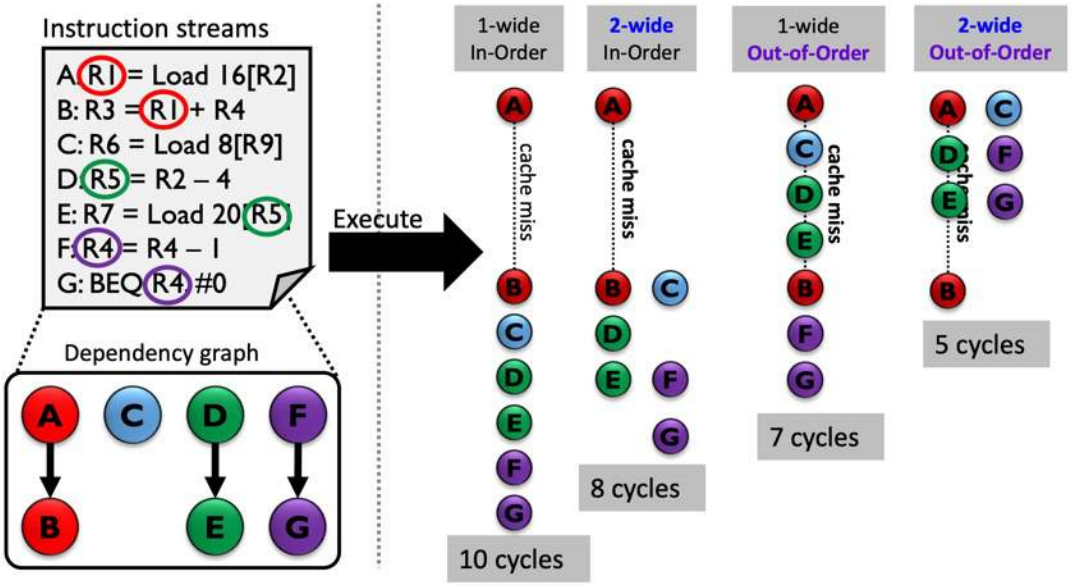
```
fdiv   F0,   F2,   F4
fadd   F10,  F0,  F8
fsub   F12,  F8,   F14
```

RAW -> Stall

No dependency

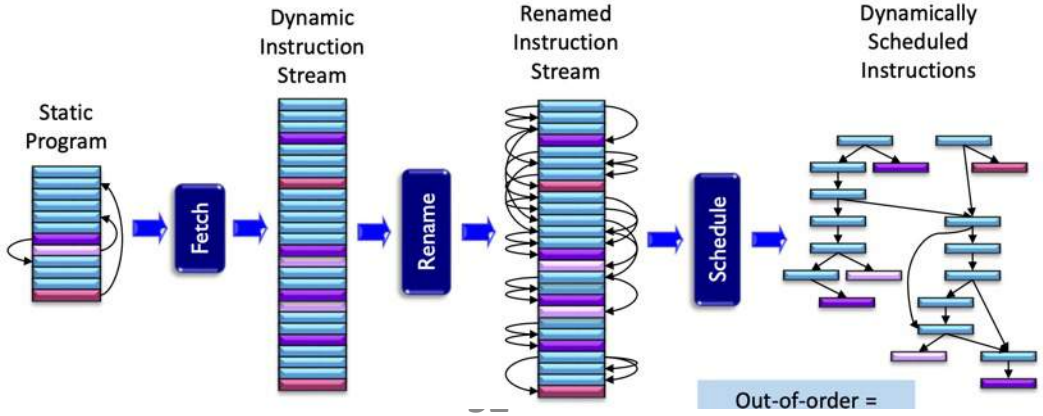
  - Enables out-of-order (OoO, O3) execution
    - Can lead to O3 completion

- Hardware rearranges instruction stream to reduce stalls



Out-of-order[乱序执行]

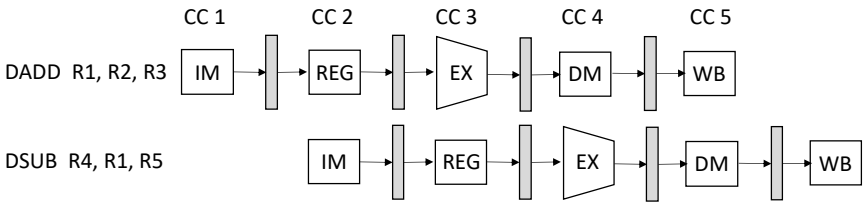
- How can O3 achieve performance benefits?
  - Hardware rearranges instruction stream to reduce stalls
- Any problems of O3?
  - Hazards! Especially for register dependencies
- How does the O3 work?
  - Step1: fetch many instructions into instruction window
  - Step2: rename regs. to avoid false deps. (WAW and WAR)
  - Step3: execute instructions as soon as dependencies (registers and memory) are known



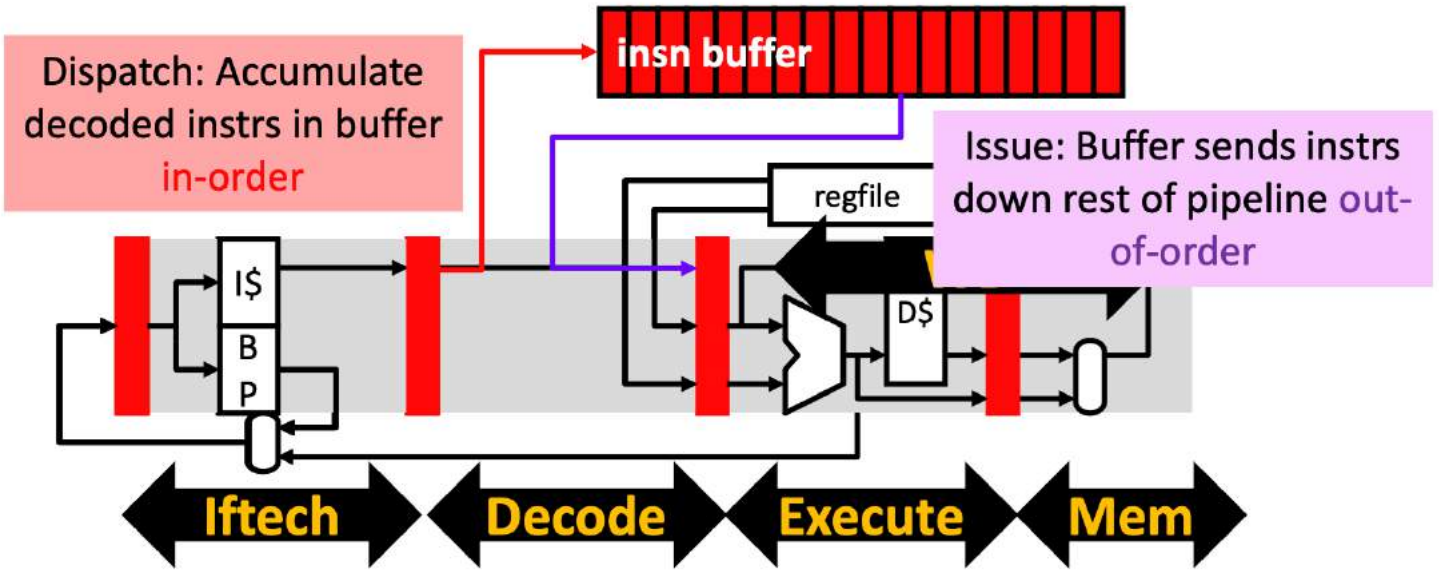
http://camelab.org/uploads/Main/lecture08-scoreboard.pdf

O3 Pipeline

- Split the ID stage into
  - Dispatch
  - Issue



- Instructions wait in a queue and may move to the EX stage (issued) out of order
  - A new kind of structural hazard : Instruction buffer is full

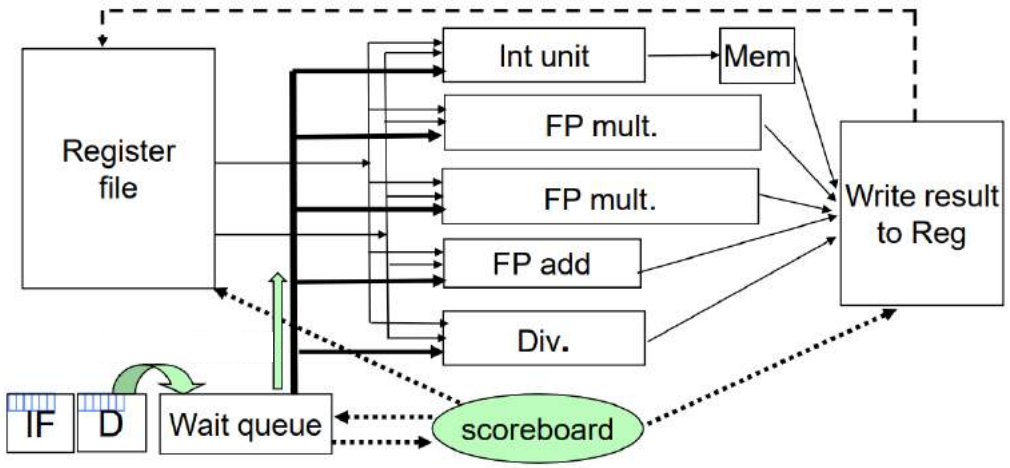


Scoreboard[记分板]

- Using Scoreboard (§C.7):
  - Dates to the first supercomputer, the CDC 6600 in 1963
- To track the flow of the instrs, register, and function units
  - Check which Datapath components are using / can be used
  - Find out which instruction could be executed without hazards

http://camelab.org/uploads/Main/lecture08-scoreboard.pdf
https://sportspectator.com/fancentral/football/scoreboard.html

A Scoreboard Architecture



- The scoreboard is responsible for instruction issue and execution, including hazard detection. It is also controlling the writing of the results
- The "scoreboard" consists of 3 tables to keep track of execution progress and the associated intelligence to determine when to dispatch instructions
- One entry (buffer) in the "wait queue" is associated with each functional unit

Scoreboard Information

- Three main components/tables
  - Instruction status
    - Which step the instruction is in
  - Functional unit status
    - Which state the FU is in
  - Register result status
    - Which FU will write registers

Scoreboard

Insn Status

Inst	dst	src1	src2	D	S	X	W
LD	F6	34+	R2				
LD	F2	45+	R3				
MULTD	F0	F2	F4				
SUBD	F8	F6	F2				
DIVD	F10	F0	F6				
ADDD	F6	F8	F2				

FU Status

FU	B	Op	dst	src1	src2	Q1	Q2	R1	R2
Int									
Mult1									
Mult2									
Add									
Div									

Reg Status

	FU
F0	
F2	
F4	
F6	
F8	
F10	
...	

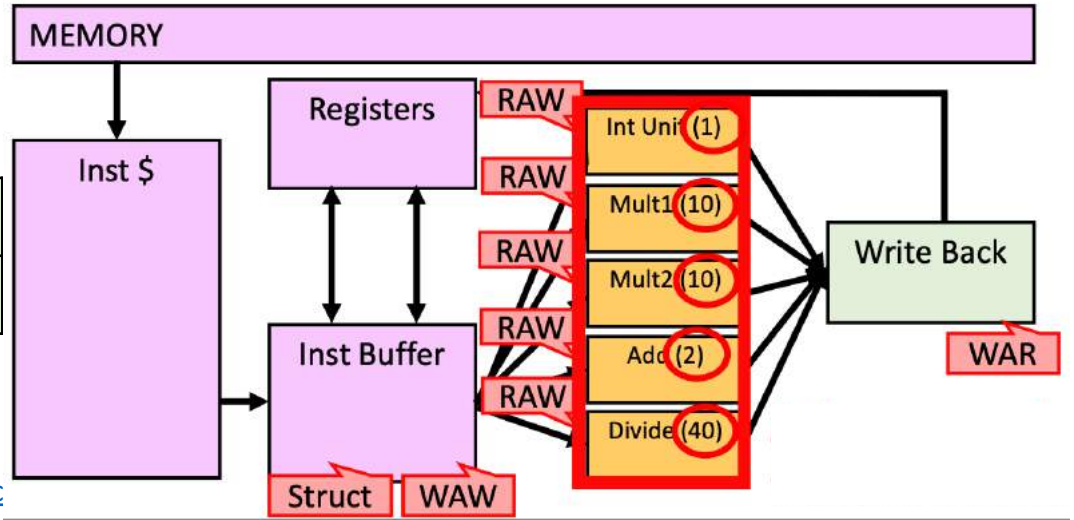
Status Tables

- **Instruction** status[指令状态]: which of 4 steps the inst is in
  - D: Issue
  - S: Read operands
  - X: Execute stage completion
  - W: Write result to registers
- **Functional Unit** (FU) Status[运算单元状态]: indicates the state of the FU
  - 9 fields for each FU
    - B: indicates whether the unit is busy or not
    - Op: operation to perform in the unit (e.g., + or -)
    - dst/Fi: destination register
    - src1,src2/Fj, Fk: source-register numbers
    - Qj, Qk: functional units producing source registers src1, src2
    - Rj, Rk: flags being set when src1/src2 is ready
- **Register Result** Status[寄存器结果状态]: indicates which FU will write each register, if one exits
  - Blank when no pending instructions will write that register

Scoreboard Workflow

- **Issue**: decode insts and check for structural, WAW hazards
  - Wait conditions: (1) the required FU is free; (2) no other inst writes to the same register dst. (to avoid WAW)
- **Read operands**: only if no RAW hazard
  - Wait conditions: all source operands are ready
- **Execution**: operate on operands
  - When execution terminates, notify the scoreboard
- **Write result**: write reg and update scb
  - Wait condition: no other inst/FU is going to read the register dst. of the inst

ID		EX	WB
Issue	Rd Ops	Execution	Write



Scoreboard Example

- when “fld F6, 34(R2)” is writing

Instruction status

Instruction		Issue	Read op.	Exec. Completed	Write result	
fld	F6, 34(R2)	X	X	X	X	done
fld	F2, 45(R3)	X	X	X		
fmul.d	F0, F2, F4	X				
fsub,d	F8, F6, F2	X				
fdiv.d	F10, F0, F12	X				
fadd.d	F6, F8, F2					Not in

Func. unit status

Unit	Busy	Op	Fi	Fj	Fk	Qj	Qk	Rj	Rk
Integer	Yes	Load	F2	R3				Yes	
Mult1	Yes	Mult	F0	F2	F4	Int.		No	Yes
Mult2	No								
Add	Yes	Sub	F8	F6	F2		Int.	Yes	No
divide	Yes	Div	F10	F0	F12	Mult1		No	Yes

Register status

	F0	F2	F4	F6	F8	F10	F12	…	F30
Func. U	Mult1	Int.			Add	Div			

Scoreboard Example (cont.)

- when “fld F2, 45(R3)” is writing

Instruction status

Instruction		Issue	Read op.	Exec. Completed	Write result	
fld	F6, 34(R2)	X	X	X	X	done
fld	F2, 45(R3)	X	X	X	X	
fmul.d	F0, F2, F4	X				
fsub,d	F8, F6, F2	X				
fdiv.d	F10, F0, F12	X				
fadd.d	F6, F8, F2					Not in

Func. unit status

Unit	Busy	Op	Fi	Fj	Fk	Qj	Qk	Rj	Rk
Integer	Yes	Load	F2	R3				Yes	
Mult1	Yes	Mult	F0	F2	F4			Yes	Yes
Mult2	No								
Add	Yes	Sub	F8	F6	F2			Yes	Yes
divide	Yes	Div	F10	F0	F12	Mult1		No	Yes

Register status

	F0	F2	F4	F6	F8	F10	F12	…	F30
Func. U	Mult1				Add	Div			

https://people.cs.pitt.edu/~melhem/courses/2410p/ch3-3.pdf

Scoreboard Example (cont.)

- 3 cycles after “fsub.d” finished writing

Instruction status

Instruction		Issue	Read op.	Exec. Completed	Write result
fld	F6, 34(R2)	X	X	X	X
fld	F2, 45(R3)	X	X	X	X
fmul.d	F0, F2, F4	X	X	X	
fsub,d	F8, F6, F2	X	X	X	X
fdiv.d	F10, F0, F12	X			
fadd.d	F6, F8, F2	X	X	X	

Func. unit status

Unit	Busy	Op	Fi	Fj	Fk	Qj	Qk	Rj	Rk
Integer	No								
Mult1	Yes	Mult	F0	F2	F4			Yes	Yes
Mult2	No								
Add	Yes	add	F4	F8	F2			Yes	Yes
divide	Yes	Div	F10	F0	F12	Mult1		No	Yes

Register status

	F0	F2	F4	F6	F8	F10	F12	…	F30
FU	Mult1		Add			Div			

https://people.cs.pitt.edu/~melhem/courses/2410p/ch3-3.pdf

Summary of Scoreboard

- Basic idea
  - Use scoreboard to track data dep. through register
- Main points of design
  - Instructions are sent to FU unit if there is no outstanding name dependence
  - RAW data dependence is tracked and enforced by scoreboard
  - Register values are passed through the register file; a child instruction starts execution after the last parent finishes execution
  - Pipeline stalls if any name dependence (WAR or WAW) is detected

http://users.utcluj.ro/~sebestyen/_Word_docs/Cursuri/SSC_course_5_Scoreboard_ex.pdf

Summary of Scoreboard

- Basic idea
  - Use scoreboard to track data dep. through register
- Main points of design
  - Instructions are sent to FU unit if there is no outstanding name dependence
  - RAW data dependence is tracked and enforced by scoreboard

    How? Just stall the insts until the RAW hazard can be addressed.
  - Register values are passed through the register file; a child instruction starts execution after the last parent finishes execution
  - Pipeline stalls if any name dependence (WAR or WAW) is detected

Summary of Scoreboard

- Basic idea
  - Use scoreboard to track data dep. through register
- Main points of design
  - Instructions are sent to FU unit if there is no outstanding name dependence
  - RAW data dependence is tracked and enforced by scoreboard

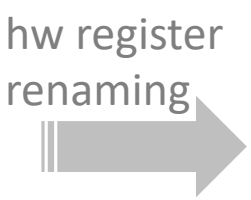
    How? Just stall the insts until the RAW hazard can be addressed.
  - Register values are passed through the register file; a child instruction starts execution after the last parent finishes execution
  - Pipeline stalls if any name dependence (WAR or WAW) is detected

    How? Just recognize the false dependencies as a hazard and stall.

Tomasulo Algorithm

- Key idea: remove dependencies with..
  - 1) HW register renaming
    - What compiler cannot do
  - 2) Data forwarding

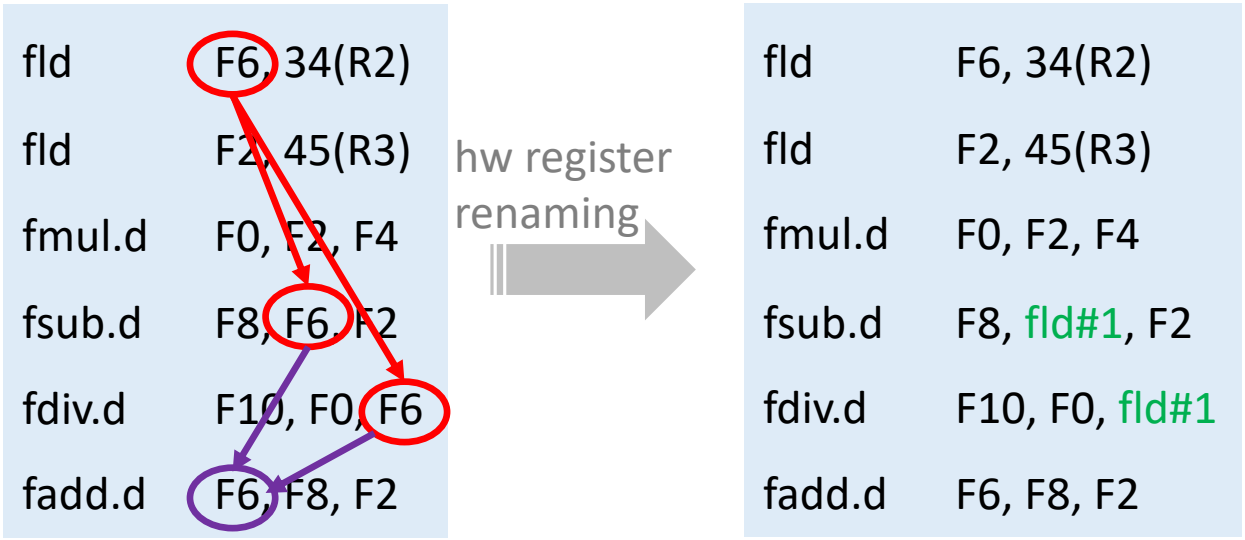
fld	F6, 34(R2)
fld	F2, 45(R3)
fmul.d	F0, F2, F4
fsub.d	F8, F6, F2
fdiv.d	F10, F0, F6
fadd.d	F6, F8, F2



fld	F6, 34(R2)
fld	F2, 45(R3)
fmul.d	F0, F2, F4
fsub.d	F8, fld#1, F2
fdiv.d	F10, F0, fld#1
fadd.d	F6, F8, F2

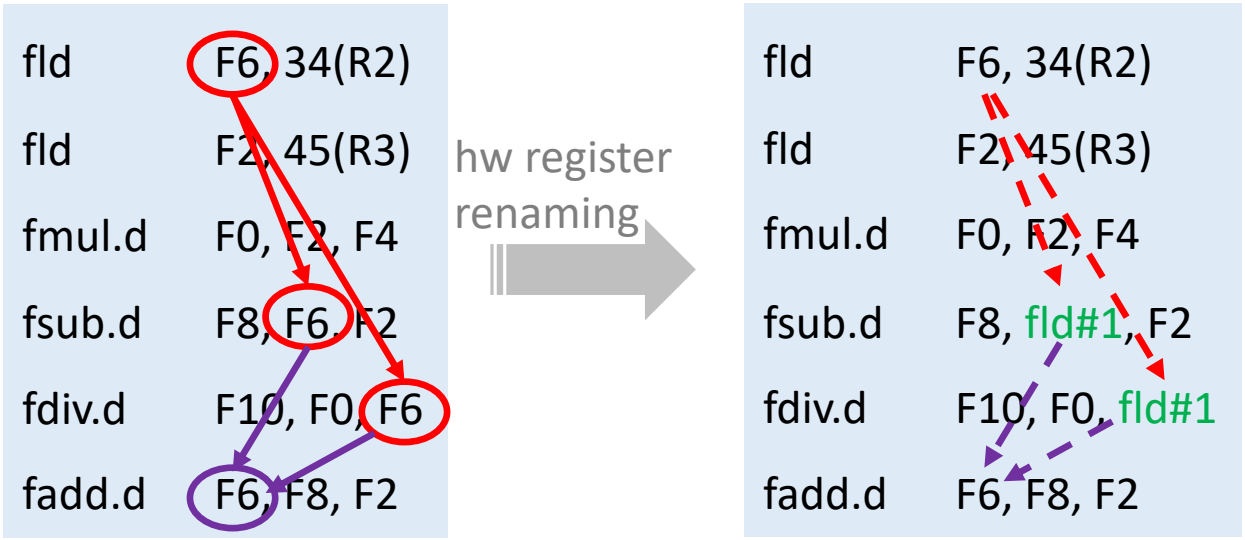
Tomasulo Algorithm

- Key idea: remove dependencies with..
  - 1) HW register renaming
    - What compiler cannot do
  - 2) Data forwarding



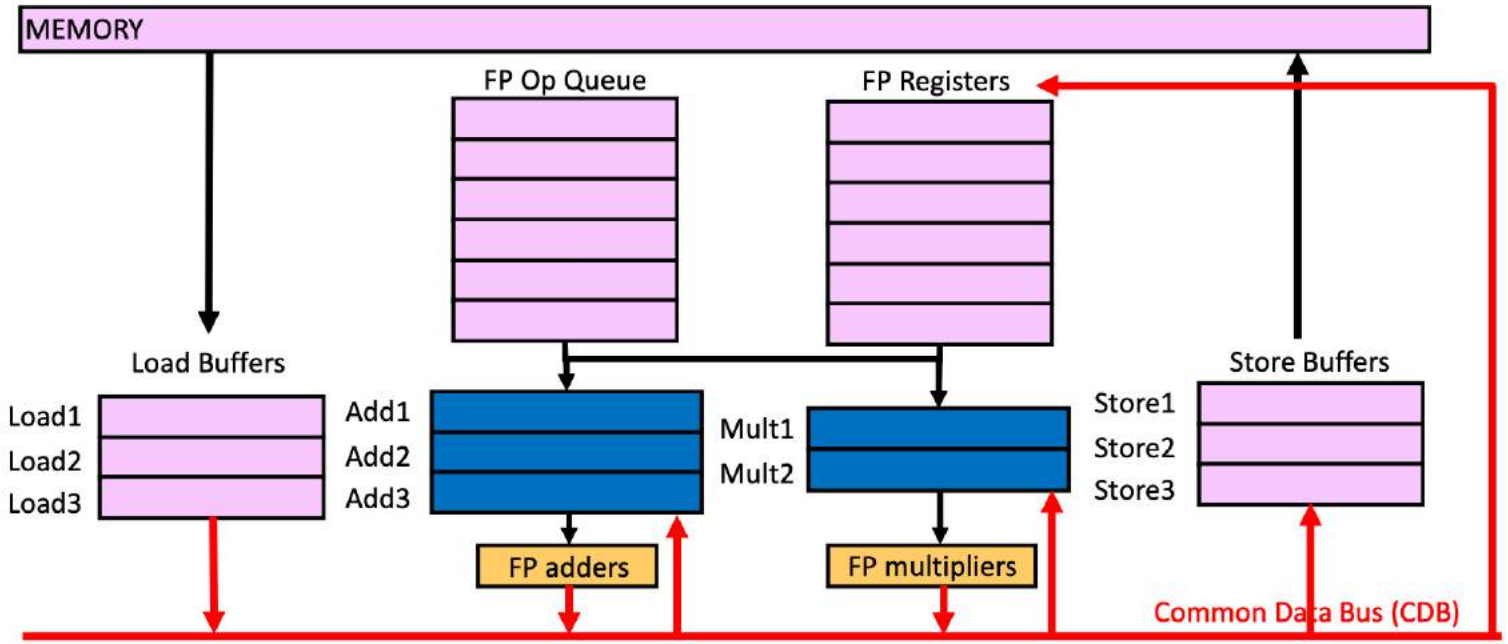
Tomasulo Algorithm

- Key idea: remove dependencies with..
  - 1) HW register renaming
    - What compiler cannot do
  - 2) Data forwarding



Tomasulo Organization

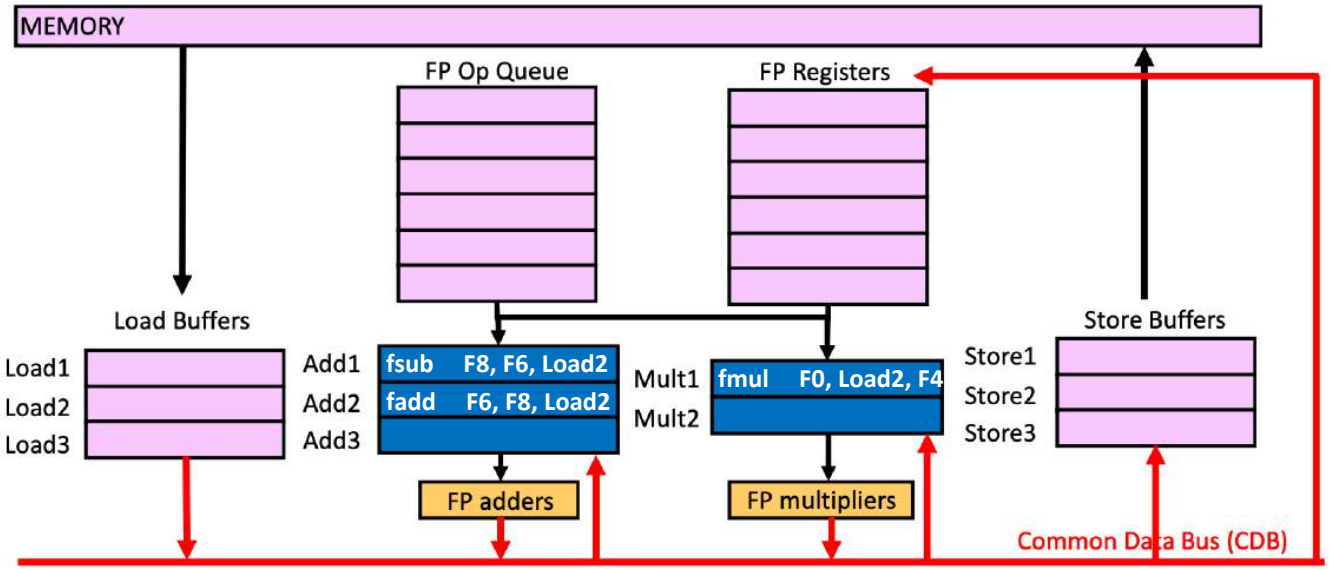
- Control & buffers are distributed with Function Units (FU)
  - FU buffers called “Reservation Stations (RS)”; have pending ops
  - Registers in instructions replaced by values or pointers to RS
- Load and Store treated as FUs with RSs as well
- Results to FU from RS, not through registers, over Common Data Bus (CDB) that broadcasts results to all FUs



Three Stages of Tomasulo

- Stage-1: **Issue**
- Get an instruction from FP Op Queue
  - If the reservation station is free (no structural hazard), the control issues such instruction and sends corresponding operands (renames registers)
    - Register are renamed in this step, eliminating WAR and WAW hazards

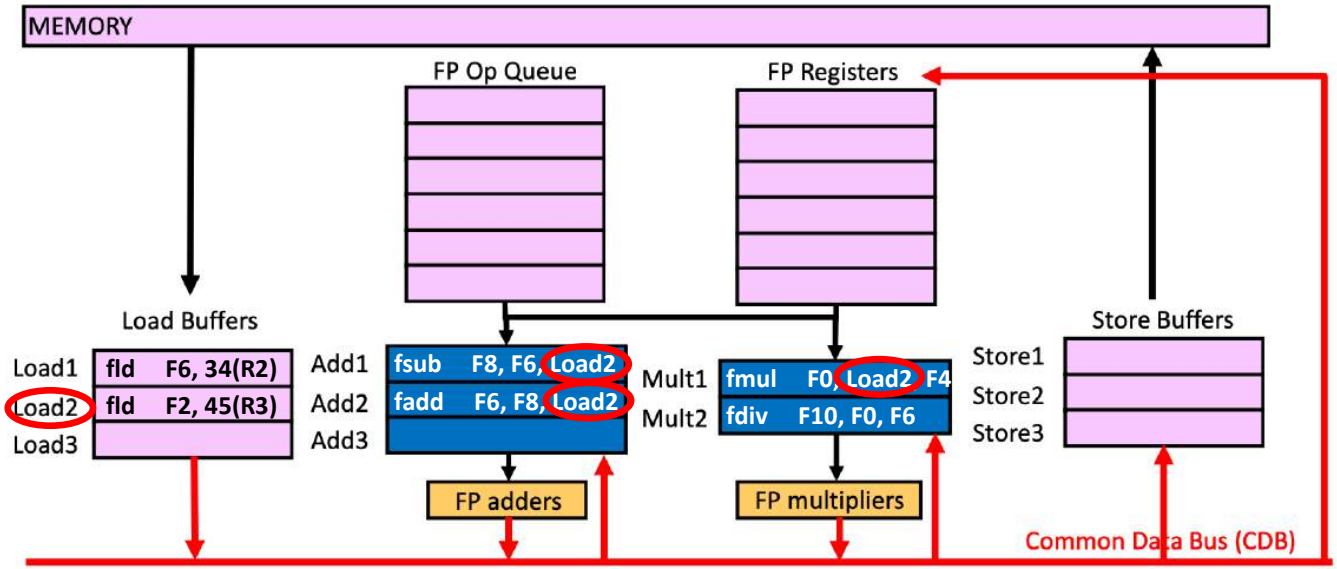
fld	F6, 34(R2)
fld	F2, 45(R3)
fmul.d	F0, F2, F4
fsub.d	F8, F6, F2
fdiv.d	F10, F0, F6
fadd.d	F6, F8, F2



Three Stages of Tomasulo (cont.)

- Stage-2: **Execute**
- Operate on operands (EX)
  - When both operands are ready, it executes; otherwise, it checks up the CDB for results
    - Instructions are delayed here until all of their operands are available, eliminating RAW hazards

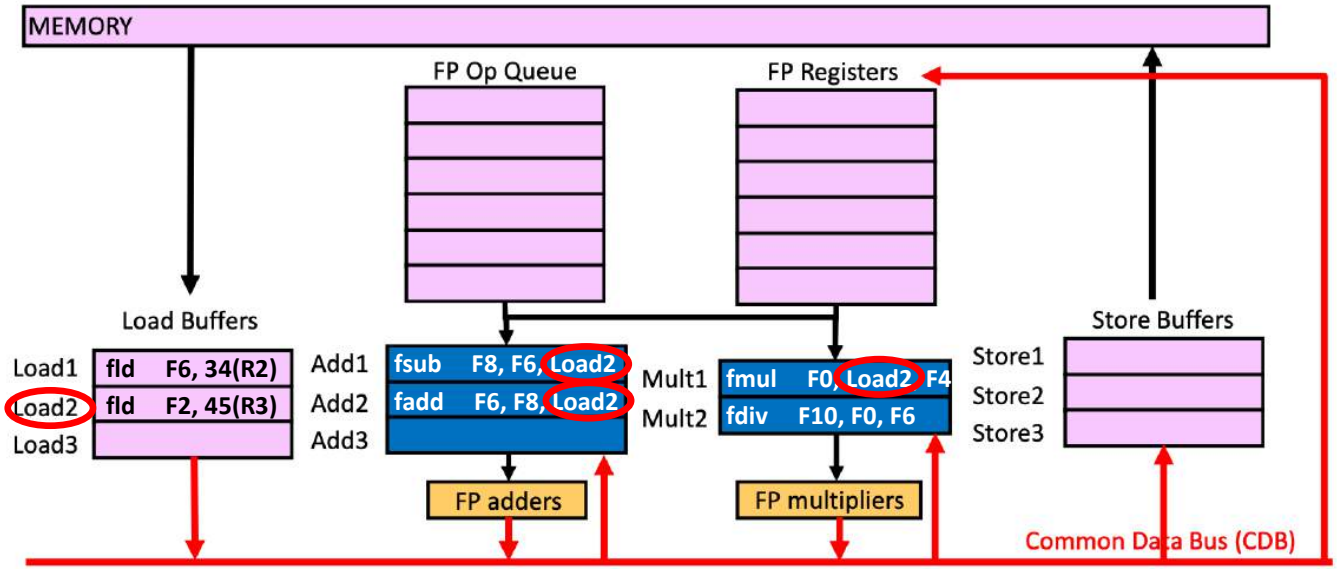
fld	F6, 34(R2)
fld	F2, 45(R3)
fmul.d	F0, F2, F4
fsub.d	F8, F6, F2
fdiv.d	F10, F0, F6
fadd.d	F6, F8, F2



Three Stages of Tomasulo (cont.)

- Stage-3: **Write result**
- Finish execution:
  - ALU operations results are written back to registers and store operations are written back to memory
    - If the result is available, write it on the CDB and from there into the registers and any reservation stations waiting for this result

fld	F6, 34(R2)
fld	F2, 45(R3)
fmul.d	F0, F2, F4
fsub.d	F8, F6, F2
fdiv.d	F10, F0, F6
fadd.d	F6, F8, F2



Simple Tomasulo Data Structures

- Three main components
  - Instruction status
  - Reservation stations (Load buffer & FU buffer)
    - Scheduling: waiting operands
    - Register renaming: remove false dep.
  - Register result status

Scoreboard

Insn Status							
Inst	dst	src1	src2	D	S	X	W
LD	F6	34+	R2				
LD	F2	45+	R3				
MULTD	F0	F2	F4				
SUBD	F8	F6	F2				
DIVD	F10	F0	F6				
ADDD	F6	F8	F2				

FU Status									
FU	B	Op	dst	src1	src2	Q1	Q2	R1	R2
Int									
Mult1									
Mult2									
Add									
Div									

Reg Status	
	FU
F0	
F2	
F4	
F6	
F8	
F10	
...	

Tomasulo

Insn Status						
Inst	dst	src1	src2	D	X	W
LD	F6	34+	R2			
LD	F2	45+	R3			
MULTD	F0	F2	F4			
SUBD	F8	F6	F2			
DIVD	F10	F0	F6			
ADDD	F6	F8	F2			

RS (Load buffer)		
	B	Addr
LD1		
LD2		
LD3		

Reservation Stations (FU buffer)						
FU	B	Op	V1	V2	Q1	Q2
Add1						
Add2						
Add3						
Mult1						
Mult2						

Reg Status	
	FU
F0	
F2	
F4	
F6	
F8	
F10	
...	

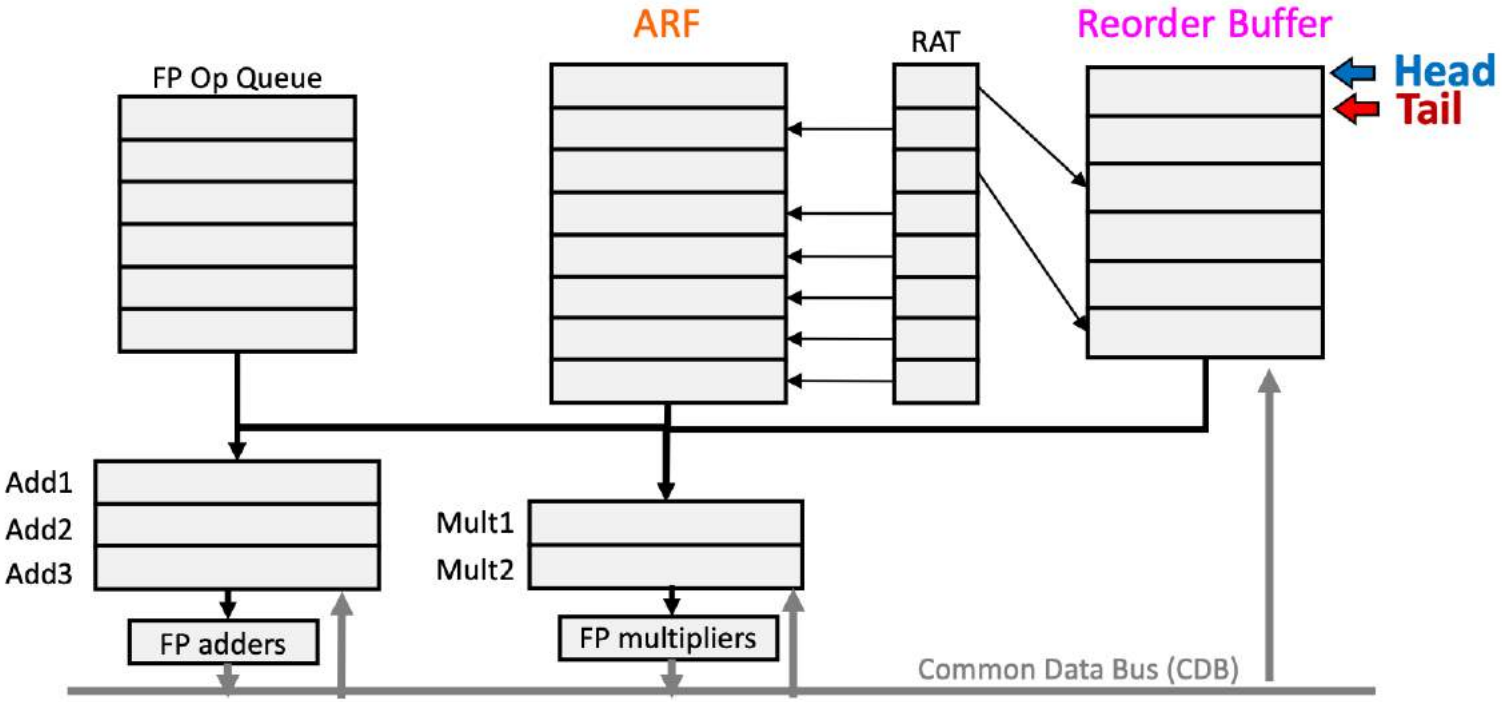
http://camelab.org/uploads/Main/lecture09-Tomasulo.pdf

Reorder Buffer[重排序缓存]

- In the Tomasulo architecture, instructions complete in an *out-of-order*
  - Exceptions are non-trivial to handle
  - Branch misprediction is also difficult to recover from
- **Reorder Buffer** (ROB) enables to finish instructions in the program order
  - And, allows to free RS earlier
  - ROB holds the result of inst between completion and commit
- Key idea of ROB: execute the insts in out of program order, but make outside world can "believe" it's in-order
  - Solution: Re-Order Buffer+ Architected Register File
    - ROB: keep the temporal results (executed in out-of-order)
    - ARF: keep the final results (illusion of in-order execution)

Tomasulo w/ ROB Organization

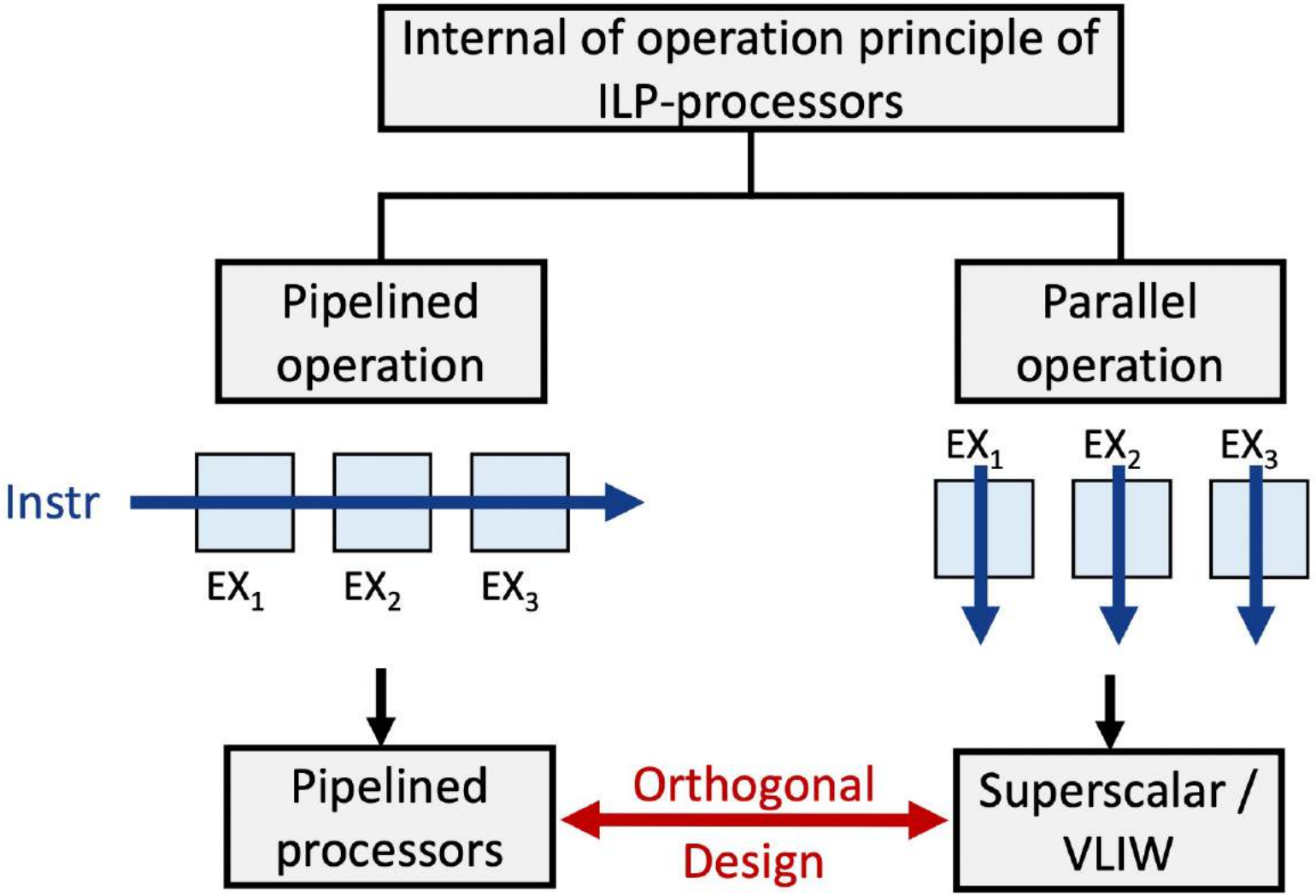
- Re-Order buffer is based on Tomasulo
- Just renamed FP register to ARF (Architected Register File)
- Add Re-Order buffer for out-of-order results
  - Buffer is managed with two pointers (head & tail)
- RAT (Register Alias Table) keeps the register renaming info



Reorder Buffer Procedure[过程]

- Issue
  - Allocate reservation station(RS) and Reorder Buffer(ROB), read available operands
- Execute
  - Begin execution when operand values are available
- Write Result
  - Write result and ROB tag on CDB
- Commit
  - When ROB reaches head, update register
  - When a mispredicted branch reaches head of ROB, discard all entries

Another ILP


Internal of operation principle of ILP-processors
Pipelined operation
Parallel operation
Instr
EX1
EX2
EX3
EX1
EX2
EX3
Pipelined processors
Orthogonal Design
Superscalar / VLIW

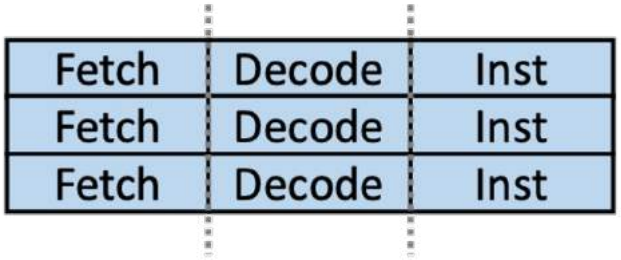
Multiple Issue[多发射]

- To achieve CPI < 1, need to complete multiple instructions per clock
- Solutions:
  - Statically scheduled superscalar processors
  - VLIW (very long instruction word) processors
  - Dynamically scheduled superscalar processors

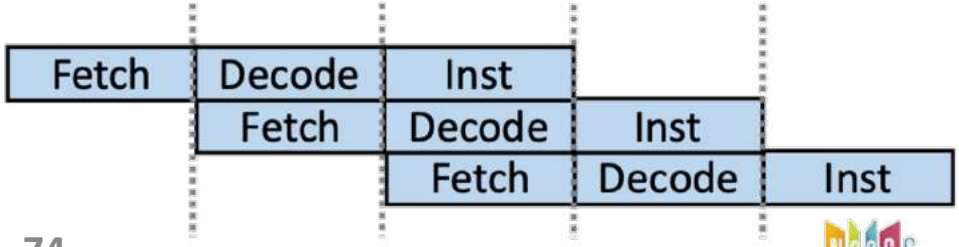
Common name	Issue structure	Hazard detection	Scheduling	Distinguishing characteristic	Examples
Superscalar (static)	Dynamic	Hardware	Static	In-order execution	Mostly in the embedded space: MIPS and ARM, including the Cortex-A53
Superscalar (dynamic)	Dynamic	Hardware	Dynamic	Some out-of-order execution, but no speculation	None at the present
Superscalar (speculative)	Dynamic	Hardware	Dynamic with speculation	Out-of-order execution with speculation	Intel Core i3, i5, i7; AMD Phenom; IBM Power 7
VLIW/LIW	Static	Primarily software	Static	All hazards determined and indicated by compiler (often implicitly)	Most examples are in signal processing, such as the TI C6x
EPIC	Primarily static	Primarily software	Mostly static	All hazards determined and indicated explicitly by the compiler	Itanium

Superscalar[超标量]

- Superscalar architectures allow several instructions to be issued and completed per clock cycle
- A superscalar architecture consists of a number of pipelines that are working in parallel (N-way Superscalar)
  - Can issue up to N instructions per cycle
- Superscalarity is Important
  - Ideal case of N-way Super-scalar
    - All instructions were independent
    - Speedup is “N” (Superscalarity)



  - What if all instructions are dependent?
    - No speed up, super-scalar brings nothing
    - (Just similar to pipelining)



http://camelab.org/uploads/Main/lecture06-istruction-paralllel-processing.pdf