中山大學
SUN YAT-SEN UNIVERSITY

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

Computer Architecture

计 算 机 体 系 结 构

第6讲：ISA & ILP（4）

张献伟

xianweiz.github.io

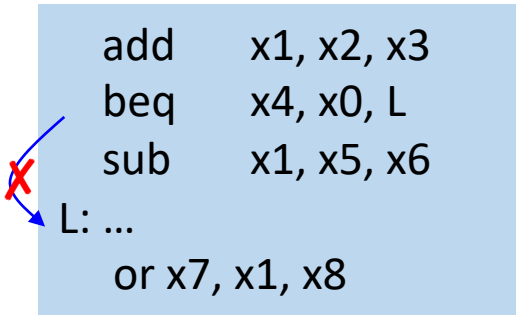
DCS3013, 10/19/2022

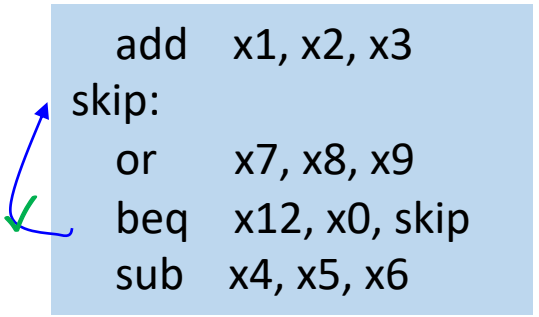
Techniques to Improve ILP

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

Branch Prediction(§3.3)[分支预测]

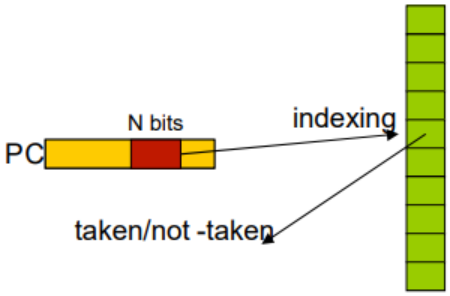
- Branches hurt pipeline performance
  - Branch hazards and stalls
- Static branch prediction[静态分支预测]
  - The default is to assume that branches are not taken
  - May have a design which predicts that branches are taken
- Reasonable to assume that[假设]
  - Forward branches are often not taken
  - Backward 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

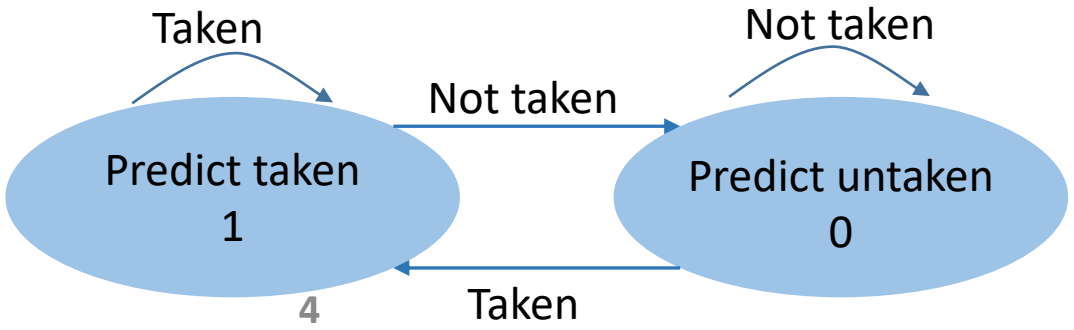




Dynamic Branch Prediction(§C2.7)[动态]

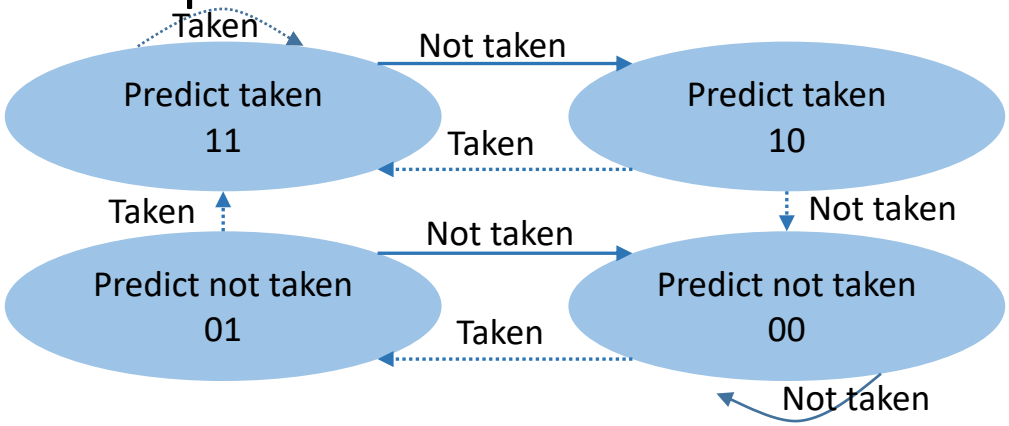
- Performance depends on the accuracy of prediction and the cost of miss-prediction[性能影响]
- The simplest branch prediction scheme: **Branch Prediction Buffer**[分支预测缓存]
  - 1-bit table (cache) indexed by some bits of the address of the branch instructions (can be accessed in decode stage) -> hashing[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)





Two-bit Branch Predictors

- Change your prediction only if miss-predict twice[稳定性]
  - A branch that strongly favors taken or not taken (many branches do), will be miss-predicted less often than with a 1-bit predictor



- In general, *n*-bit predictors are called **Local Predictors**[局部预测器]
  - Use a saturated counter (++ on correct prediction, -- on wrong prediction)
  - *n*-bit prediction is not much better than 2-bit prediction (n > 2).
  - A BHT with 4K entries is as good as an infinite size BHT[无限缓冲区]

Correlating Branch Predictors[关联预测]

- Hypothesis[假设]: recent branches are correlated (behavior of recently executed branches affects prediction of current branch)
- Example 1:

```
if (aa==2)
      aa=0;
if (bb==2)
      bb=0;
if (aa!=bb) {
```

```
      addi   x3, x1, -2
      bnez   x3, L1 …        //B1 (aa != 2)
      add    x1, x0, x0      //aa=0
L1:   addi   x3, x2, -2
      bnez   x3, L2          //B2 (bb != 2)
      add    x2, x0, x0      //bb=0
L2:   sub    x3, x1, x2      //x3=aa-bb
      beqz   x3, L3          //B3 (aa == bb)
```

If B1 is not taken (aa==2) and B2 is not taken (bb==2), then B3 will be taken (aa==bb)

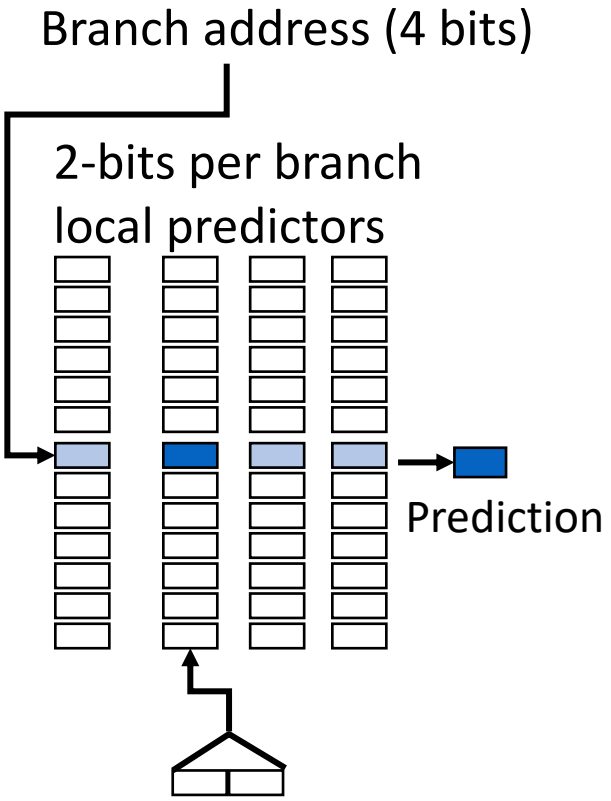
If B1 and B2 are taken (aa!=2, bb!=2), then B3 will probably not be taken

- Example 2:

```
if (d == 0)  d = 1 ;
if (d == 1)  .....
```

Correlating Branch Predictors (cont.)

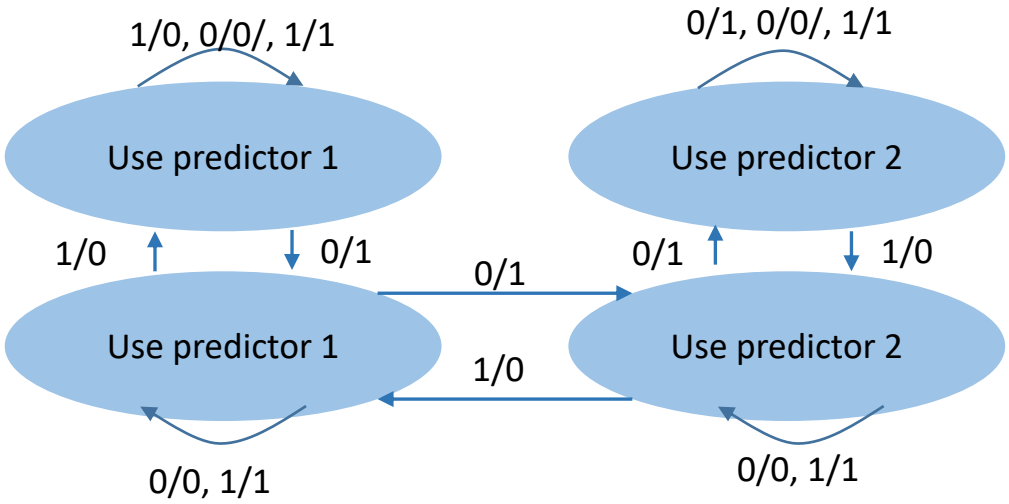
- Keep history of the *m* most recently executed branches in an *m*-bit shift register[移位寄存器]
  - Record the prediction for each branch inst, and each of the 2^m combinations
- In general, *(m,n)* predictor means record last *m* branches to select between 2^m history tables each with *n*-bit predictor
  - Simple access scheme (double indexing).
  - A *(0,n)* predictor is a local *n*-bit predictor.
- Size of table is $N*n*2^m$
  - *N* is the number of table entries
  - There is a tradeoff between *N* (determines collision), *n* (accuracy of local prediction) and *m* (determines history)



2-bit global
branch history
(01 = not taken then taken)

Tournament Predictor[竞赛预测器]

- Combines a global predictor and a local predictor with a strategy for selecting the appropriate predictor (*multi-level* predictors)

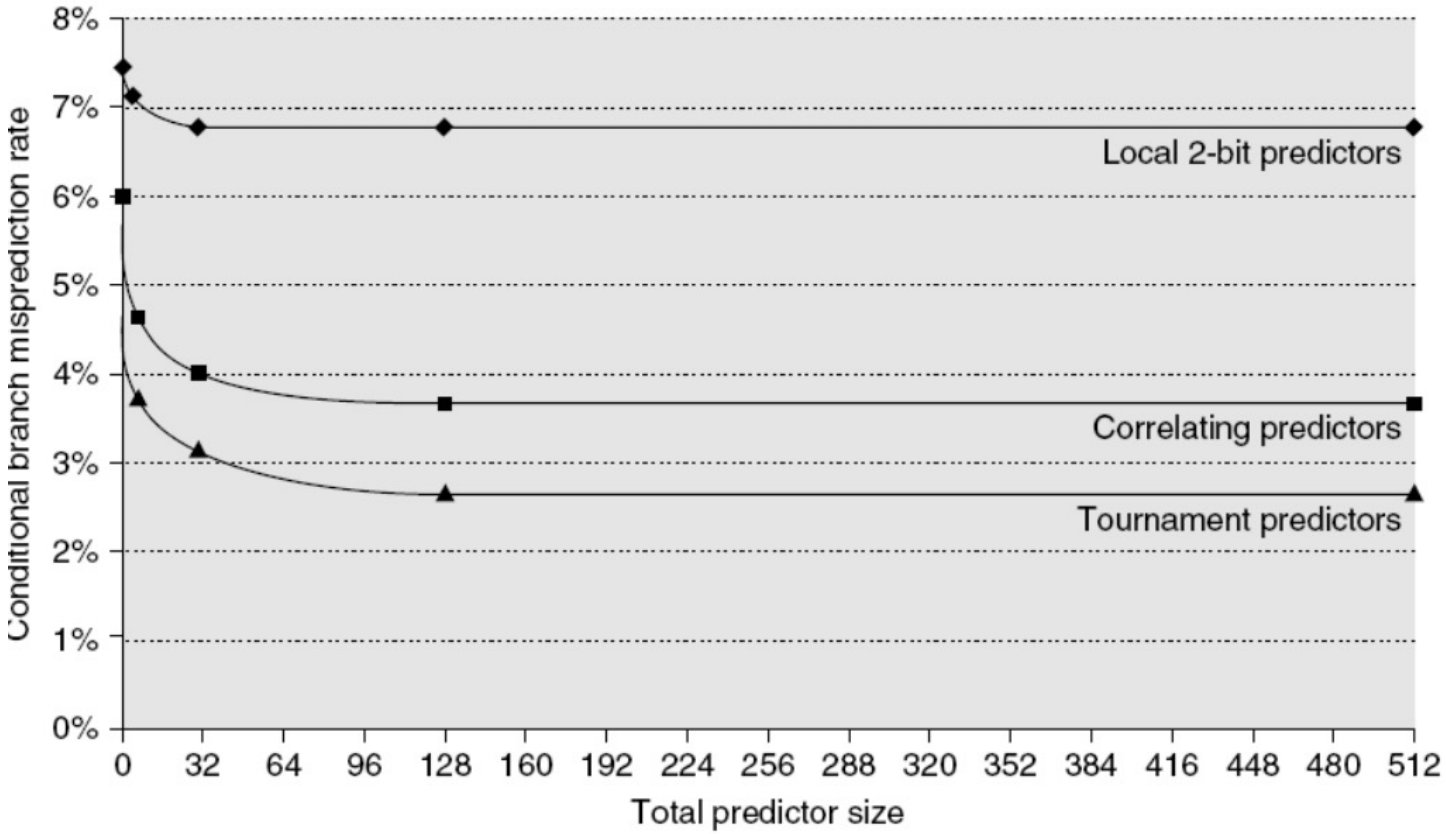


p1/p2 == predictor 1 is correct/ predictor 2 is correct

- The Alpha 21264 selects between
  - A (12,2) global predictor with 4K entries
  - A local predictor which selects a prediction based on the outcome of the last 10 executions of any given branch.

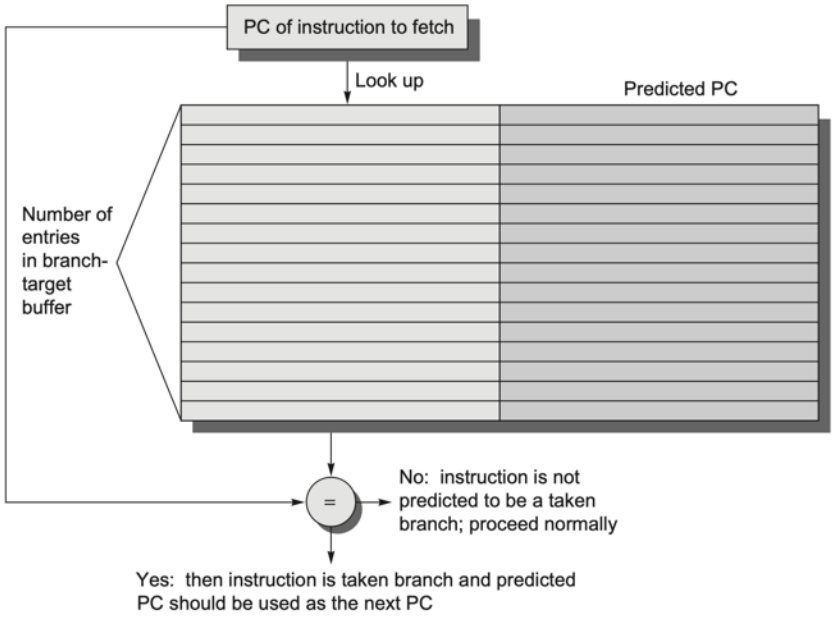
Performance[性能]

- Miss prediction rate for three different predictors



Branch Target Buffers(§3.9)[目标缓冲区]

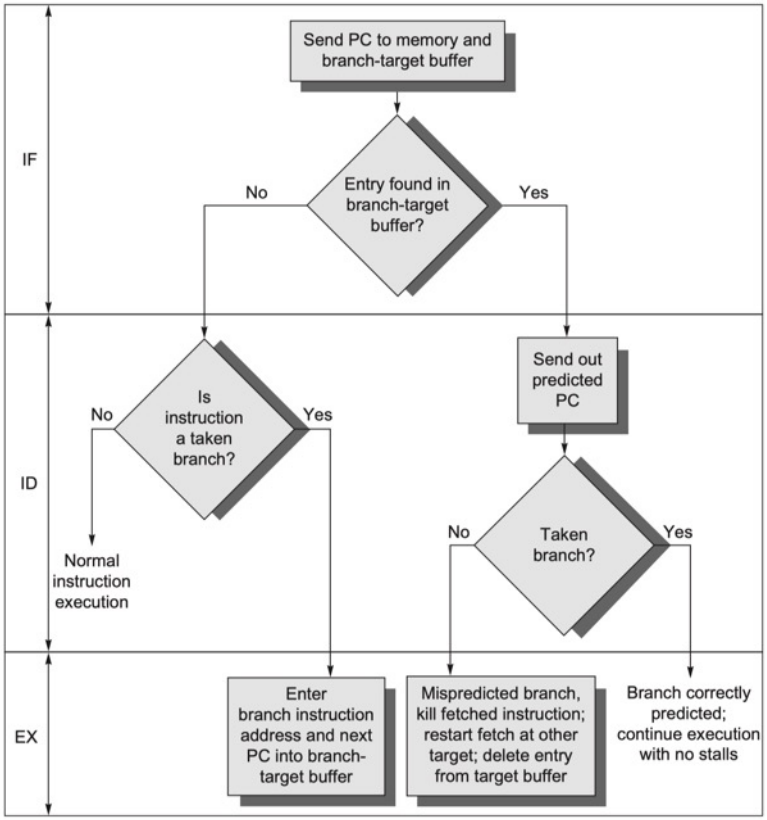
- To increase instruction fetch bandwidth
  - Store the *address* of the branch's target, in addition to the prediction



- 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

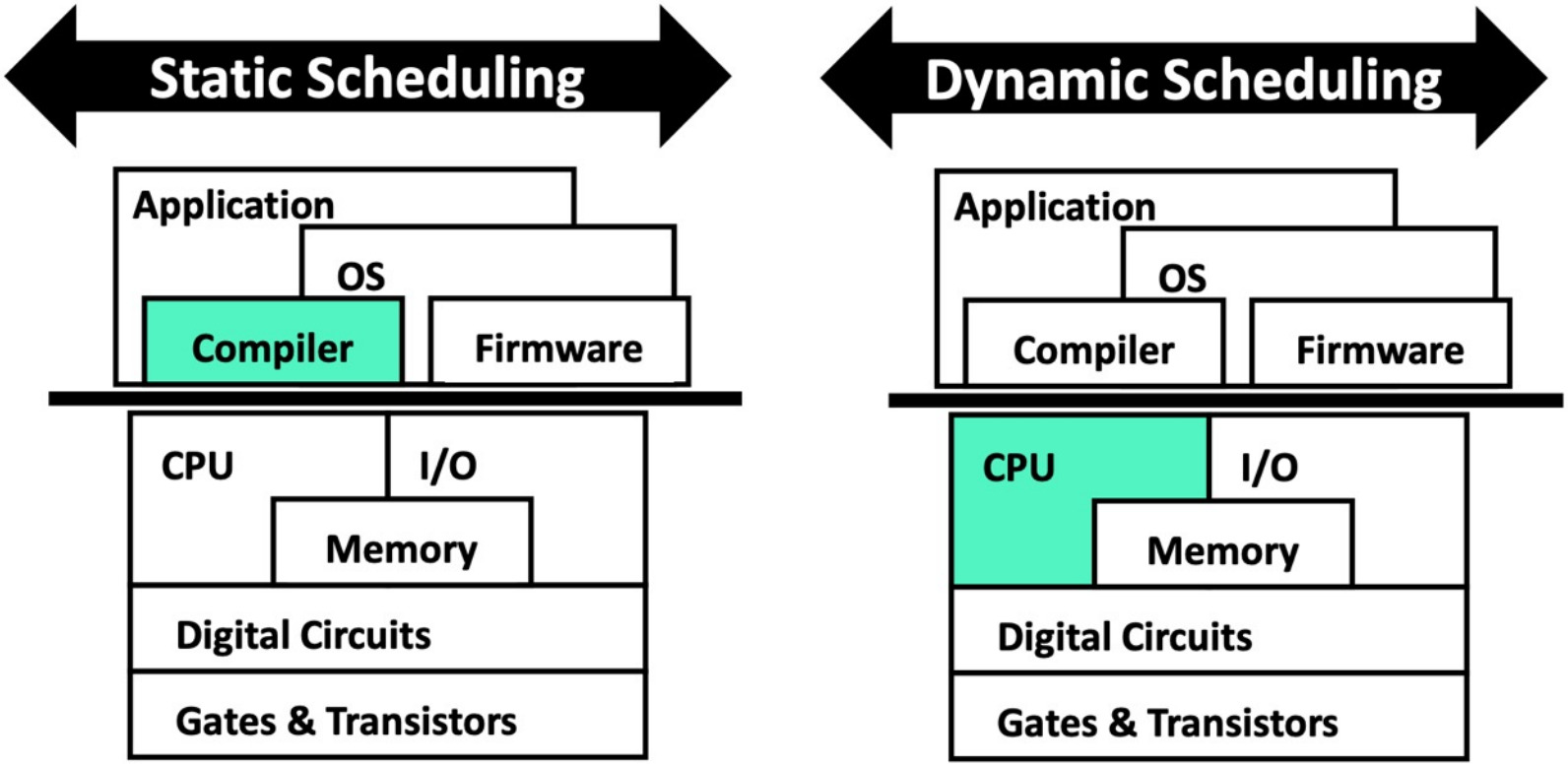
- Assuming that branch condition and target are resolved in *ID* stage



- 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 available 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
```

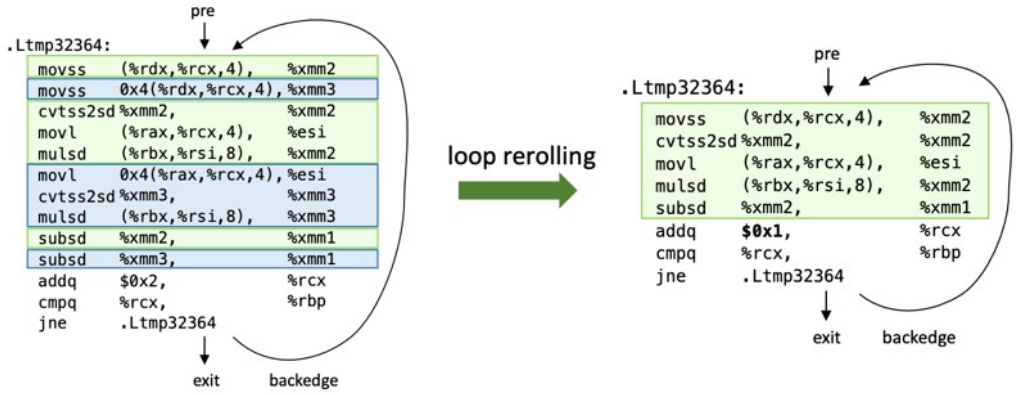
A total of 14 clock cycles
(3.5 cycles per iter)

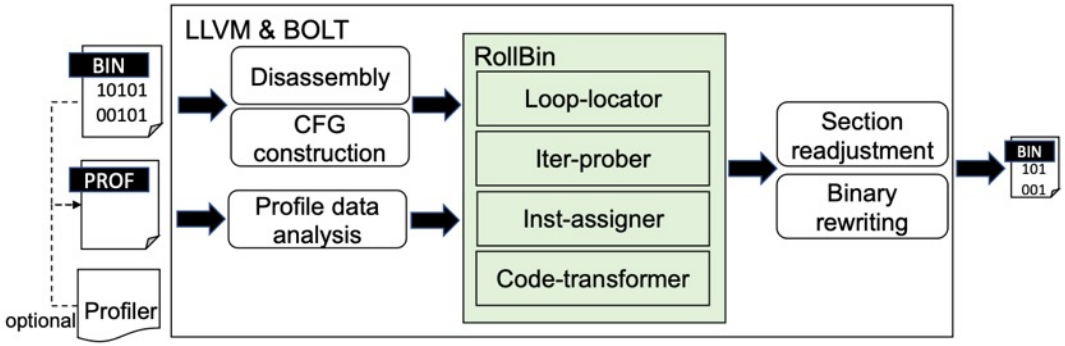
Unrolling Limitations[限制]

- 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].

Compiler Optimization: Example

```
int find_min(const int* array, const int len) {
  int min = a[0];
  for (int i = 1; i < len; i++) {
    if (a[i] < min) { min = a[i]; }
  }
  return min;
}
int find_max(const int* array, const int len) {
   int max = a[0];
   for (int i = 1; i < len; i++) {
     if (a[i] > max) { max = a[i]; }
   }
   return min;
}
void main() {
  int* array, len, min, max;
  initialize_array(array, &len);
  min = find_min(array, len);
  max = find_max(array, len);
  ...
}
```


```
void main() {
   int* array, len, min, max;
   initialize_array(array, &len);
   min = a[0]; max = a[0];
   for (int i = 0; i < len; i++) {
     if (a[i] < min) { min = a[i]; }
     if (a[i] > max) { max = a[i]; }
   }
   ...
}
```

Dynamically Scheduled Pipelines (§3.4)

- Key idea: allow instructions behind stall to proceed

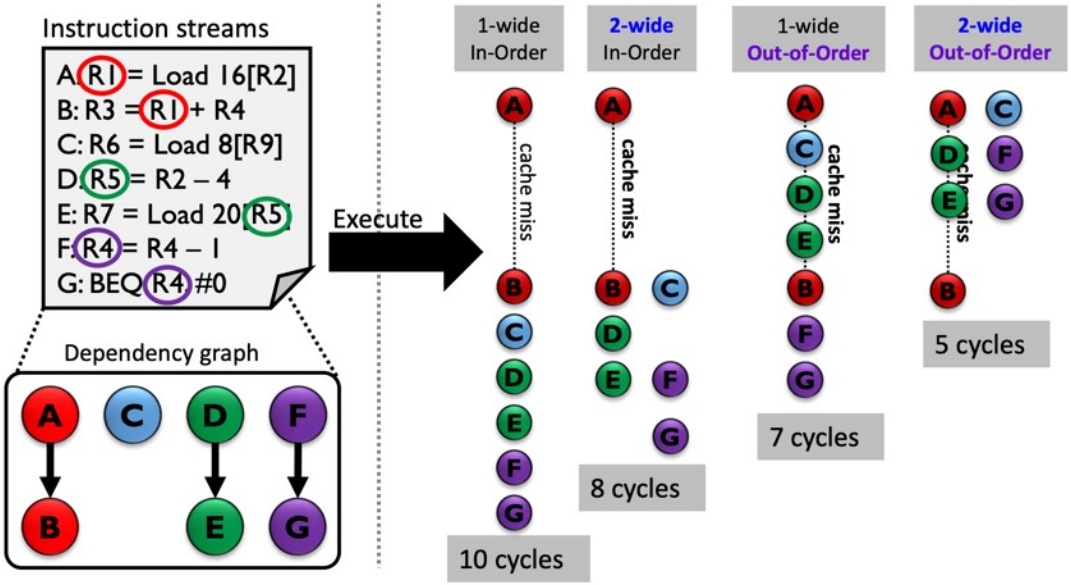
```
fdiv   F0,   F2,   F4
fadd   F10,  F0,  F8
fsub   F12,  F8,   F14
```

RAW -> Stall

No dependency

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

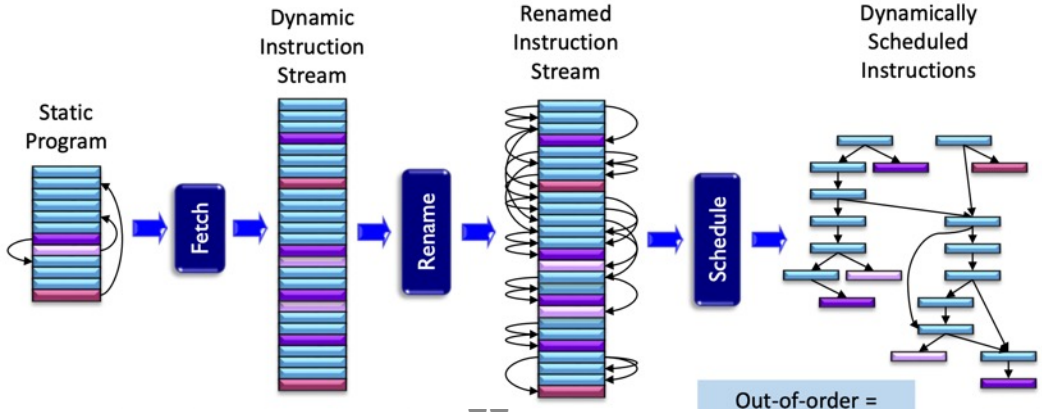
- Hardware rearranges instruction stream to reduce stalls



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

Out-of-order[乱序执行]

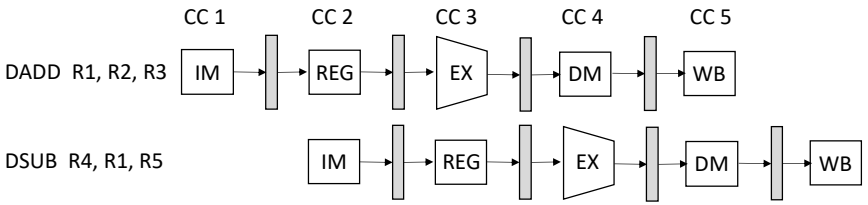
- How can O3 achieve performance benefits?
  - Hardware rearranges instruction stream to reduce stalls
- Any problems of O3?
  - Hazards! Especially for register dependencies
- How does the O3 work?
  - Step1: fetch many instructions into instruction window
  - Step2: rename regs. to avoid false deps.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



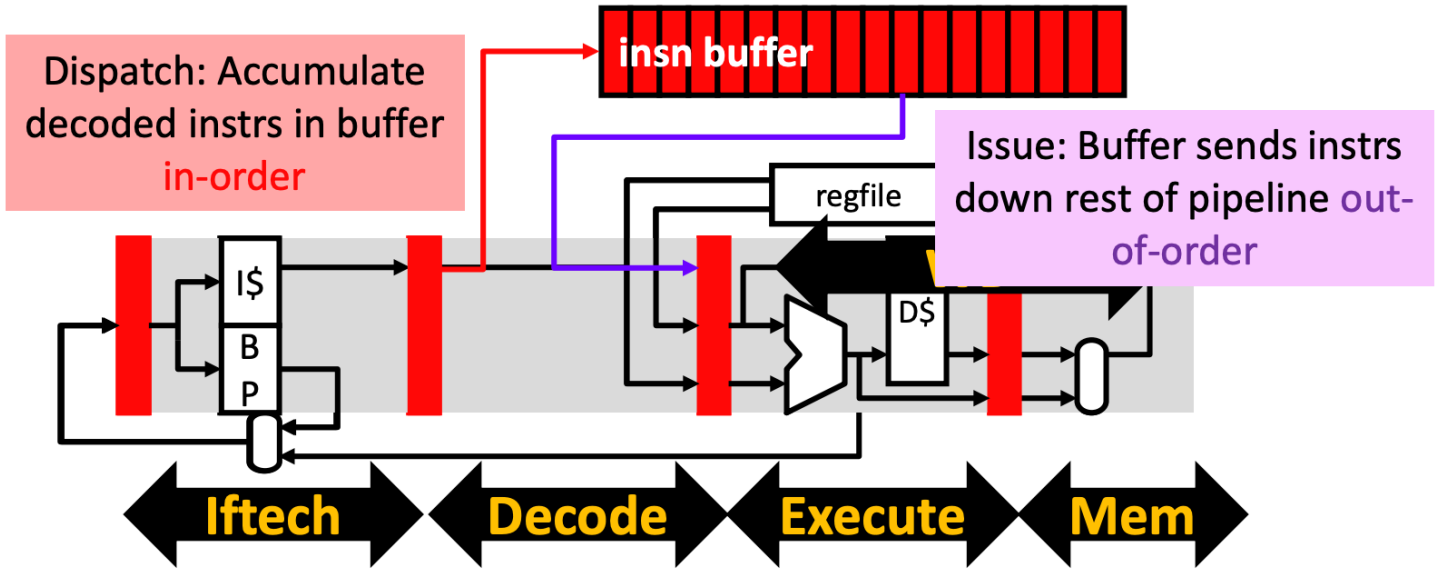
http://camelab.org/uploads/Main/lecture08-scoreboard.pdf

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

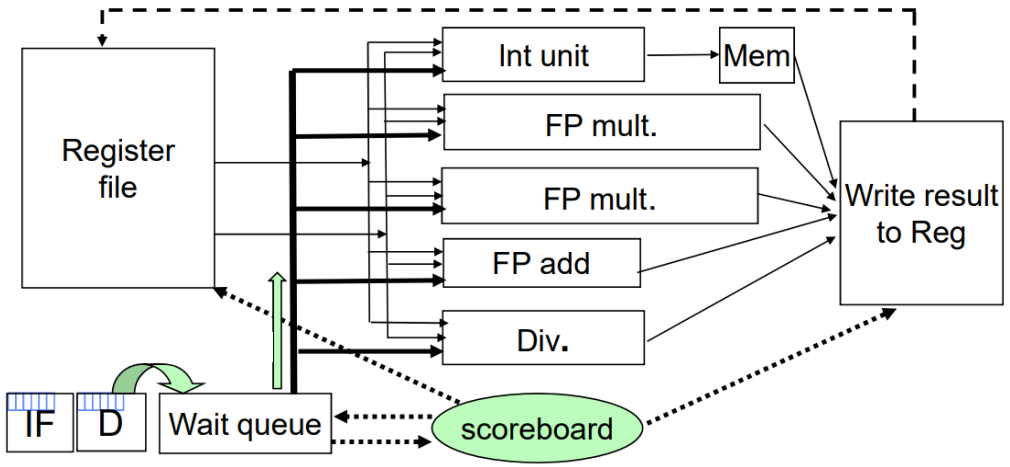


Scoreboard[记分板]

- 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	
...	

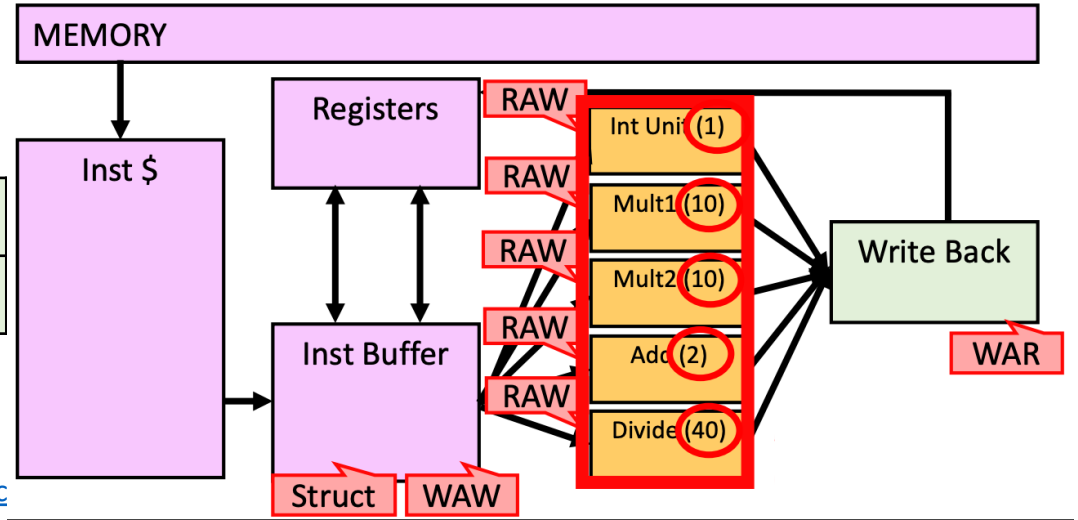
Status Tables

- **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

    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.