361.1.4201
Computer Architecture
Precise Exceptions, State Maintenance, State Recovery

Dr. Guy Tel-Zur

Base on slides by Prof. Onur Mutlu

Carnegie Mellon University

Spring 2015

and comments by Dr. Danny Seidner

Last update: 14/6/2022, 31/5/2023, 10/7/2024, 19/6/2025

Agenda for Today & Next Few Lectures

- Single-cycle Microarchitectures
- Multi-cycle and Microprogrammed Microarchitectures
- Pipelining
- Issues in Pipelining: Control & Data Dependence Handling, State Maintenance and Recovery, …
- Out-of-Order Execution
- Issues in OoO Execution: Load-Store Handling, …

Readings for past and the next few lectures

- **Guy: P&H CO&D, Chapter 4 – "The Processor"**
- **Guy: H&P CAQA, Appendix C - "Pipelining: Basic and**
- **Intermediate Concepts"**
- Smith and Plezskun, "Implementing Precise Interrupts in Pipelined Processors," IEEE Trans on Computers 1988 (earlier version in ISCA 1985).
- Smith and Sohi, "The Microarchitecture of Superscalar Processors," Proceedings of the IEEE, 1995
  - More advanced pipelining
  - Interrupt and exception handling
  - Out-of-order and superscalar execution concepts

Recall: How to Handle Data Dependences

- Anti and output dependences are easier to handle
  - write to the destination only in last stage and in program order

נכון, אך טרם למדנו זאת...

- Flow dependences are more interesting

- Six fundamental ways of handling flow dependences
  - Detect and wait until value is available in register file
  - Detect and forward/bypass data to dependent instruction
  - Detect and eliminate the dependence at the software level
    - No need for the hardware to detect dependence
  - Detect and move it out of the way for independent instructions
  - Predict the needed value(s), execute "speculatively", and verify
  - Do something else (fine-grained multithreading)
    - No need to detect

Review: How to Handle Control Dependences

- Critical to keep the pipeline full with correct sequence of dynamic instructions.

- Potential solutions if the instruction is a control-flow instruction:
- Stall the pipeline until we know the next fetch address
- Guess the next fetch address (branch prediction)
- Employ delayed branching (branch delay slot)
- Do something else (fine-grained multithreading)
- Eliminate control-flow instructions (predicated execution)
- Fetch from both possible paths (if you know the addresses of both possible paths) (multipath execution)

Review of Last Few Lectures

- Control dependence handling in pipelined machines
  - Delayed branching
  - Fine-grained multi-threading
  - Branch prediction
    - Compile time (static)
      - Always NT, Always T, Backward T Forward NT, Profile based
    - Run time (dynamic)
      - Last time predictor
      - Hysteresis: 2BC predictor
      - Global branch correlation --> Two-level global predictor קורלציות
      - Local branch correlation --> Two-level local predictor
      - Hybrid branch predictors
  - Predicated execution (Arm)
  - Multipath execution (RS6000) – רק הזכרנו
  - Return address stack & Indirect branch prediction – לא דיברנו

Pipelining and Precise Exceptions:

Preserving Sequential Semantics

Credit: Modern Processor Design by Shen & Lipasti

מעבד סופר-סקלארי

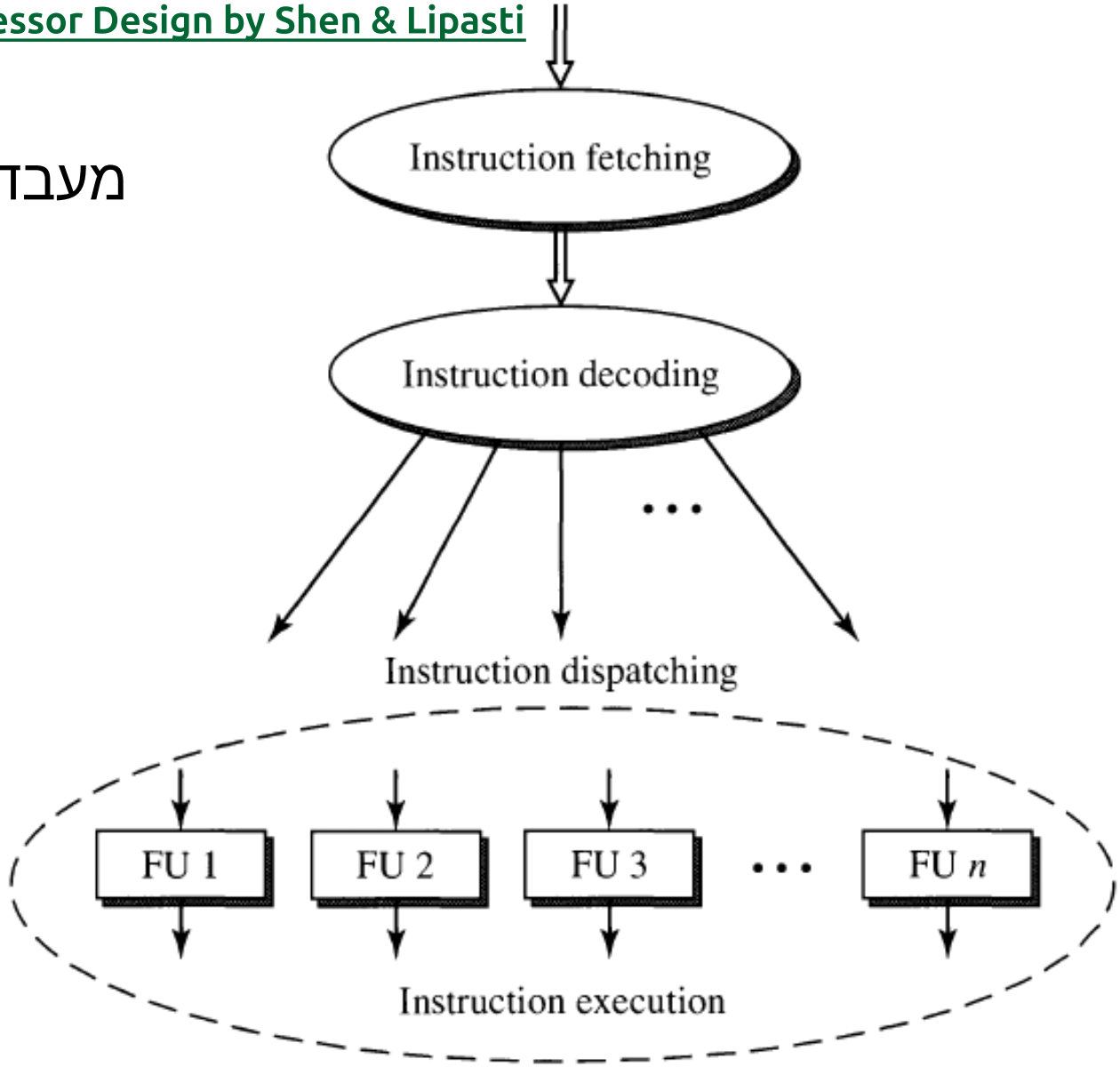
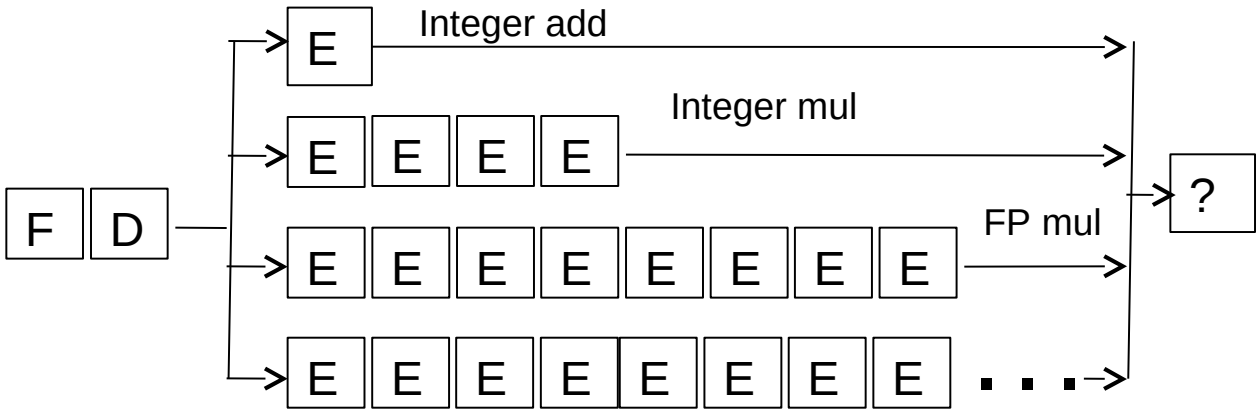


Figure 4.16
The Necessity of Instruction Dispatching in a Superscalar Pipeline.

Multi-Cycle Execution

- Not all instructions take the same amount of time for "execution"

- Idea: Have multiple different functional units that take different number of cycles
  - Can be pipelined or not pipelined
  - Can let <u>independent instructions</u> start execution on a different functional unit before a previous long-latency instruction finishes execution



Issues in Pipelining: Multi-Cycle Execute

- Instructions can take different number of cycles in EXECUTE stage
  - Integer ADD versus FP MULtiply

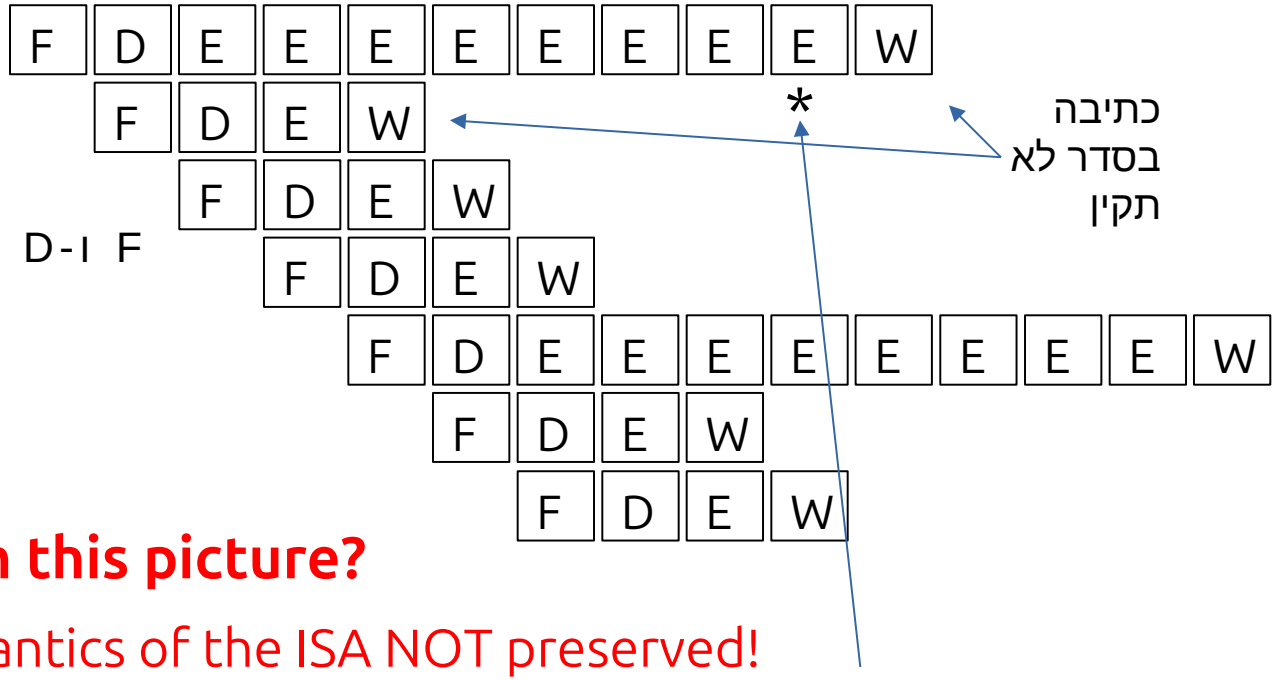
ישנן כמה יחידות E

8 מחזורי ביצוע

FMUL R4 ← R1, R2
ADD R3 ← R11, R12

F ו-D לפי הסדר, לכאורה הכל תקין

FMUL R2 ← R5, R6
ADD R7 ← R15, R16



What is wrong with this picture?

- Sequential semantics of the ISA NOT preserved!
- What if FMUL incurs (גורם להתרחשות) an exception? * למשל בשלב
  - אי אפשר לדבג מצבים כאלה. סתירה למכונת פון נוימן
  - אם יש הוראות עם זמן שהות שונה לא ניתן לשחזר את המצב הארכיטקטוני

Exceptions and Interrupts

- "Unplanned" changes or interruptions in program execution
- Due to **internal** problems in execution of the program

  → Exceptions = חריגה
- Due to **external** events that need to be handled by the processor

  → Interrupts = פסיקה
- Both exceptions and interrupts require
  - stopping of the current program
  - saving the architectural state
  - handling the exception/interrupt → switch to handler
  - return back to program execution (if possible and makes sense)

Exceptions vs. Interrupts

- Cause
- Exceptions: internal to the running thread . למשל חלוקה באפס
- Interrupts: external to the running thread

לחיצה על מקש (מקור חיצוני)

למי שיצא לעבוד עם דבגר (תכף נדגים).
break point הוא למעשה exception.
ההנחה היא שכל מה שקרה לפני כן הסתיים, אחרת לא ניתן לדבג.

- When to Handle
  - Exceptions: when detected (and known to be non-speculative)
  - Interrupts: when convenient
    - Except for very high priority ones
      - Power failure
      - Machine check (error), e.g error in the cache

חשיבות החריגה תלויה בחשיבות התהליך

- Priority: process (exception), depends (interrupt)

גיא: Page Fault, divide by zero – internal to the thread

- Handling Context: process (exception), system (interrupt)

בהמשך לשקף הקודם...

<u>גורם הסיבה:</u>

חריגה – אירוע פנימי בשביל ה- THREAD.

פסיקה – אירוע חיצוני ל-THREAD.

<u>מתי לטפל:</u>

חריגה – כאשר מתגלה, במיידי.

פסיקה – כאשר "נוח" (להוציא מקרים חריגים כגון תקלת מטמון או ירידת אספקת כוח...).

<u>תעדוף:</u>

חריגה מחייבת טיפול.

פסיקה – זה תלוי.

<u>קונטקסט (היקף ההתייחסות)</u>

חריגה: ברמת התהליך

פסיקה: ברמת המערכת.

המצב הארכיטקטוני מתגלה לנו באמצעות הדבאגר

File Edit View Program Commands Status Source Data

(): ./fibonacci.c:12

```
#include <string.h>
#include <stdio.h>
int fib(int n)
{
  int i, j;
  if (n<2)
    return n;
  else
    {
      i=fib(n-1);
      j=fib(n-2);
      return i+j;
    }
```

```
0x0000555555555163 <+26>:    mov    -0x14(%rbp),%eax
0x0000555555555166 <+29>:    sub    $0x1,%eax
0x0000555555555169 <+32>:    mov    %eax,%edi
0x000055555555516b <+34>:    callq  0x555555555149 <fib>
```

```
(gdb) step
fib (n=3) at ./fibonacci.c:4
(gdb) step
(gdb) step
(gdb)
```

Step

Registers

rax	0x2	2
rbx	0x5555555551d0	93824992235984
rcx	0x5555555551d0	93824992235984
rdx	0x1	1
rsi	0x7fffffffd078	140737488343160
rdi	0x2	2
rbp	0x7fffffffce00	0x7fffffffce00
rsp	0x7fffffffcde0	0x7fffffffcde0
r8	0x0	0
r9	0x7ffff7fe0d50	140737354009936
r10	0x7	7
r11	0x2	2

Integer registers All registers

Close Help

Run Interrupt Step Stepi Next Nexti Until Finish Cont Kill Up Down Undo Redo Edit Make

```
1/code $ gcc -g -o fibonacci ./fibonacci.c
1/code $ ./fibonacci

1/code $ ddd ./fibonacci
C
```

Precise Exceptions/Interrupts

- **The architectural state should be consistent (precise) when the exception/interrupt is ready to be handled**

בהתאם למודל פון–נוימן:

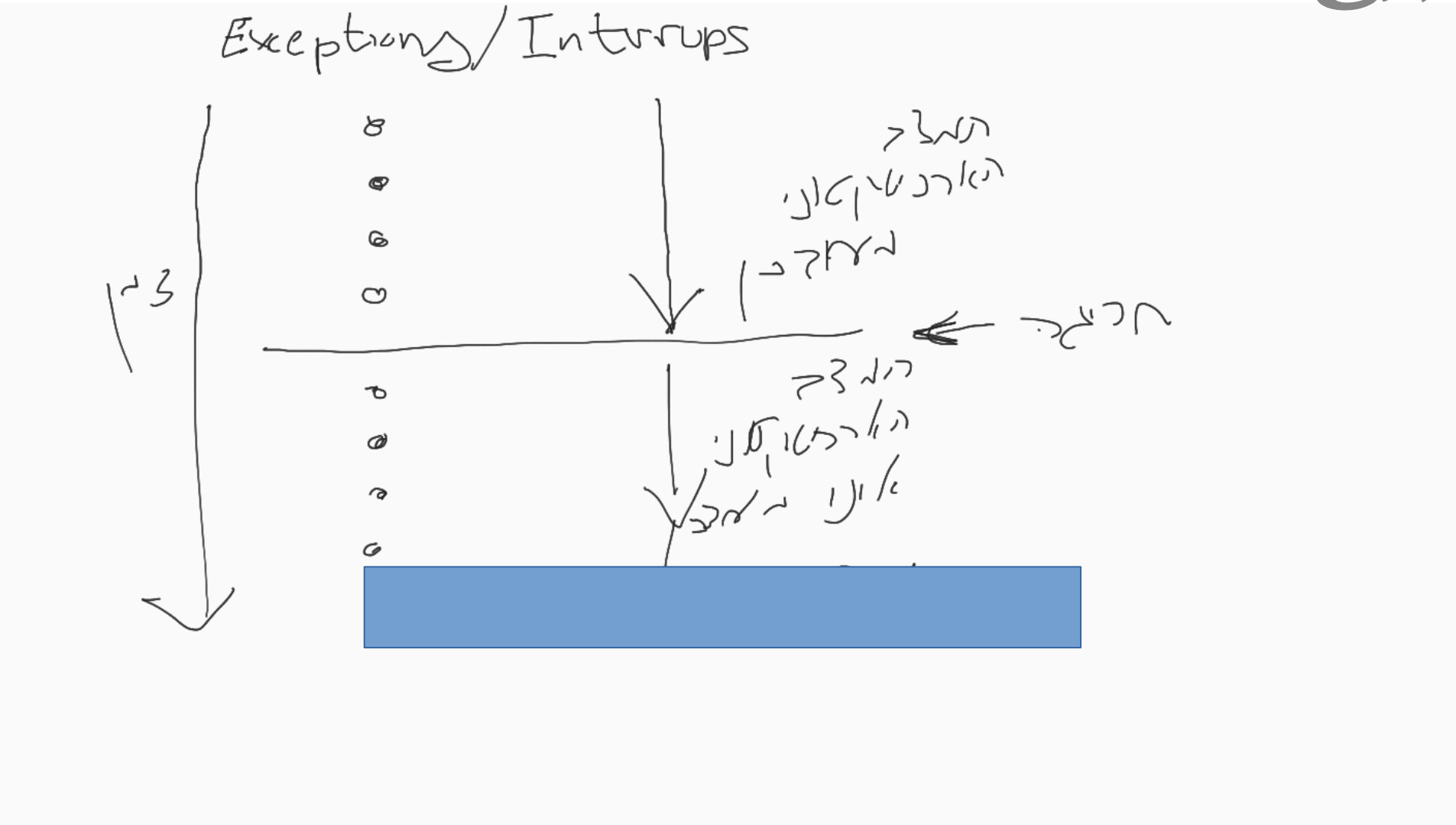
1. All previous instructions should be completely retired.

2. No later instruction should be retired.

Retire = commit = finish execution and update arch. state
retired=סיימו לעדכן את המצב הארכיטקטוני

Checking for and Handling Exceptions in Pipelining

- When the oldest instruction ready-to-be-retired is detected to have caused an exception, the control logic should:
  - Ensures architectural state is **precise** (register file, PC, memory)
  - Flushes all younger instructions in the pipeline
  - Saves PC and registers (as specified by the ISA)
  - Redirects the fetch engine to the appropriate exception handling routine



ציות לכללי הממשק בין החומרה לתכנה

Why Do We Want Precise Exceptions?

- Semantics of the von Neumann model ISA specifies it
  - Remember von Neumann vs. Dataflow
- Aids software debugging – לצורך זה נדרש שיחזור מצב ארכיטקטוני, לשאר המקרים אפשר גם לשמור מצב חלקי אך זה מסובך
- Enables (easy) recovery from exceptions, e.g. page faults

Aside: A memory page fault occurs when a program tries to access data that is not currently loaded in the physical memory (RAM). The operating system then retrieves the required data from secondary storage, allowing the program to continue running.

- Enables (easily) restartable processes

אם מה שרוצים זה יכולת לעשות אתחול אז מה שנדרש היא יכולת להוריד אל הזיכרון את כל המצב המיקרוארכיטקטוני ואח"כ לקרוא אותו בחזרה

- Enables traps into software (e.g., software implemented opcodes)

A try-catch exception is a programming construct used to handle errors in code. The "try" block contains code that might cause an error, and if an error occurs, the "catch" block executes to handle the error gracefully, allowing the program to continue running.

הדגמה: חלוקה באפס

demo:

/home/telzur/science/Teaching/CPU/lectures/11/code/except.c

File Edit View Program Commands Status Source Data

./except.c:11

```
  int x;
  n = 5;
  for (i=n;i>-2;i--)
        x = n/i;
return 0;
}
```

```
0x0000555555555514b <+34>:    idivl   -0xc(%rbp)
0x0000555555555514e <+37>:    mov     %eax,-0x4(%rbp)
0x0000555555555151 <+40>:    subl    $0x1,-0xc(%rbp)
0x0000555555555155 <+44>:    cmpl    $0xffffffff,-0xc(%rbp)
```

```
(gdb) step

Program terminated with signal SIGFPE, Arithmetic exception.
The program no longer exists.
(gdb)
```

Program terminated with signal SIGFPE, Arithmetic exception. The program no longer exists.

Registers

The program has no registers now.

Integer registers All registers

Close Help

```
./except

ddd ./except
```

Run Interrupt Step Stepi Next Nexti Until Finish Cont Kill Up Down Undo Redo Edit Make

RISC-V tolerates div by 0 and no exception will be thrown

http://127.0.0.1:5000/

target remote :1234 see: readme

Load Binary ./except.riscv reverse NI SI

show filesystem | fetch disassembly | reload file | intel | jump to line | /home/telzur/science/Teaching/CPU/lectures/11/code/except.c:14 (20 lines total)

```
1  // 2.6.2021, Guy Tel-Zur
2  //   a demo code for CPU architecture course, lecture 11 - precise expetion.
3  //   compile: gcc -g -o ./except ./except.c
4  //   check this code under ddd debugger. add a breakpoint on line 16.
5
6  #include <string.h>
7  #include <stdio.h>
8
9
10 int main(int arc, char **argv)
11 {
12   int i, n;
13   int  x;
14   n = 5;
15   for (i=n;i>-2;i--)
```

```
0x101ec addi sp,sp,-48
0x101ee sd ra,40(sp)
0x101f0 sd s0,32(sp)
0x101f2 addi s0,sp,48
0x101f4 mv a5,a0
0x101f6 sd a1,-48(s0)
0x101fa sw a5,-36(s0)
0x101fe li a5,5
0x10200 sw a5,-24(s0)
0x10220 lw a5,-20(s0)
0x10224 addiw a5,a5,-1
0x10226 sw a5,-20(s0)
0x1022a lw a5,-20(s0)
0x1022e sext.w a4,a5
0x10232 li a5,-1
0x10234 bge a4,a5,0x1020e <main+34>
```

threads

local variables

no locals in this context

expressions

expression or variable

no expressions in this context

Tree

width (px) height (px)

memory

start address end address (8 no memory to display

breakpoints

main thread groups: i1 condition

/home/telzur/science/Teaching/CPU/lectures/11/code/except.c:14

n = 5;

signals

send SIGINT to gdb (pid 1970500)

other pid pid

registers

no data to display

```
warning: BFD: warning: system-supplied DSO at 0x2aaaab2ac000 has a
corrupt string table index
0x000000000001014e in _start ()
(gdb) [Inferior 1 (process 1970796) exited normally]

(gdb) target remote :1234
warning: `target:/home/telzur/science/Teaching/CPU/lectures/11/code
/except.riscv' has disappeared; keeping its symbols.
Remote debugging using :1234
0x000000000001014e in _start ()
(gdb) [Inferior 1 (process 1970899) exited normally]
```

```
1-stack-list-frames
Fetching assembly since file is missing
2-data-disassemble -s 0x1014e -e 0x101b2 -- 0
-exec-continue
-exec-arguments
-file-exec-and-symbols ./except.riscv
-break-insert main
-break-list
-break-list
4-data-disassemble -f /home/telzur/science/Teaching/CPU/lectures/11
/code/except.c -l 14 -n 1000 -- 4
```

```
Program output -- Programs being debugged are connected to this ter
minal. You can read output and send input to the program from here.
/bin/bash: line 1: /home/telzur/science/Teaching/CPU/lectures/11/co
de/target:/home/telzur/science/Teaching/CPU/lectures/11/code/except
.riscv: No such file or directory
```

Why RISC-V tolerates divide by zero and doesn't throw an exception?

What RISC-V does:
Integer divide-by-zero returns all 1s (0xFFFFFFFF = -1 for signed, UINT_MAX for unsigned). No exception.

Why:

1. Simplicity — Exception handling adds hardware complexity (priority logic, trap vectors, saved state). RISC-V aims for minimal, clean architecture.

2. Performance — Not checking for zero before every division saves cycles. The hardware just does the operation.

3. Consistent, defined behavior — Returning all 1s is predictable. Software that cares can explicitly check for zero first. It's easier to add a check in software than to handle exceptions when you don't want them.

4. Some code wants that value — Saturating arithmetic, certain crypto routines, or bitwise operations may actually need 0xFFFFFFFF. Having a defined result is more useful than crashing.

5. Comparison:

   - x86 and MIPS → throws #DE exception (divide error)
   - ARM → configurable, can trap or return all 1s
   - RISC-V → returns all 1s, no trap

Bottom line: RISC-V follows the principle of "hardware does the simple thing, software handles edge cases when needed." It's not tolerating bad behavior — it's explicitly defined.

Ensuring Precise Exceptions

- Easy to do in single-cycle and multi-cycle machines

- Single-cycle
  - Instruction boundaries == Cycle boundaries

- Multi-cycle
  - Add special states in the control FSM that lead to the exception or interrupt handlers
  - Switch to the handler only at a precise state → before fetching the next instruction

See H&H Section 7.7 for a treatment of exceptions in multi-cycle uarch

Precise Exceptions in MIPS Multi-Cycle Datapath

תמיכה בשתי חריגות

EPC register: Holds the exception causing PC

Cause register: Holds the cause of the exception

Exception Handler starts at address 0x80000180

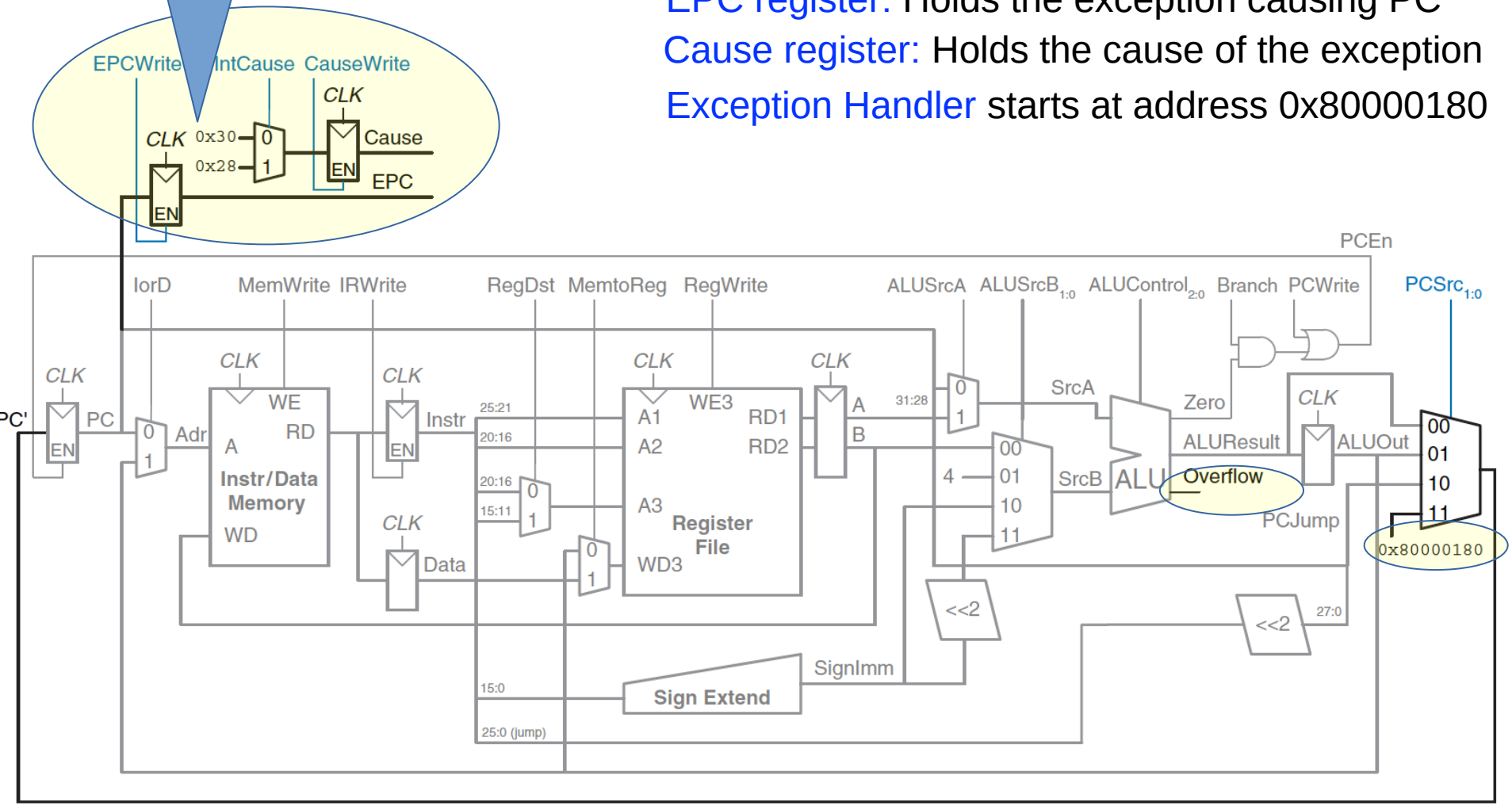
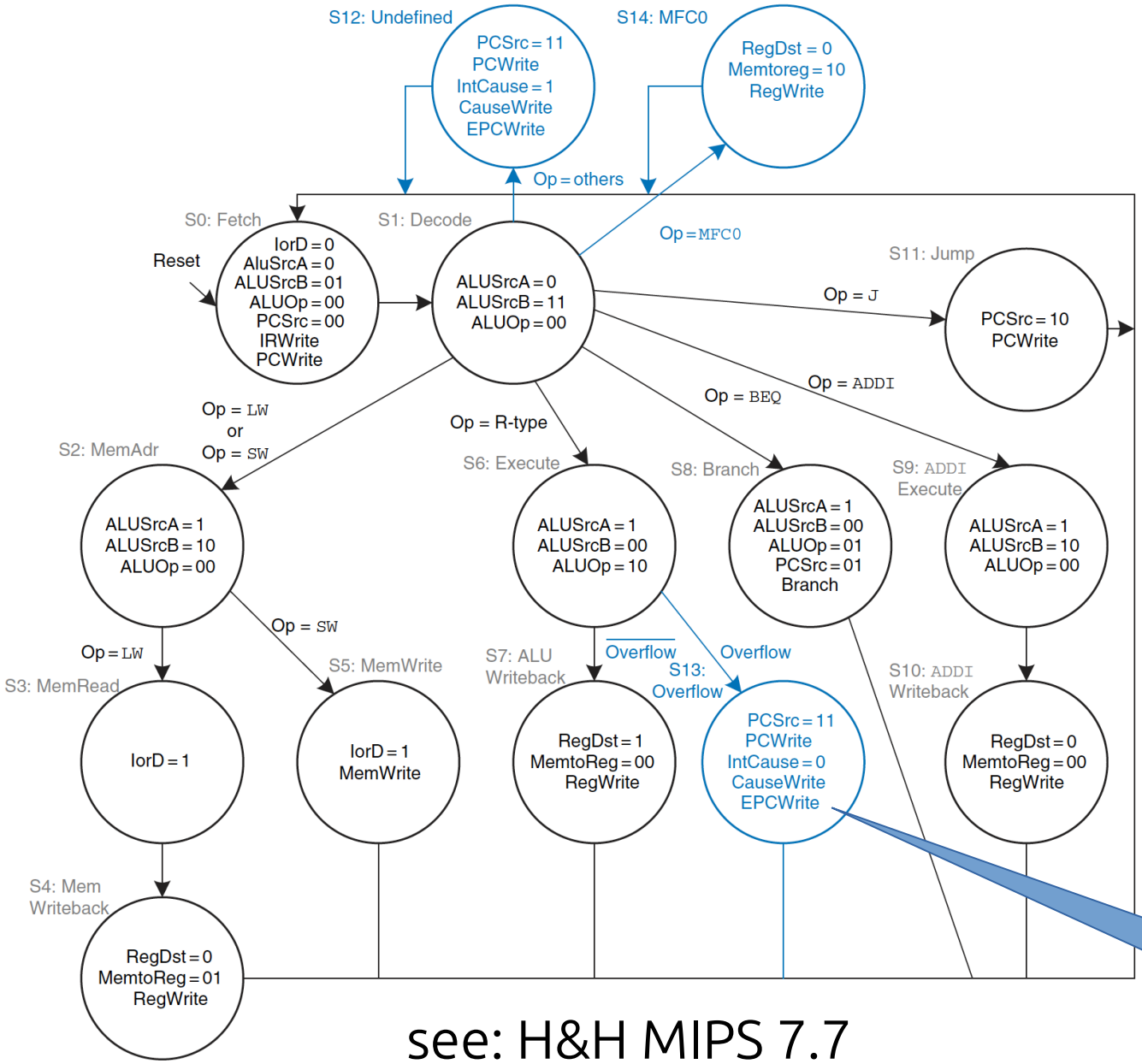


Figure 7.62 Datapath supporting overflow and undefined instruction exceptions

See H&H Section 7.7 for a treatment of exceptions in multi-cycle uarch

Precise Exceptions in MIPS Multi-Cycle FSM



- Supports
  - Overflow
  - Undefined instruction
- mfc0 instruction is used to copy the exception cause into a general-purpose register

see: H&H MIPS 7.7

Figure 7.64 Controller supporting exceptions and `mfc0`

במחזור הבא תתרחש
קפיצה ל- ECP handler

Precise Exceptions in MIPS Multi-Cycle Datapath

In summary, an exception causes the processor to jump to the exception handler. The exception handler saves registers on the stack, then uses `mfc0` to look at the cause and respond accordingly. When the handler is finished, it restores the registers from the stack, copies the return address from `EPC` to `$k0` using `mfc0`, and returns using `jr $k0`.

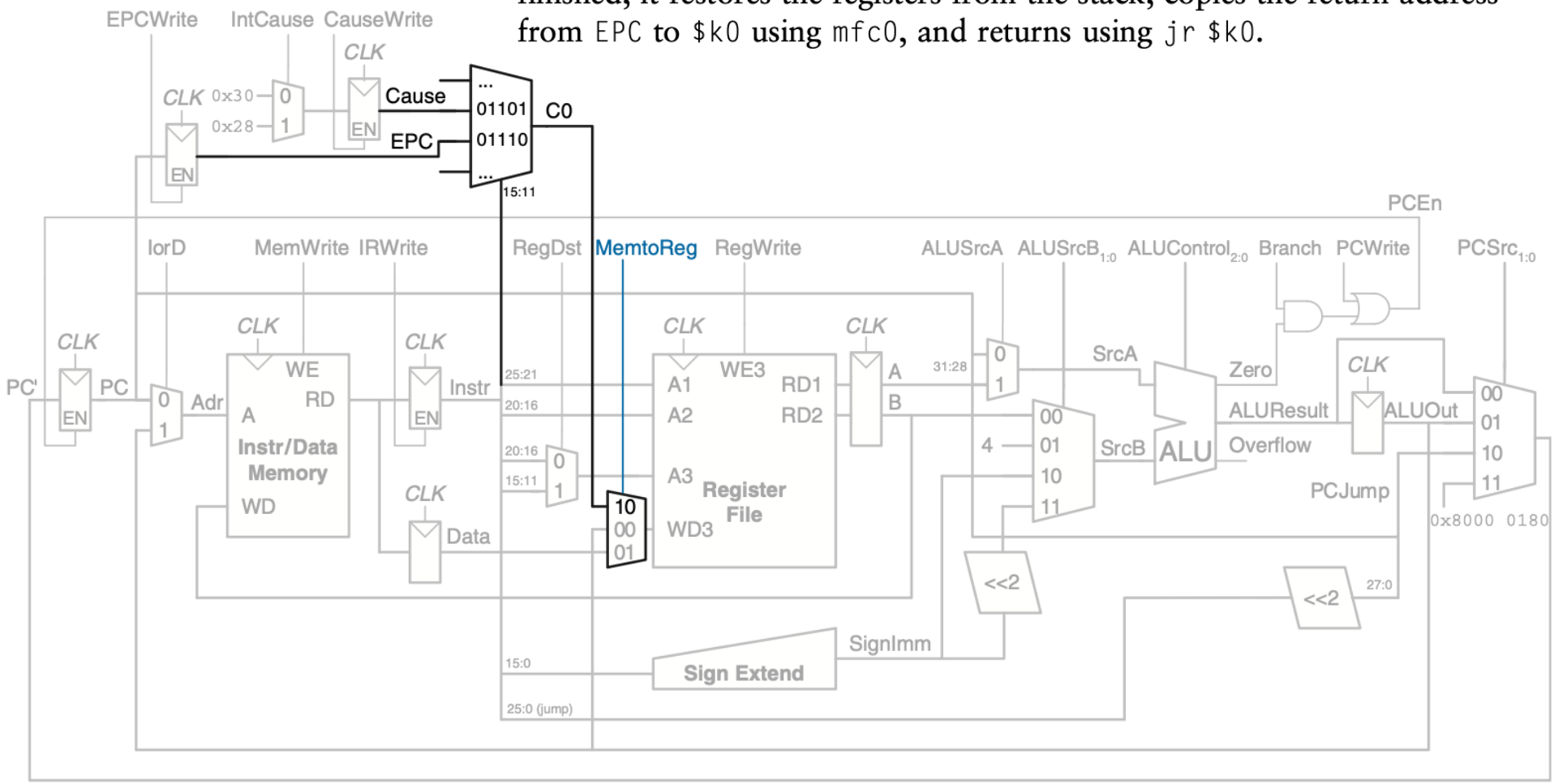
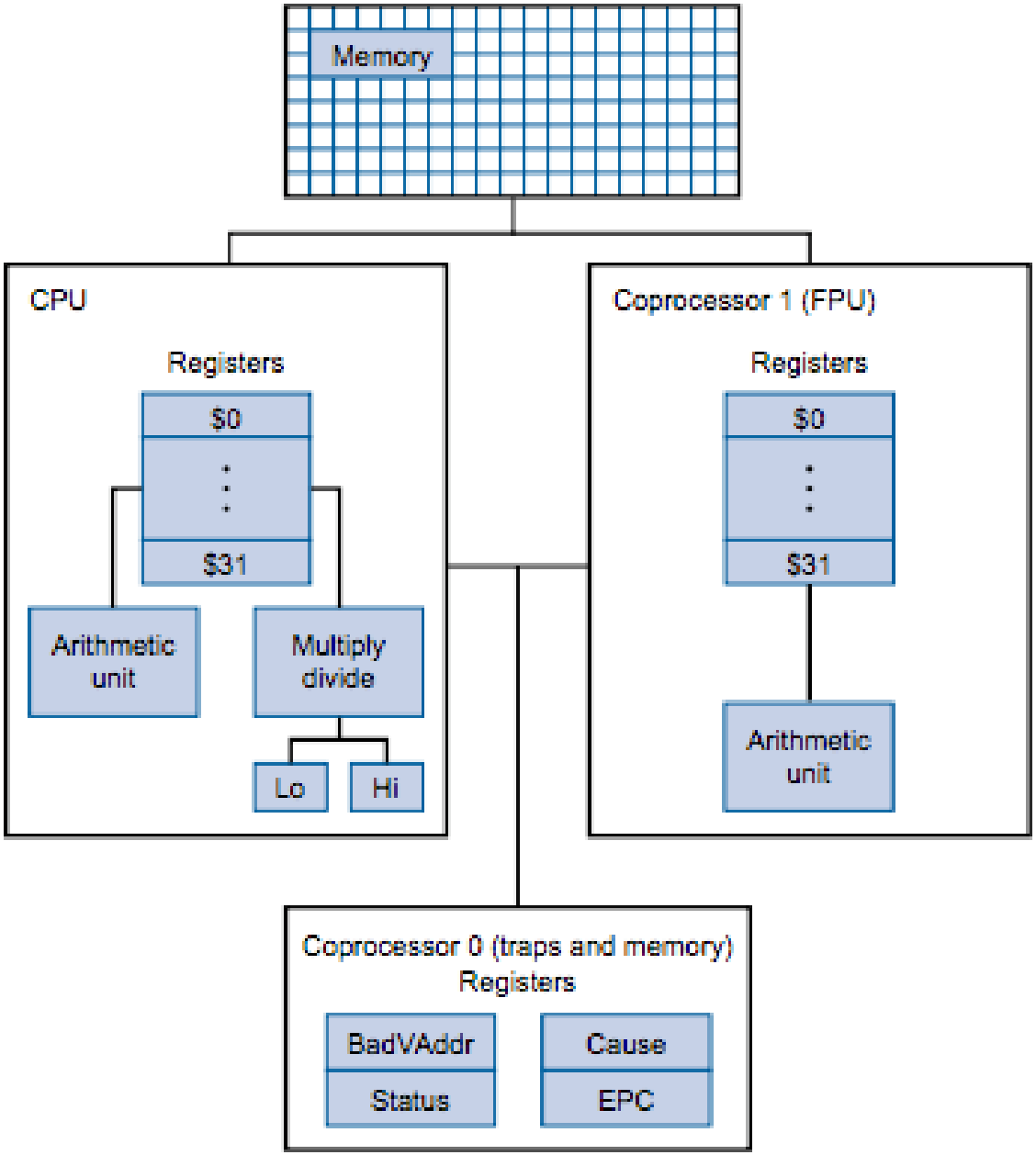
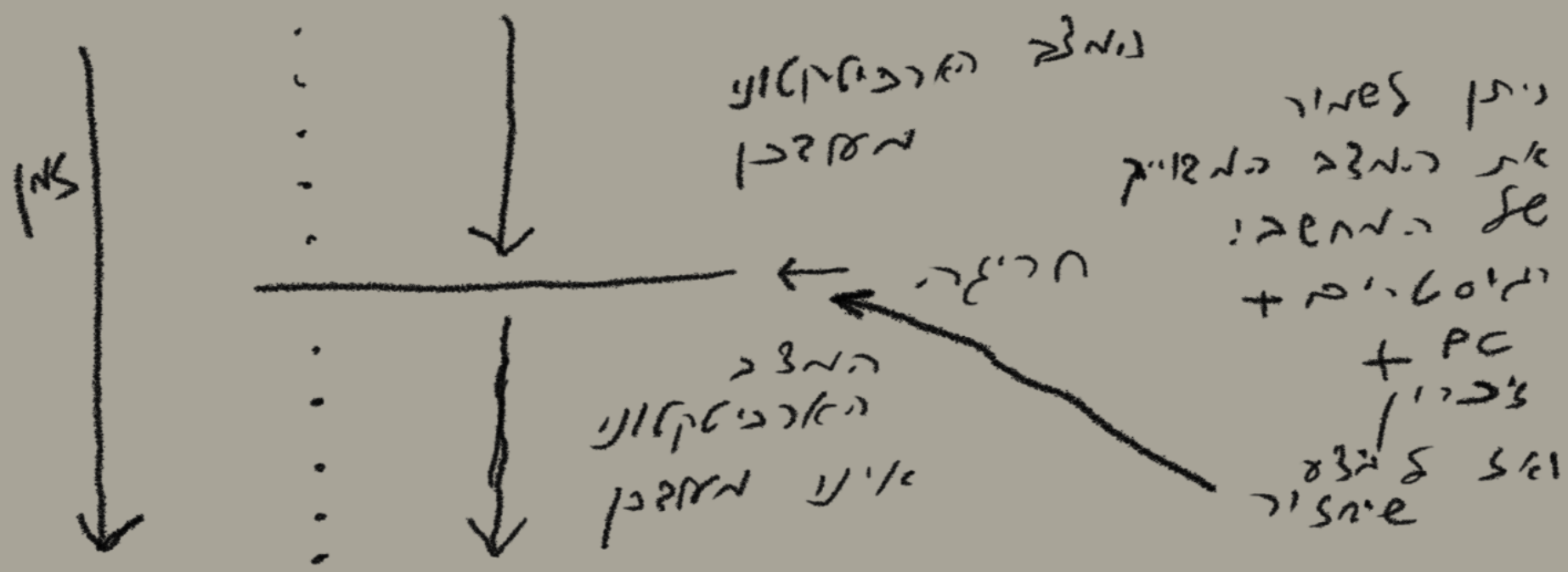


Figure 7.63 **Datapath supporting** `mfc0`

Mips coprocessor 0



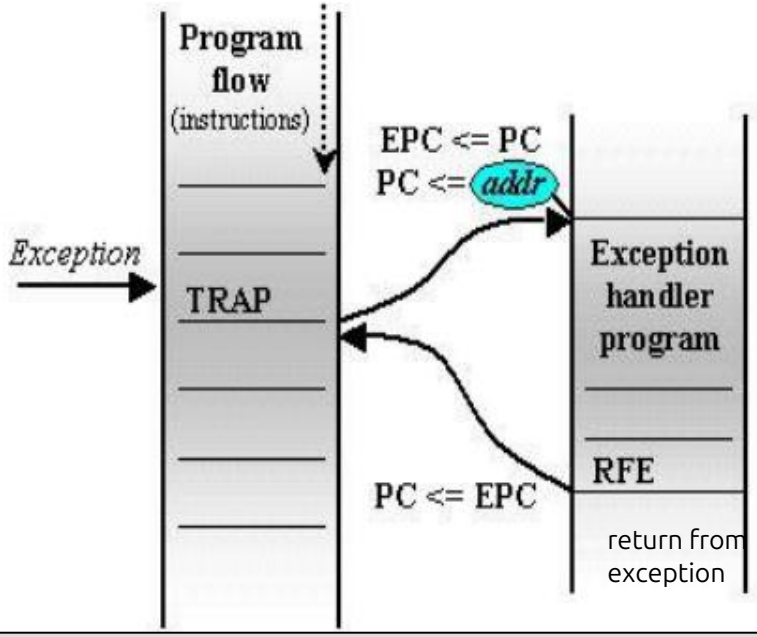
Exception/Interrupt



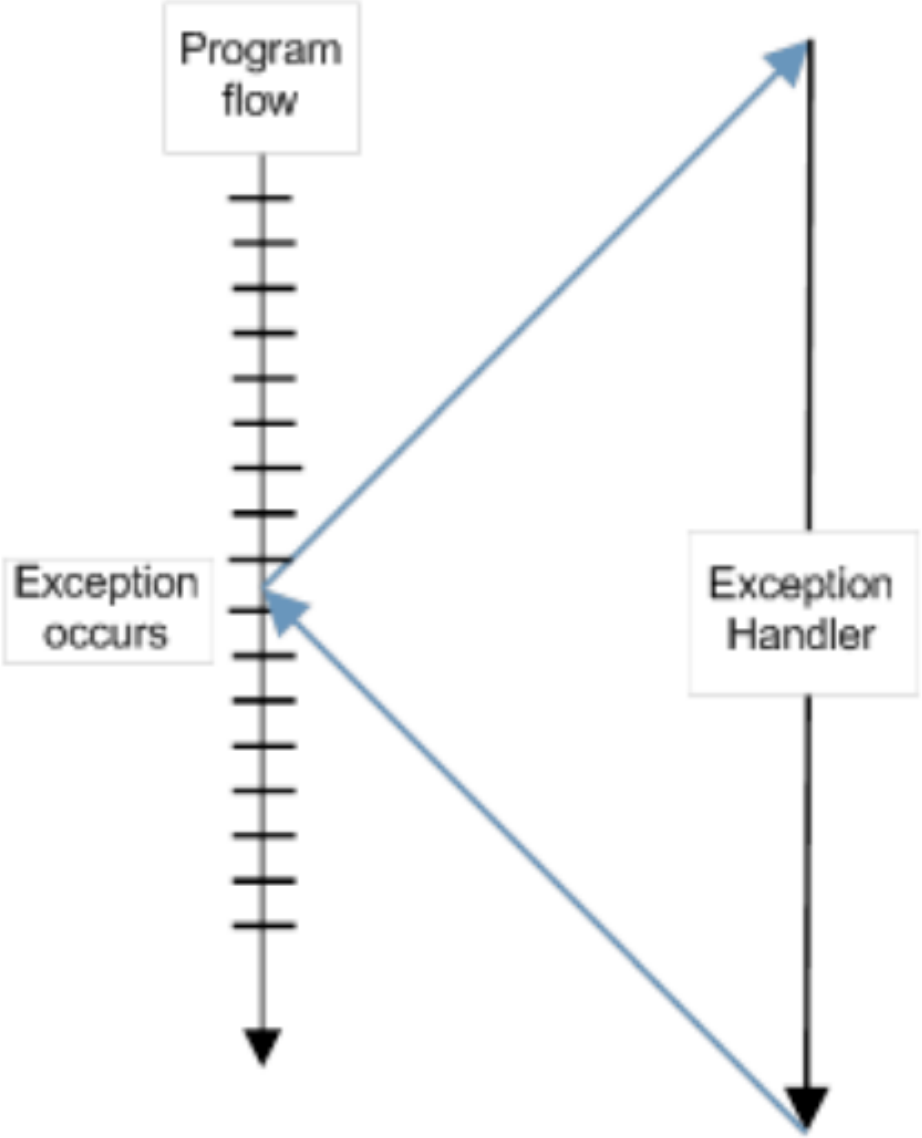
גיא תל-צור

Interrupts and Exceptions

- What to do?
  - Execute code in response to event (handler)
    - Save PC (EPC reg,)
    - Record cause (Cause reg.)
    - Set new PC (4)
  - Return from handler
    - Restore PC
    - Enable e/i (shift Status reg.)
- Determining type of exception
  - Use *vectored* exceptions
    - Infer type from address
  - Use *polled* exceptions
    - Use Cause register
    - This is what MIPS does



Vectored vs. Polled exception: A vectored exception uses a specific address in memory to directly call the appropriate interrupt handler based on the type of exception, allowing for faster response times. In contrast, a polled exception requires the CPU to continuously check each device to see if it needs attention, which can be less efficient.


Program flow
Exception occurs
Exception Handler

Multi-Cycle Execute: More Complications

- Instructions can take different number of cycles in EXECUTE stage → **This complicates exception/interrupt handling**
  - Integer ADD versus Integer DIVide

Exception-causing instruction

DIV R4 ← R1, R2	F	D	E	E	E	E	E	E	E	E	W										
ADD R3 ← R1, R2		F	D	E	W																
			F	D	E	W															
				F	D	E	W														
DIV R2 ← R5, R6					F	D	E	E	E	E	E	E	E	E	W						
ADD R7 ← R5, R6						F	D	E	W												
							F	D	E	W											

  - What is wrong with this picture in a Von Neumann architecture?
    - Sequential semantics of the ISA NOT preserved!
    - What if DIV incurs an exception?

The clean and minimalist RISC-V way

Register	Purpose
mtvec	Trap vector — address of exception handler
mcause	Exception cause code (what happened)
mepc	Saved PC where exception occurred
mtval	Additional info (faulting address, instruction)
mstatus	Status flags (interrupt enable, privilege mode)

1. Trap occurs (exception or interrupt)
2. Hardware auto-saves: mepc ← PC, mcause ← reason, mtval ← details, mstatus flags updated
3. Jump to mtvec handler address
4. Handler runs — reads mcause, decides what to do
5. Return via MRET/SRET/uret — restores PC and privilege mode

Exception Codes (mcause)

- 0 = instruction address misaligned
- 1 = instruction access fault
- 2 = illegal instruction
- 3 = breakpoint (ebreak)
- 4 = load address misaligned
- 5 = load access fault
- ...and so on

* Exceptions always trap up to a higher privilege level.

Ensuring Precise Exceptions in Pipelining

- Idea: Make each operation take the same amount of time

FMUL R3 ← R1, R2	F	D	E	E	E	E	E	E	E	E	W				
ADD R4 ← R1, R2		F	D	E	E	E	E	E	E	E	E	W			
			F	D	E	E	E	E	E	E	E	E	W		
				F	D	E	E	E	E	E	E	E	E	W	
					F	D	E	E	E	E	E	E	E	E	W
						F	D	E	E	E	E	E	E	E	E	W
							F	D	E	E	E	E	E	E	E	E	W

כעת אין בעיה של תזמון אבל:

- Downside
  - Worst-case instruction latency determines all instructions' latency
  - What about memory operations?
  - Each functional unit takes worst-case number of cycles?

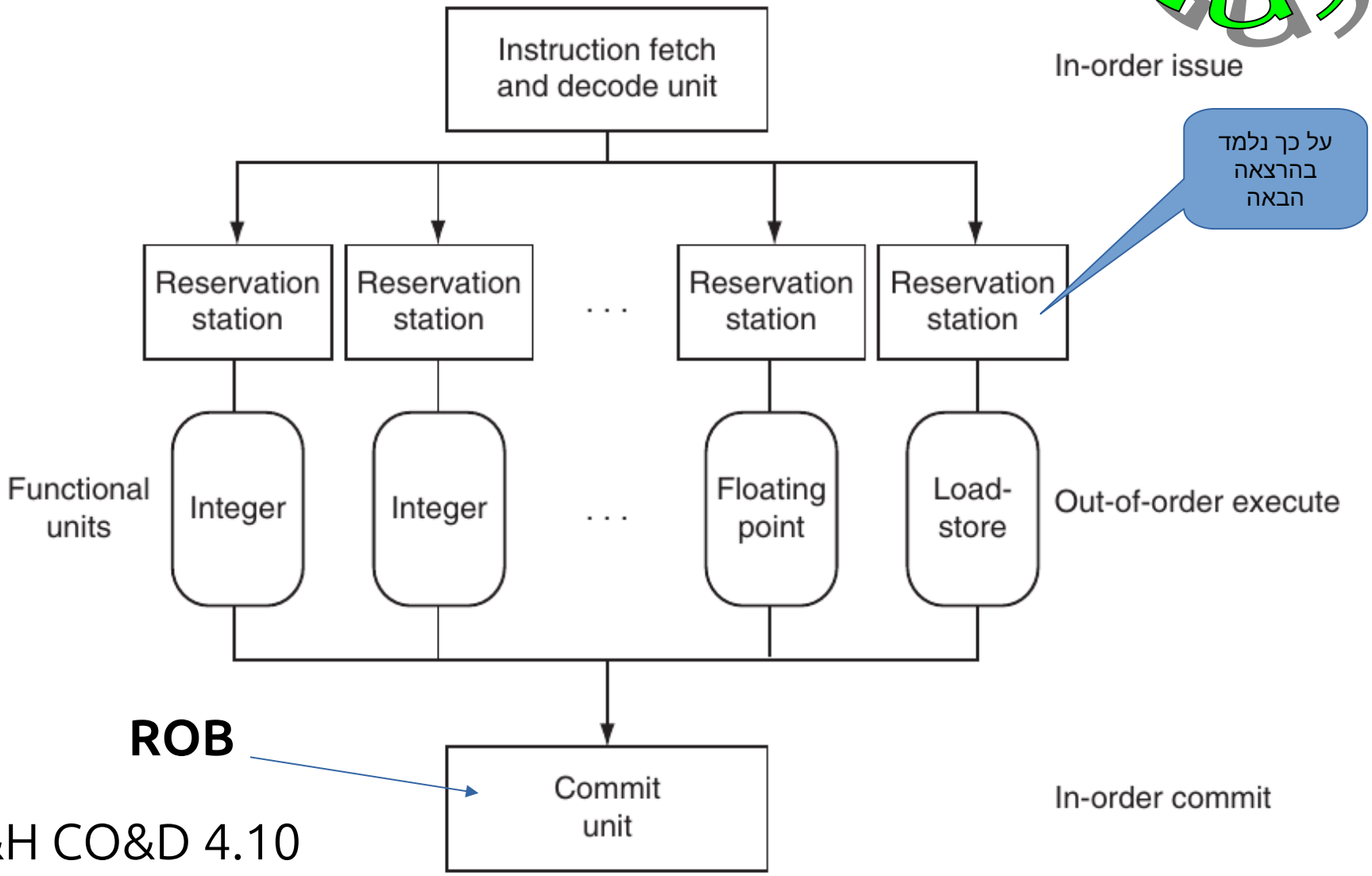
Solutions

How do we support precise exceptions in the presence of instructions completing out of program order?

- Reorder buffer
- History buffer
- Future register file
- Checkpointing

בשלושת אלה נדון אך לא לעומק

- Readings
  - Smith and Plezskun, "Implementing Precise Interrupts in Pipelined Processors," IEEE Trans on Computers 1988 and ISCA 1985.
  - Hwu and Patt, "Checkpoint Repair for Out-of-order Execution Machines," ISCA 1987.



P&H CO&D 4.10

FIGURE 4.72 The three primary units of a dynamically scheduled pipeline. The final step of updating the state is also called retirement or graduation.

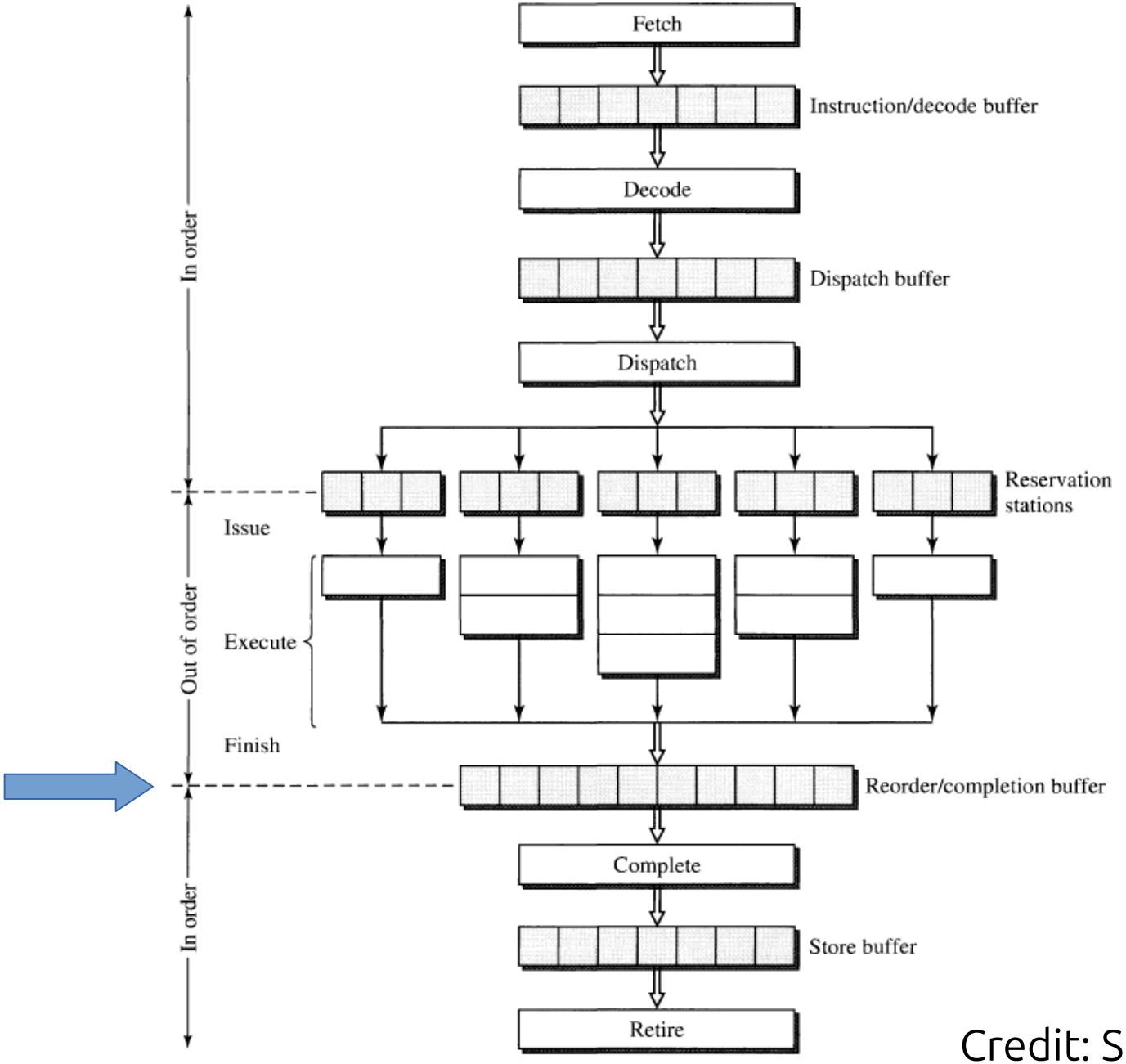
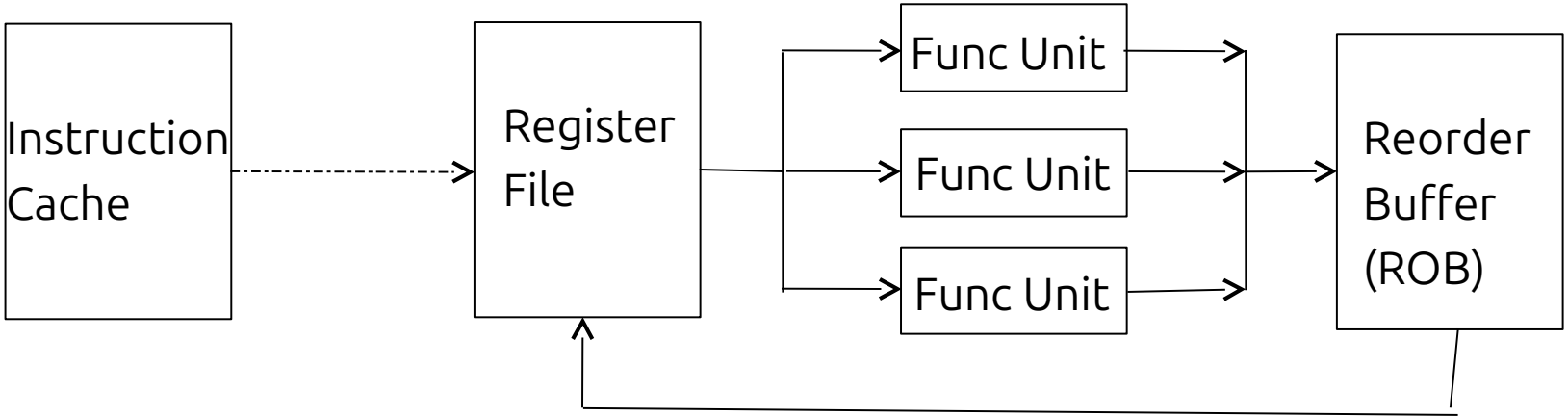


Figure 4.20
A Dynamic Pipeline with Reservation Station and Reorder Buffer.

Credit: Shen & Lipasti
Mod.Proc.Des.

Solution I: Reorder Buffer (ROB)

- Idea: Complete instructions out-of-order, but reorder them before making results visible to architectural state
- When instruction is **decoded** it reserves an entry in the ROB
- When instruction **completes**, it writes result into ROB entry
- When instruction **oldest in ROB** and it has completed without exceptions, its result moved to reg. file or memory



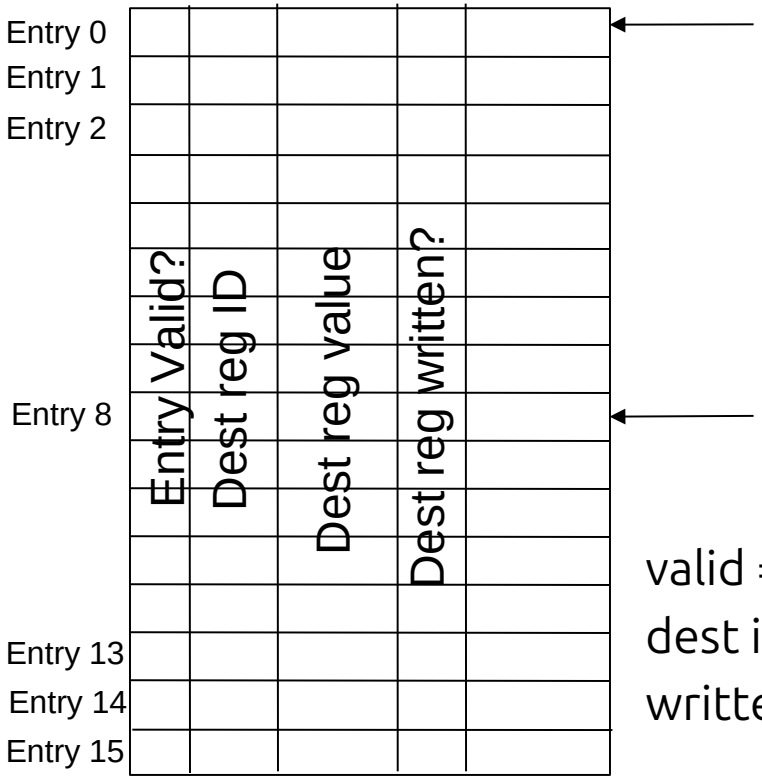
Reorder Buffer

- Buffers information about all instructions that are **decoded** but **not yet retired**/committed

A re-order buffer is a hardware unit [used in an extension to the **Tomasulo algorithm** – *next lecture*] to support out-of-order and speculative instruction execution. The extension forces instructions to be committed in-order.

Reorder Buffer

- A hardware structure that keeps information about all instructions that are **decoded** but **not yet retired**/committed



Oldest instruction (pointer to ROB entry that contains information about oldest instruction in the machine)

Youngest instruction

valid = נעשתה הקצאה להוראה

dest id / value = מה השם / הערך

written? = הערך כבר נכתב או לא?

ROB is implemented as a circular queue in hardware

What's in a ROB Entry? תכולה

V	DestRegID	DestRegVal	StoreAddr	StoreData	PC	Valid bits for reg/data + control bits	Exception?

מצביע

רגיסטר

זיכרון
(לא נדון בכך)

סטטוס ביצוע

חריגה?

- Everything required to:
  - correctly reorder instructions back into the program order
  - update the architectural state with the instruction's result(s), if instruction can retire without any issues
  - handle an exception/interrupt precisely, if an exception/interrupt needs to be handled before retiring the instruction
- Need **valid bits** to keep track of readiness of the result(s) and find out if the instruction has completed execution

Busy	Issued	Finished	Instruction address	Rename register	Speculative	Valid

(a)

Next entry to be allocated (tail pointer)

Next instruction to complete (head pointer)

לכן תכנס ההוראה הכי צעירה

ההוראה הכי ותיקה

אם נסובב את הטבלה ב-90 מעלות בכיוון השעון, היא תהיה תואמת לחלק העליון

B	0	0	0	0	0	1	1	1	1	1	1	1
I												
F												
IA												
RR												
S												
V												

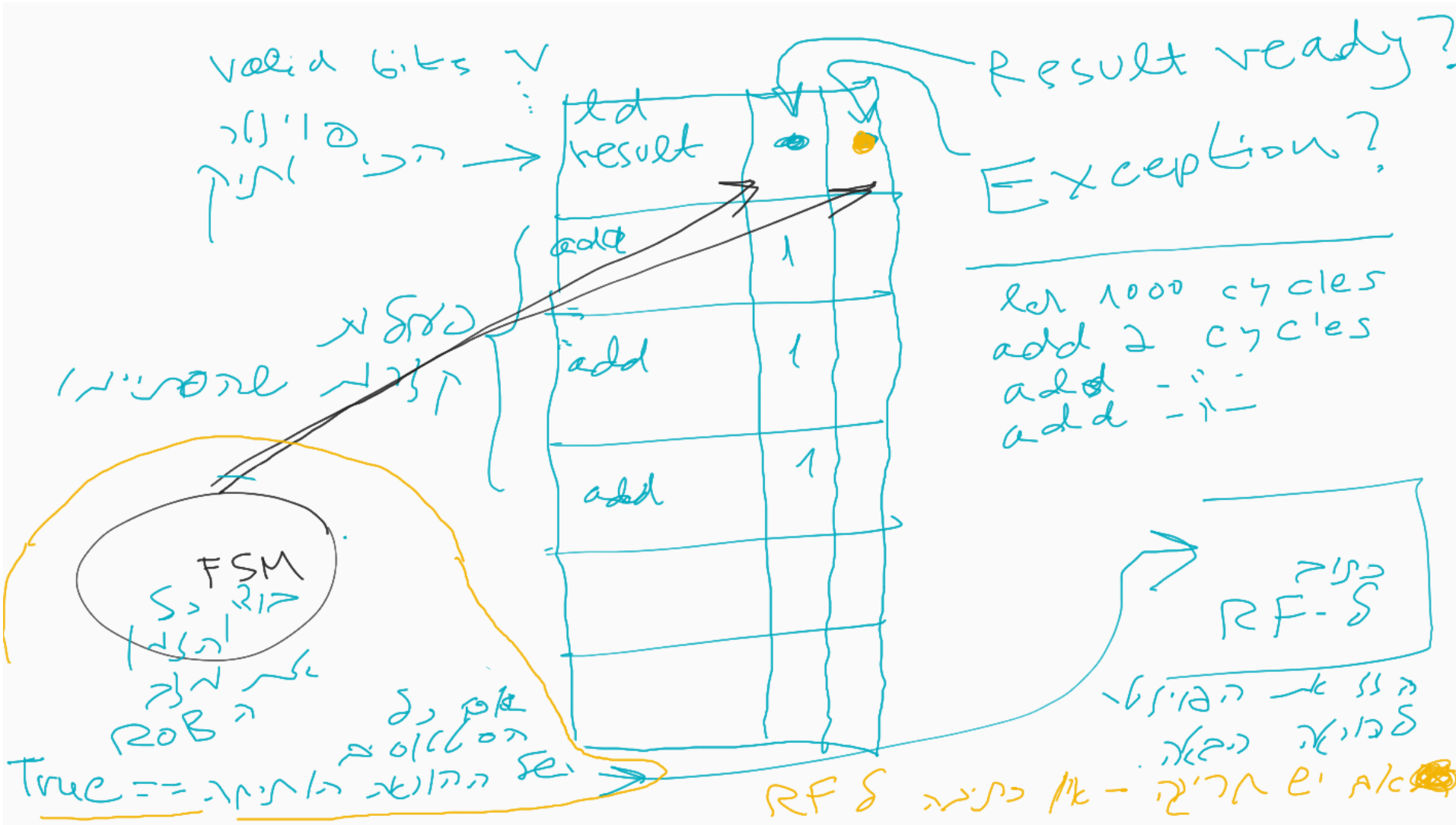
Reorder buffer

(b)

Figure 5.28

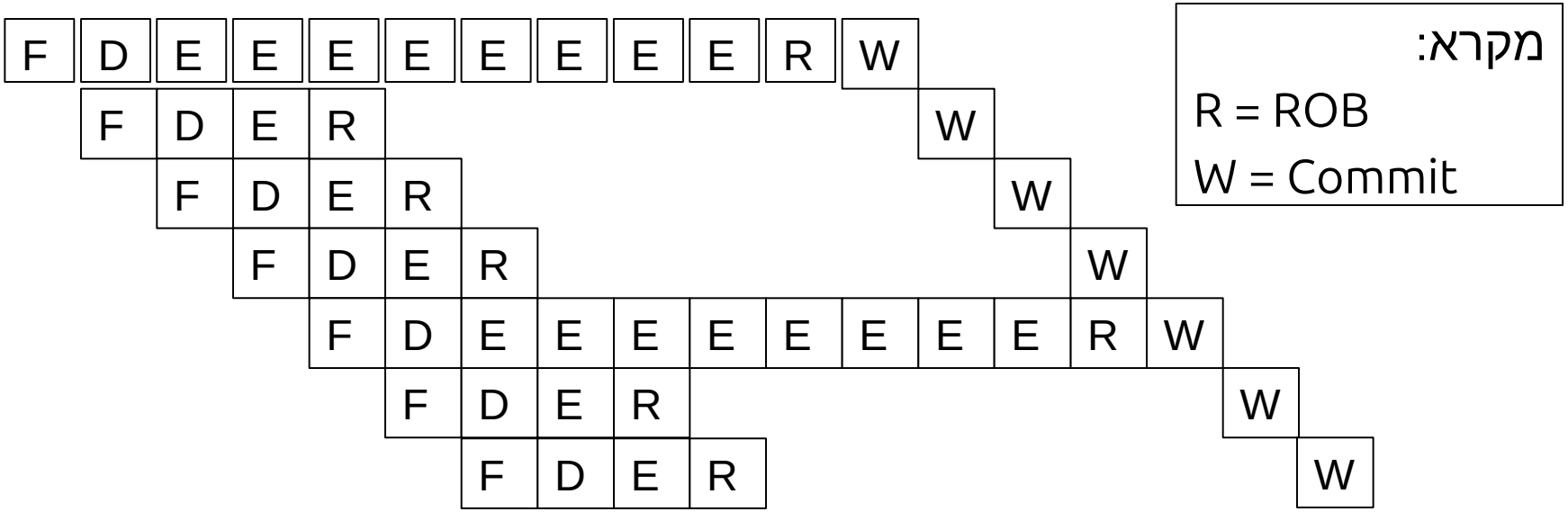
(a) Reorder Buffer Entry; (b) Reorder Buffer Organization.

Credit: Shen & Lipasti Mod.Proc.Des.


Guy
valid bits V
סוגר
היכן אתה
ld
result
add
add
add
1
1
1
Result ready?
Exception?
לכל 1000 cycles
add 2 cycles
add
add
בעקבות
קבלת שהסתיים
FSM
כתיבה כל
הנתון
אל מקום
ה ROB
אם כל
הסטטוסים
של הוראה הוא
True ==
כתיבה
RF-ל
הגע אל הבוינט
לבוינט הבאה
אבל יש חריגה - אין כתיבה RF ל

Reorder Buffer: Independent Operations

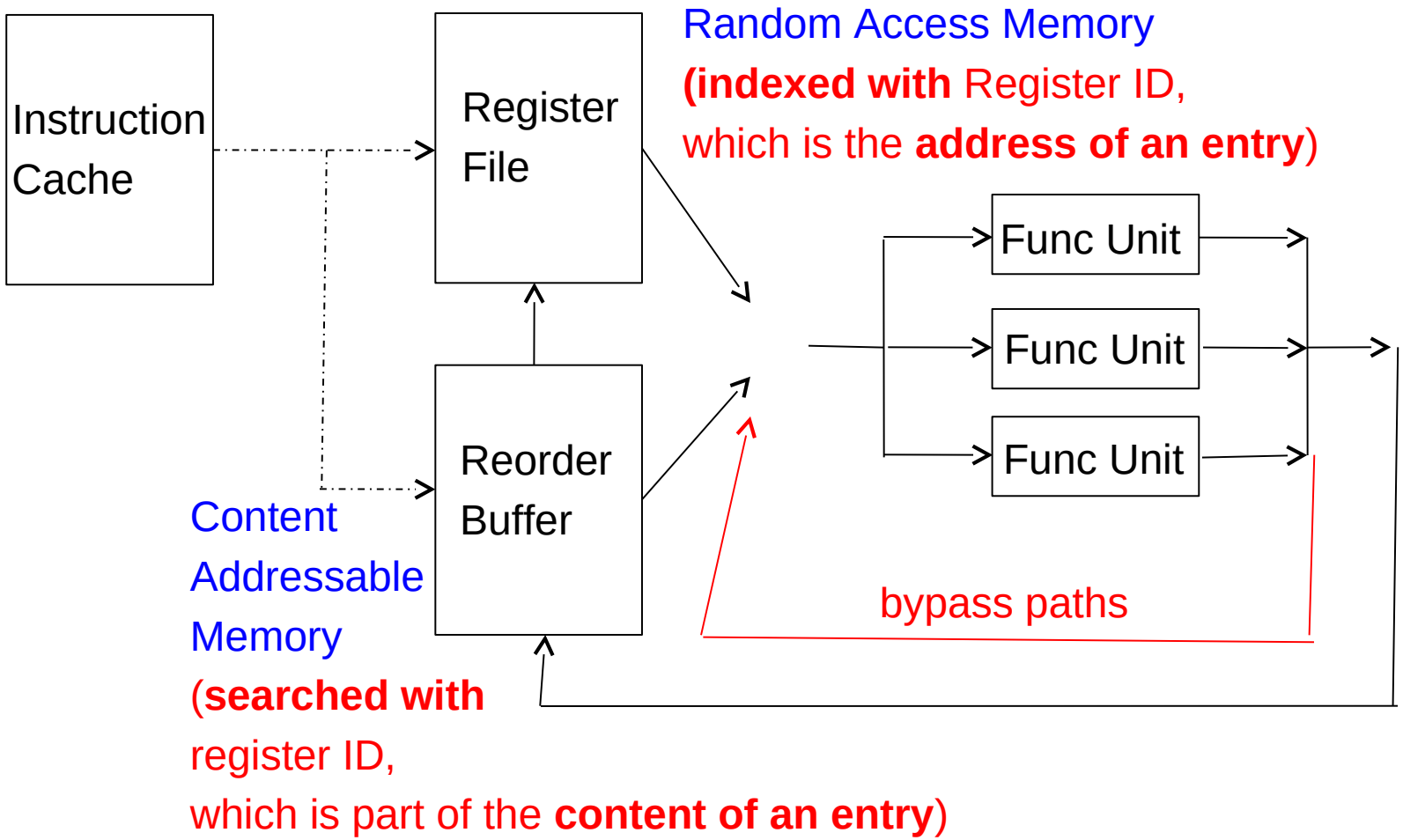
- Result first written to ROB on instruction completion
- Result written to register file at commit time



- What if a later instruction needs a value in the reorder buffer?
  - One option: stall the operation → stall the pipeline
  - Better: Read the value from the reorder buffer. How?

Reorder Buffer: How to Access?

- A register value can be in the register file, reorder buffer, (or bypass/forwarding paths)



Reorder Buffer Example

Register File (RF)

	Value Valid?	Value
R0		
R1		
R2		
R3		
R4		
R5		
R6		
R7		

Reorder Buffer (ROB)

	Entry Valid?	Dest reg ID	Dest reg value	Dest reg written?	
Entry 0					
Entry 1					
Entry 2					
Entry 8					
Entry 13					
Entry 14					
Entry 15					

Oldest instruction

Youngest instruction

Initially: **all registers are valid in RF & ROB is empty**

Simulate:

MUL R1, R2 → R3
MUL R3, R4 → R11
ADD R5, R6 → R3
ADD R3, R8 → R12

המצב הראשוני

Reorder Buffer Example

Register File (RF)

	Value Valid?	Value
R0	1	
R1	1	
R2	1	
R3	1	
R4	1	
R5	1	
R6	1	
R7	1	

Reorder Buffer (ROB)

	Entry Valid?	Dest reg ID	Dest reg value	Dest reg written?	
Entry 0	0				
Entry 1	0				
Entry 2	0				
	0				
	0				
	0				
	0				
Entry 8					
Entry 13					
Entry 14					
Entry 15	0				

Oldest instruction

Youngest instruction

Initially: **all registers are valid in RF & ROB is empty**

Simulate:

MUL R1, R2 → R3
MUL R3, R4 → R11
ADD R5, R6 → R3
ADD R3, R8 → R12

decode להוראה הראשונה

Reorder Buffer Example

Register File (RF)

	Value Valid?	Value
R0	1	
R1	1	1
R2	1	2
R3	0	
R4	1	
R5	1	
R6	1	
R7	1	

Reorder Buffer (ROB)

	Entry Valid?	Dest. reg ID	Dest reg value	Reg written?
Entry 0	1	R3		0
Entry 1	0			
Entry 2	0			
	0			
	0			
	0			
	0			
Entry 8				
Entry 13				
Entry 14				
Entry 15	0			

Oldest instruction

Youngest instruction

Initially: **all registers are valid in RF & ROB is empty**

Simulate:
MUL R1, R2 → R3
MUL R3, R4 → R11
ADD R5, R6 → R3

decode להוראה השניה

Reorder Buffer Example

Register File (RF)

	Valid?	Value
R0	1	
R1	1	1
R2	1	2
R3	0	
R4	1	4
R5	1	
R6	1	
R7	1	

R11 | 0 | 1

Reorder Buffer (ROB)

	Entry Valid?	Dest reg ID	Dest reg value	Dest reg written?
Entry 0	1	R3		0
Entry 1	1	R11		0
Entry 2	0			
	0			
	0			
	0			
	0			
Entry 8				
Entry 13				
Entry 14				
Entry 15	0			

Oldest instruction

Youngest instruction

Initially: **all registers are valid in RF & ROB is empty**

Simulate:

MUL R1, R2 → R3

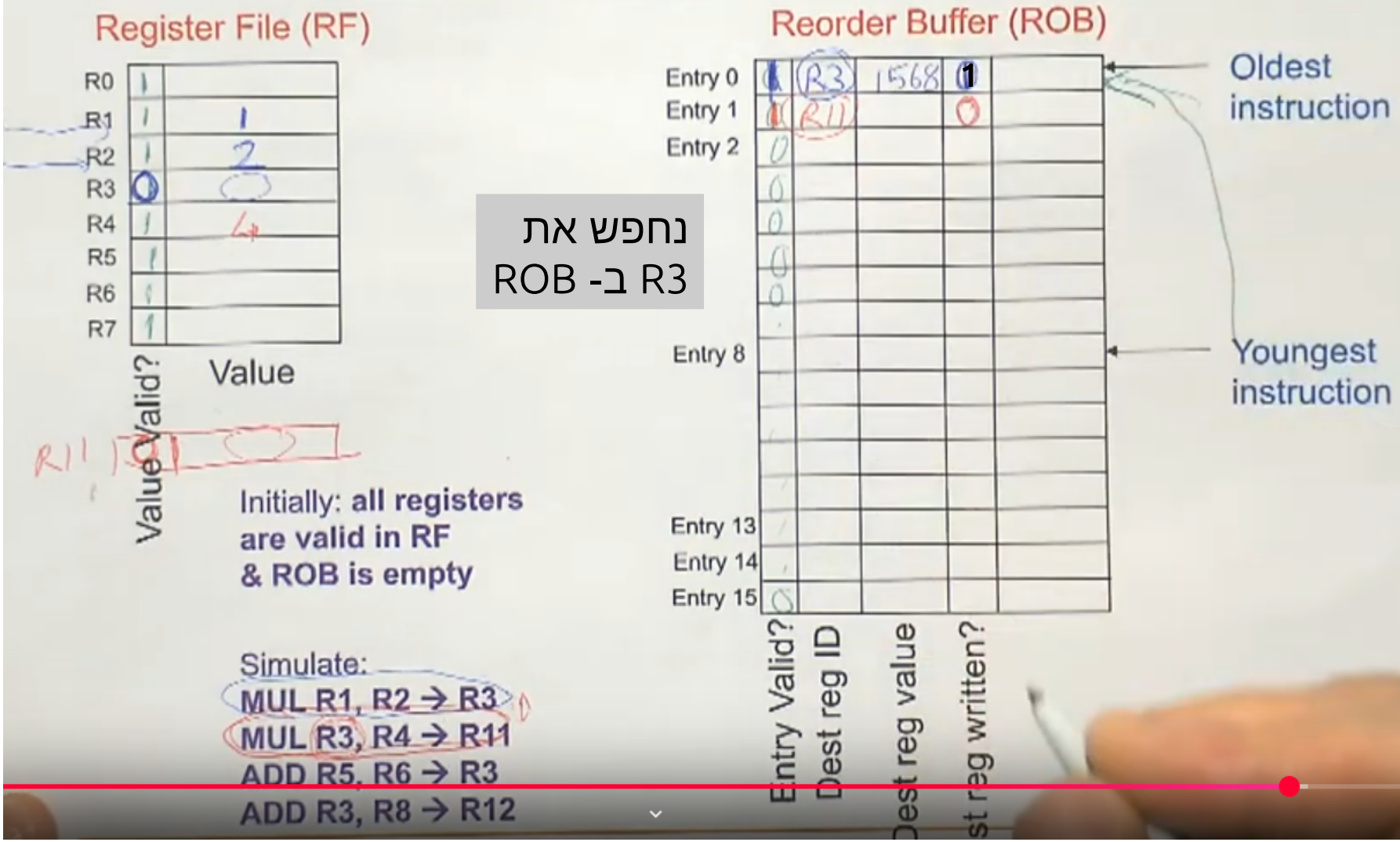
MUL R3, R4 → R11

ADD R5, R6 → R3

ADD R3, R8 → R12

עתה נניח שההוראה הראשונה סיימה ויש ערך ל-R3

Reorder Buffer Example



נחפש את R3 ב- ROB

decode להוראה השלישית

Reorder Buffer Example

Register File (RF)

	Valid?	Value
R0	1	
R1	1	1
R2	1	2
R3	0	
R4	1	4
R5	1	5
R6	1	6
R7	1	

Reorder Buffer (ROB)

	Entry Valid?	Dest reg ID	Dest reg value	Dest reg written?
Entry 0	1	R3	1568	1
Entry 1	1	R11		0
Entry 2	1	R3		0
Entry 8				
Entry 13				
Entry 14				
Entry 15	0			

Oldest instruction

Youngest instruction

Initially: **all registers are valid in RF & ROB is empty**

Simulate:

MUL R1, R2 → R3

MUL R3, R4 → R11

ADD R5, R6 → R3

ADD R3, R8 → R12

WAW dependence on R3 which is a false dependence

decode להוראה הרביעית

Reorder Buffer Example

Register File (RF)

	Value Valid?	Value
R0	1	
R1	1	1
R2	1	2
R3	0	
R4	1	4
R5	1	5
R6	1	6
R7	1	
R11	1	
R12	1	

Initially: all registe
are valid in RF
& ROB is en

Simulate:
MUL R1, R2 → R3
MUL R3, R4 → R11
ADD R5, R6 → R3
ADD R3, R8 → R12

Reorder Buffer (ROB)

	Entry Valid?	Dest reg ID	Dest reg value	Dest reg written?
Entry 0	1	R3	1568	1
Entry 1	1	R11		0
Entry 2	1	R3		
	1	R7		
	0			
	0			
Entry 13				
Entry 14				
Entry 15	0			

Oldest instruction

Youngest instruction

R3 = ?

פעולת החיפוש ב-ROB "יקרה"

יש לחפש את R3 ב-ROB בהתאם למודל ה- CAM (פעולות השוואה חוזרות) ולהשתמש בערך האחרון שלו שלפני ההוראה הנדונה לכשיהיה זמין כלומר הדגל ישתנה ל-1

RAM vs. CAM (Guy)

RAM (Random Access Memory) is used for storing data and programs that the CPU needs in real-time, while CAM (Content Addressable Memory) allows for high-speed **data retrieval based on content rather than specific addresses**. CAM is particularly useful in applications requiring rapid searches

Difference between Random Access Memory (RAM) and Content Addressable Memory (CAM)

Last Updated : 13 May, 2019

RAM:

Random Access Memory (RAM) is used to read and write. It is the part of primary memory and used in order to store running applications (programs) and program's data for performing operation. It is mainly of two types: Dynamic RAM (or DRAM) and Static RAM (or SRAM).

CAM:

[redacted]emory, in which the user supplies data word and associative memory searches its entire memory and if the data word is found, It returns the list of addresses where that data word was located.

The difference table is given below on the basis of their properties:

Credit:
https://www.geeksforgeeks.org/difference-between-random-access-memory-ram-and-content-addressable-memory-cam/

Simplifying Reorder Buffer Access

- **Idea: Replace CAM with indirection**

  התרשים הקודם מסורבל: הארכת הנתיב הקריטי, הוספת לוגיקה והגדלת כמות הרכיבים

- **– הצעה לפתרון החיפוש המסורבל ב-CAM**

- Access register file first תחילה ניגשים לרגיסטר פייל
  - If register not valid, register file stores the ID of the reorder buffer entry that contains (or will contain) the value of the register
  - Mapping of the register to a ROB entry: Register file maps the register to a reorder buffer entry if there is an in-flight instruction writing to the register

- Access reorder buffer next ולאחר מכן ניגשים לבאפר

מיפוי שכזה ישמש אותנו גם בהמשך כשנלמד על OOO (ההרצאה הבאה)

Reorder Buffer Example

Register File (RF)

	Value Valid?	Value	Tag (pointer to ROB entry)
R0			
R1			
R2			
R3			
R4			
R5			
R6			
R7			

Reorder Buffer (ROB)

	Entry Valid?	Dest reg ID	Dest reg value	Dest reg written?	
Entry 0					
Entry 1					
Entry 2					
Entry 8					
Entry 13					
Entry 14					
Entry 15					

Oldest instruction

Youngest instruction

Initially: **all registers are valid in RF & ROB is empty**

Simulate:

MUL R1, R2 → R3
MUL R3, R4 → R11
ADD R5, R6 → R3
ADD R3, R8 → R12

Decode להוראה הראשונה

Reorder Buffer Example

Register File (RF)

	Value Valid?	Value	Tag (pointer to ROB entry)
R0	1		
R1	1	1	
R2	1	2	
R3	0		ROB0
R4	1		
R5	1		
R6	1		
R7	1		

Reorder Buffer (ROB)

	Entry Valid?	Dest reg ID	Dest reg value	Dest reg written?
Entry 0	1	R3		0
Entry 1	0			
Entry 2	0			
	0			
	0			
	1			
Entry 8				
	1			
	1			
	1			
Entry 13	1			
Entry 14	1			
Entry 15	1			

Oldest instruction

Youngest instruction

Initially: **all registers are valid in RF & ROB is empty**

Simulate:

MUL R1, R2 → R3
MUL R3, R4 → R11
ADD R5, R6 → R3
ADD R3, R8 → R12

Decode להוראה השלישית

Reorder Buffer Example

Register File (RF)

	Value Valid?	Value	Tag (pointer to ROB entry)
R0	1		
R1	1	1	
R2	1	2	
R3	0	3	ROB2
R4	1	4	
R5	1	5	
R6	1	6	
R7	1		

Reorder Buffer (ROB)

	Entry Valid?	Dest reg ID	Dest reg value	Dest reg written?
Entry 0	1	R3		0
Entry 1	1	R11		0
Entry 2	1	R3		0
Entry 3	0			
Entry 4	0			
Entry 5	1			
Entry 6	1			
Entry 7	1			
Entry 8	1			
Entry 9	1			
Entry 10	1			
Entry 11	1			
Entry 12	1			
Entry 13	1			
Entry 14	1			
Entry 15	1			

Oldest instruction

Youngest instruction

R11 0 ROB1

Initially: all registers are valid in RF & ROB is empty

Simulate:

MUL R1, R2 → R3
MUL R3, R4 → R11
ADD R5, R6 → R3
ADD R3, R8 → R12

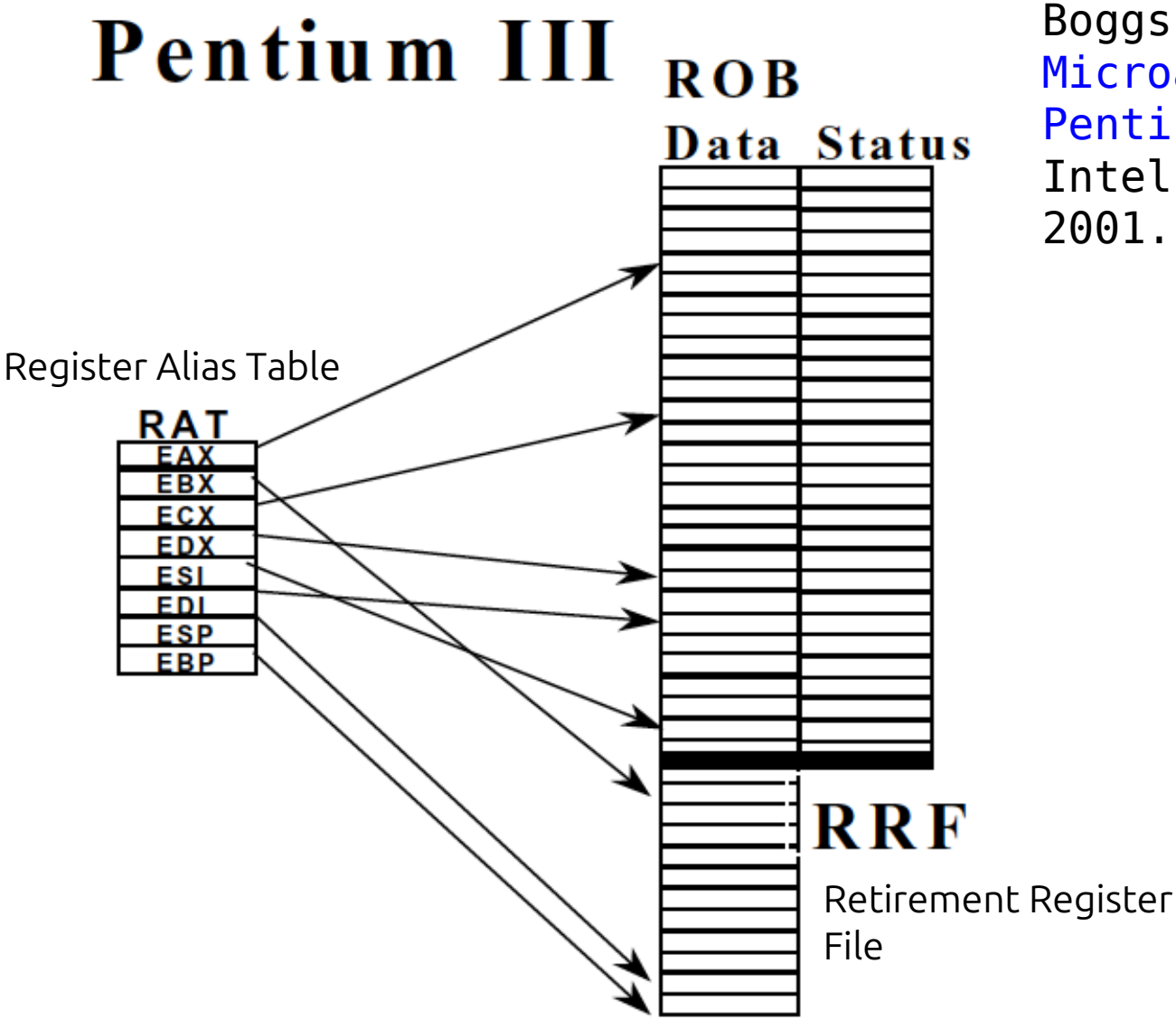
בהנחה
שה-decode של
הוראה 3 נעשה
בטרם ההוראה
הראשונה סיימה

ב- RF כמות הרגיסטרים מוגבלת בעוד שב- ROB אפשר לשמור הרבה נתונים וכל נתון יכול להיות מיפוי שונה של רגיסטר

Architectural registers → Physical registers (ROB or another structure…)

Size of ROB in Pentium 4: 128 entries

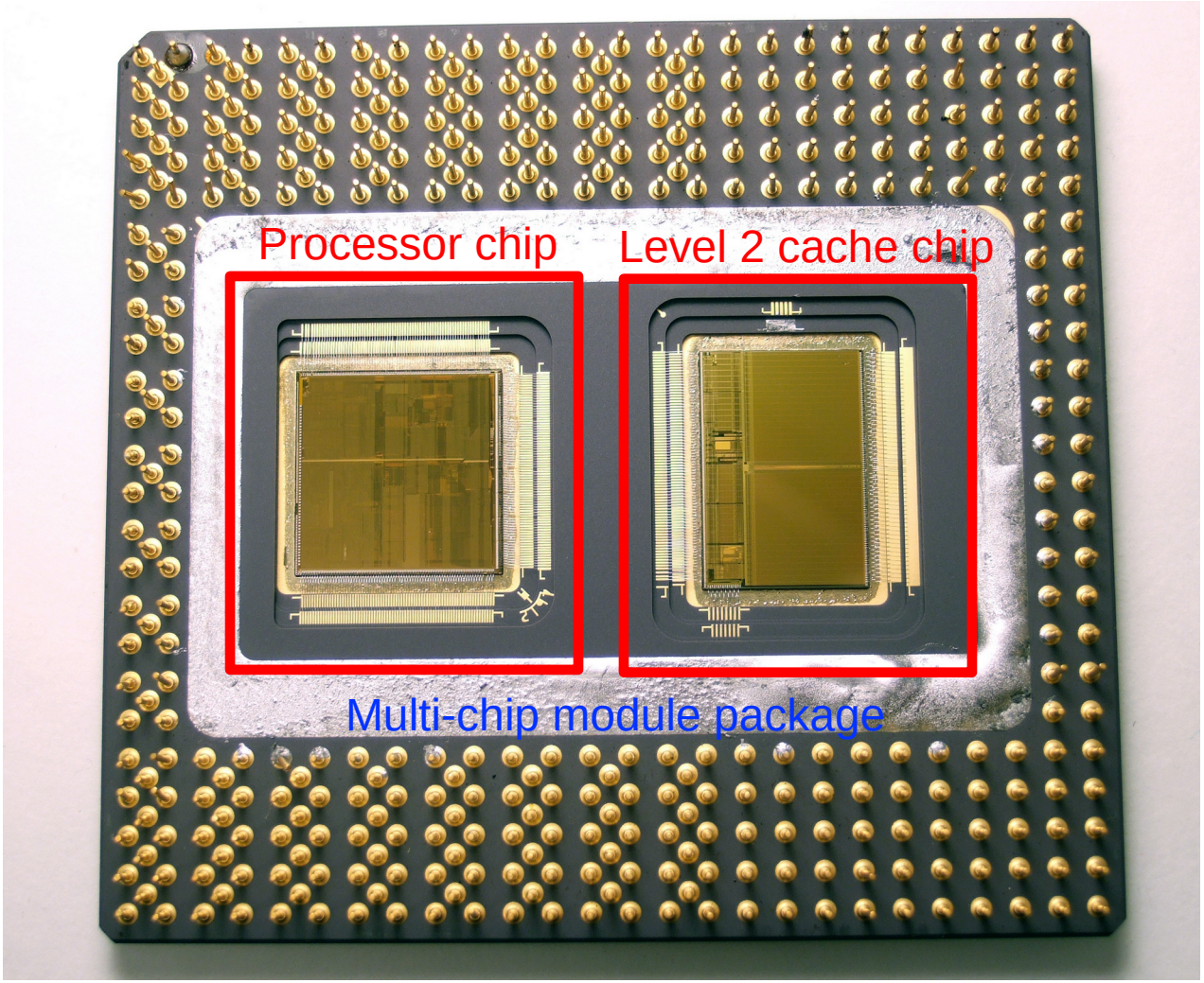
Reorder Buffer in Intel Pentium III



Boggs et al., "The Microarchitecture of the Pentium 4 Processor," Intel Technology Journal, 2001.

רגיסטרים מוכנים

Intel Pentium Pro (1995)



By Moshen - http://en.wikipedia.org/wiki/Image:Pentiumpro_moshen.jpg, CC BY-SA 2.5, https://commons.wikimedia.org/w/index.php?curid=2262471

Important: Register Renaming with a Reorder Buffer

- Output and anti dependencies are not true dependencies
  - WHY? The same register refers to values that have nothing to do with each other
  - **They exist due to lack of register ID's (i.e. names) in the ISA**
- The register ID is **renamed** to the reorder buffer entry that will hold the register's value
  - Register ID → ROB entry ID (a larger name space!)
  - Architectural register ID → Physical register ID
  - After renaming, ROB entry ID used to refer to the register
- **This eliminates anti- and output- dependences**
  - Gives the illusion that there are a large number of registers

בהרצאה הבאה נקשור את הרעיונות שנוצרו כאן להרצה שלא עפ"י הסדר OOO

Recall: Data Dependence Types

True (flow) dependence

$r_3 \leftarrow r_1 \text{ op } r_2$

$r_5 \leftarrow r_3 \text{ op } r_4$

Read-after-Write (RAW) -- **True**

Anti dependence

$r_3 \leftarrow r_1 \text{ op } r_2$

$r_1 \leftarrow r_4 \text{ op } r_5$

Write-after-Read (WAR) -- **Anti**

Output-dependence

$r_3 \leftarrow r_1 \text{ op } r_2$

$r_5 \leftarrow r_3 \text{ op } r_4$

$r_3 \leftarrow r_6 \text{ op } r_7$

Write-after-Write (WAW) -- **Output**

Register renaming example תרגיל עבורכם

- Assume
  - Register file has a pointer to the reorder buffer entry that contains or will contain the value, if the register is not valid
  - Reorder buffer works as described before

- Where is the latest definition of R3 for each instruction below in sequential order?

  LD R0(0) → R3

  LD R3, R1 → R10

  MUL R1, R2 → R3

  MUL R3, R4 → R11

  ADD R5, R6 → R3

  ADD R7, R8 → R12

Reorder Buffer Example

Register File (RF)

	Value Valid?	Value	Tag (pointer to ROB entry)
R0			
R1			
R2			
R3			
R4			
R5			
R6			
R7			

Reorder Buffer (ROB)

	Entry Valid?	Dest reg ID	Dest reg value	Dest reg written?	
Entry 0					
Entry 1					
Entry 2					
Entry 8					
Entry 13					
Entry 14					
Entry 15					

Oldest instruction

Youngest instruction

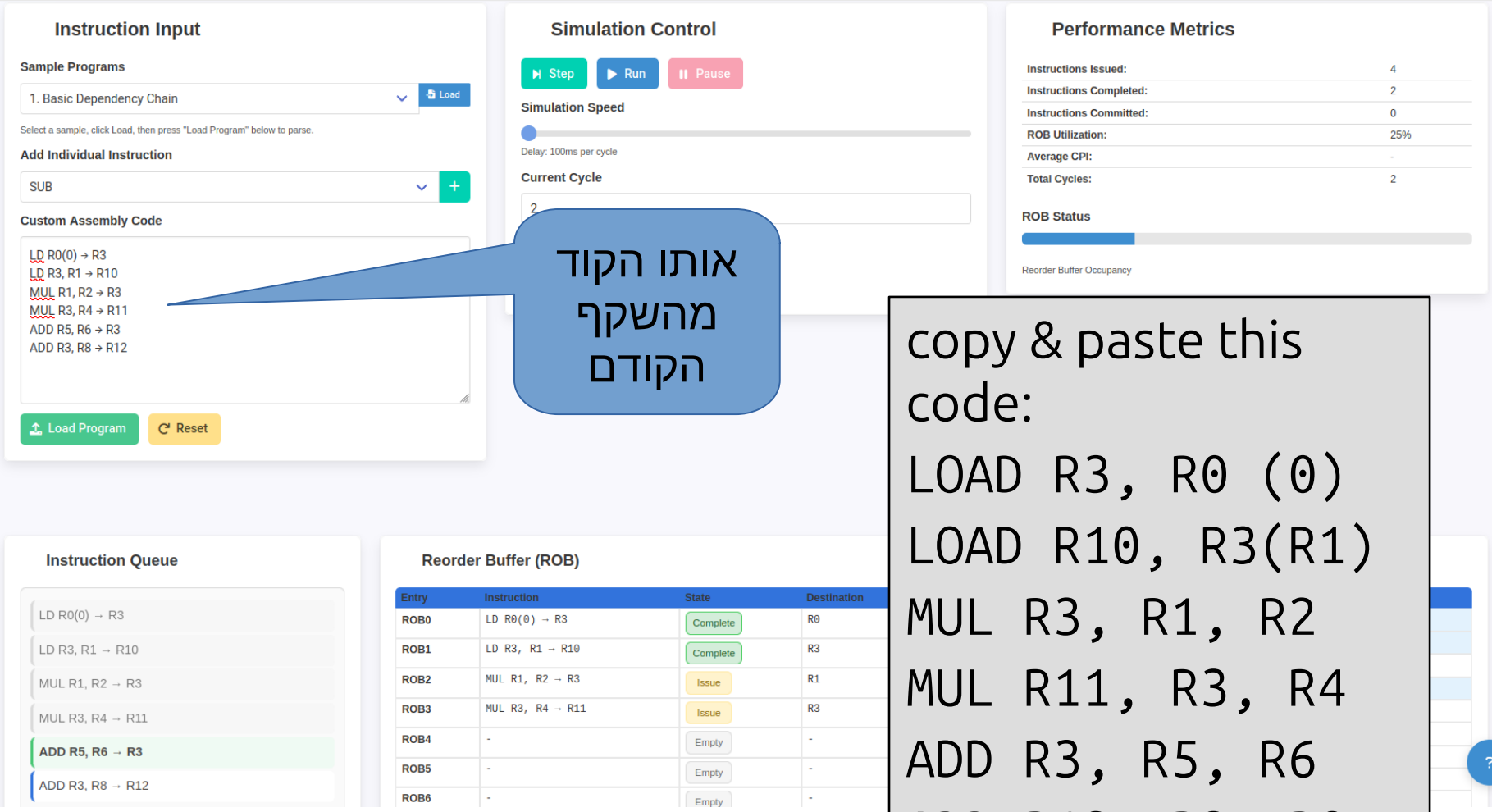
Initially: **all registers are valid in RF & ROB is empty**

Simulate:

LD R0(0) → R3
LD R3, R1 → R10
MUL R1, R2 → R3
MUL R3, R4 → R11
ADD R5, R6 → R3
ADD R3, R8 → R12

Reorder Buffer Simulation

https://aca-iiith.vlabs.ac.in/exp/reorder-buffers/simulation/index.html



אותו הקוד מהשקף הקודם

copy & paste this code:

```
LOAD R3, R0 (0)
LOAD R10, R3(R1)
MUL R3, R1, R2
MUL R11, R3, R4
ADD R3, R5, R6
ADD R12, R3, R8
```

Valid and Ready bits

Valid Bit

- Means: The instruction has finished executing and its result is ready
- Set to 1 when the execution unit writes the result back to the ROB entry
- Before this, the entry holds a placeholder (or the instruction is still executing)

Ready Bit

- Means: All operands needed by this instruction are available
- Set to 1 when all source registers have been resolved (either from the register file or forwarded from earlier instructions that completed)
- The instruction can only start executing once the ready bit is 1

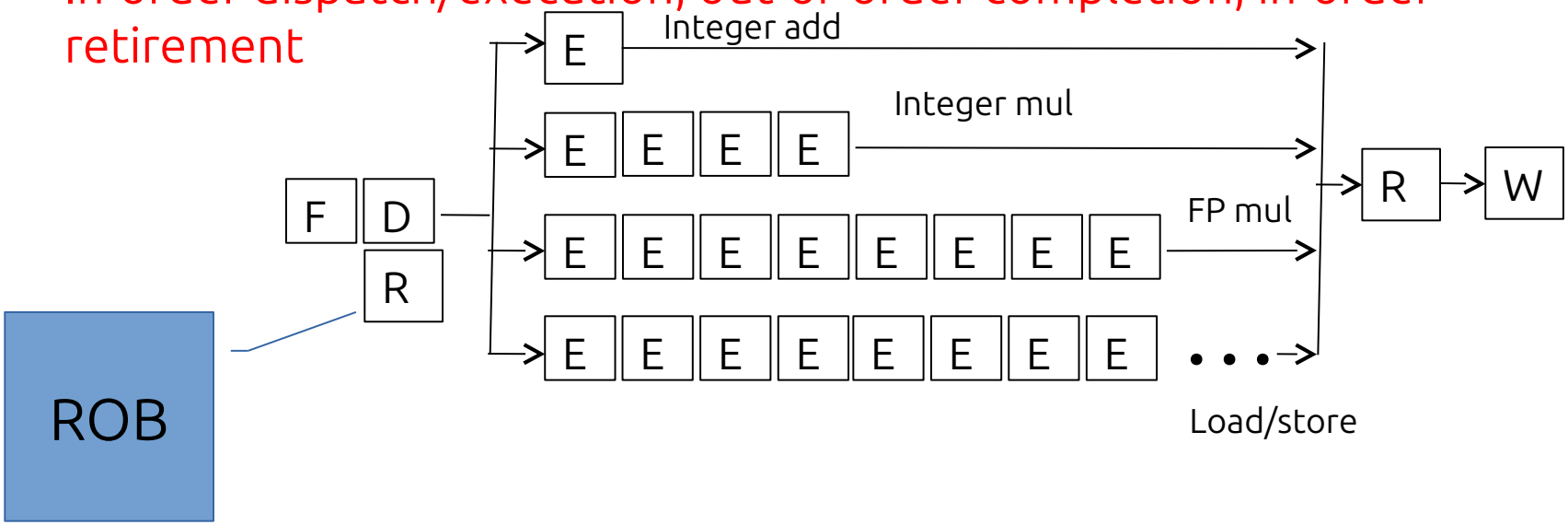
תרשים נוסף של תכולת ה- ROB

	ENTRY NUMBER	DEST. REG.	RESULT	EXCEP-TIONS	VALID	PROGRAM COUNTER
	3					
HEAD→	4	4			0	6
	5	0			0	7
TAIL →	6					
	⋮	⋮	⋮	⋮	⋮	⋮

REORDER BUFFER

In-Order Pipeline with Reorder Buffer

- Decode (D): Access regfile/ROB, allocate entry in ROB, check if instruction can execute, if so **dispatch** שילוח instruction
- Execute (E): Instructions can complete out-of-order
- Completion (R): Write result to reorder buffer
- Retirement/Commit (W): Check for exceptions; if none, write result to architectural register file or memory; else, flush pipeline and start from exception handler לסיכום
- In-order dispatch/execution, out-of-order completion, in-order retirement



New 2023

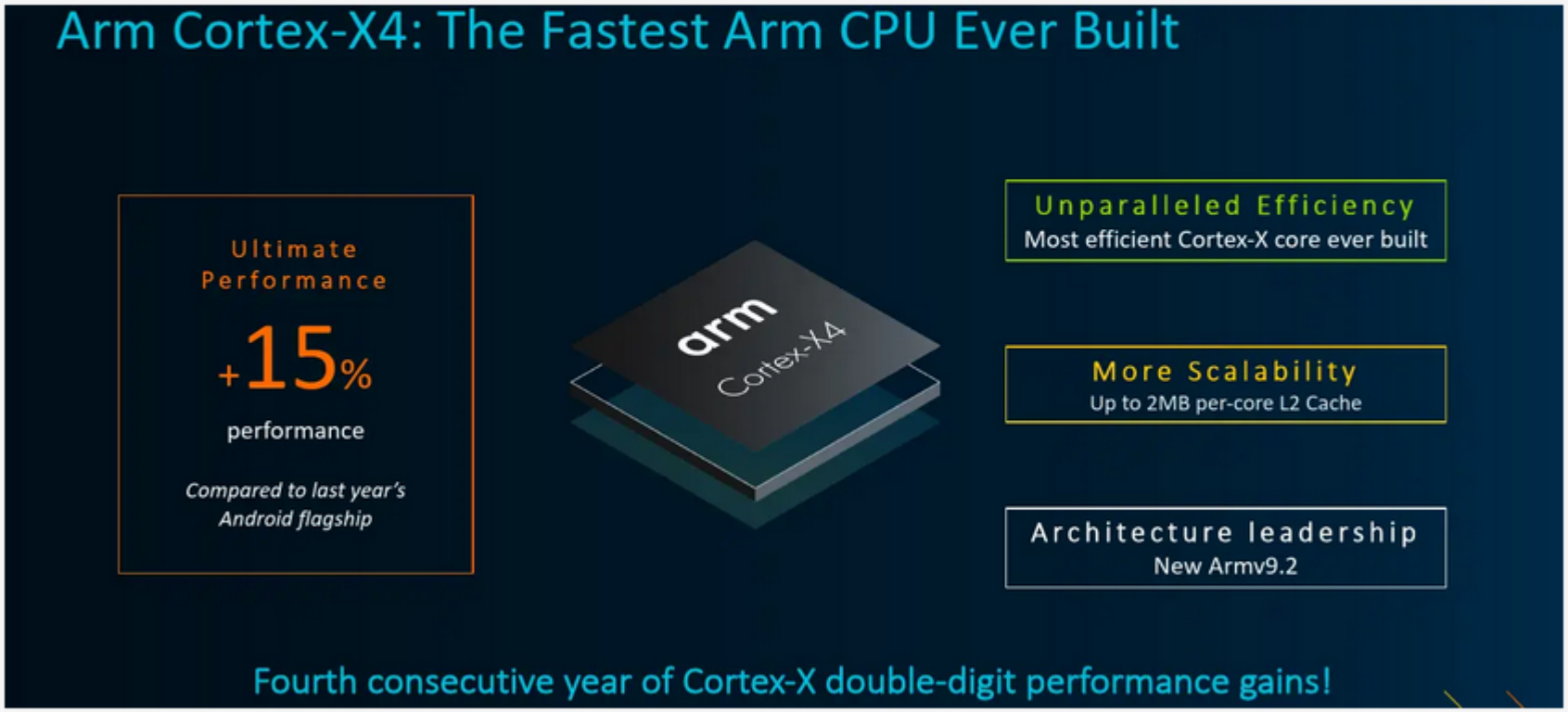
דוגמה

Credit: https://fuse.wikichip.org/news/7531/arm-introduces-the-cortex-x4-its-newest-flags

Architectures | Desktop Processors | Mobile Processors | Server Processors

Arm Introduces The Cortex-X4, Its Newest Flagship Performance Core

May 28, 2023 David Schor ARM, ARMv9, ARMv9.2, Cortex, Cortex-X, Cortex-X3, Cortex-X4



Today, Arm is launching the Cortex-X4, the company's next-generation flagship performance core and the highest-performance Arm core designed to date.

דוגמה

Credit:
https://fuse.wikichip.org/news/7531/arm-introduces-the-cortex-x4-its-newest-flagship-performance-core/
see also:
https://en.wikipedia.org/wiki/ARM_Cortex-X4

The out-of-order buffer on the Cortex-X4 was increased by 20% from 320 on the Cortex-X3 to 384 here. In fact, Arm has been increasing the ROB by anywhere from 10-30% each generation. To put that in perspective, the Cortex-X4 ROB is now bigger than Intel's Sunny Cove core which had 352 entries – albeit not nearly as much as Golden Cove with a whopping 512-entries ROB.

Reorder Buffer				
uArch	**Cortex-X1**	**Cortex-X2**	**Cortex-X3**	**Cortex-X4**
Dispatch	8/cycle	8/cycle	8/cycle	10/cycle
Max In-flight	224	288	320	384

On the floating-point side, Arm fully pipelined the divider/sqrt units. The pipes and units themselves remain unchanged.

Reorder Buffer Tradeoffs

- Advantages
  - Conceptually simple for supporting precise exceptions
  - Can eliminate false dependencies
- Disadvantages
  - Reorder buffer needs to be accessed to get the results that are yet to be written to the register file
    - CAM or indirection → increased latency and complexity
- Other solutions aim to eliminate the disadvantages
  - History buffer
  - Future file
  - Checkpointing

Guy: Wikipedia: Content-addressable memory (CAM) is a special type of computer memory used in certain very-high-speed searching applications. It is also known as associative memory or associative storage[a] and compares input search data (tag) against a table of stored data, and returns the address of matching data

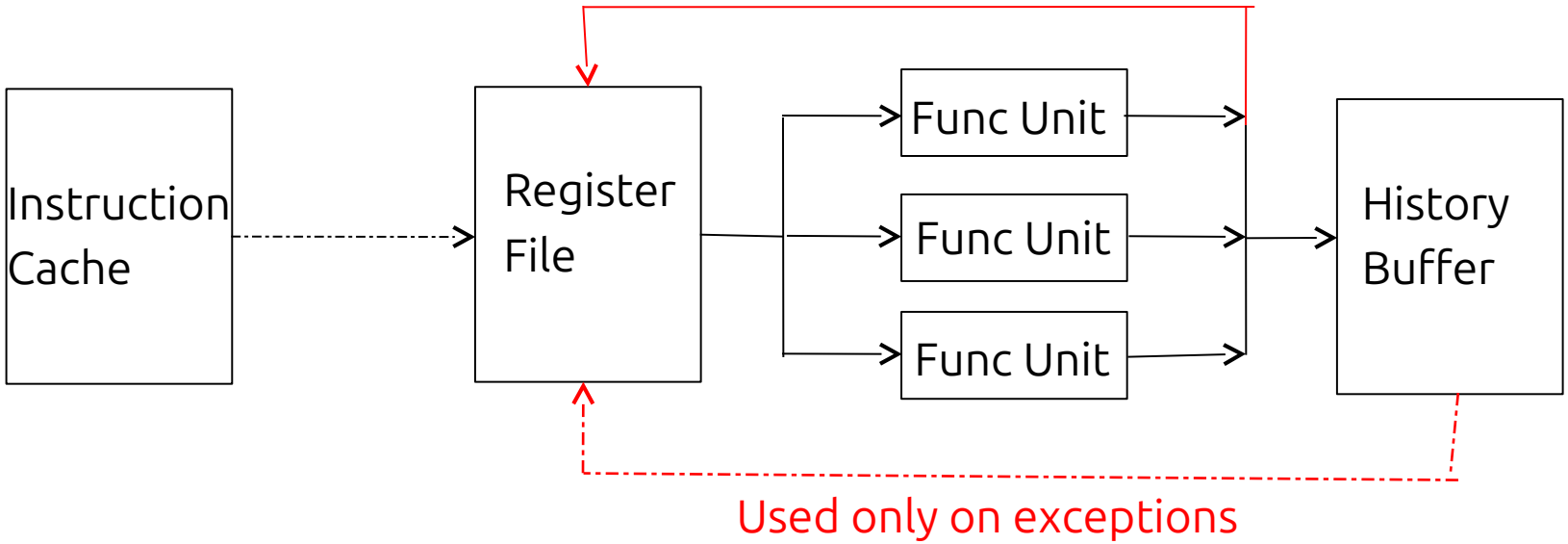
Solution II: History Buffer (HB)

- Idea: Update the register file when instruction completes, but UNDO UPDATES when an exception occurs

- When instruction is decoded, it reserves an HB entry
- When the instruction completes, it stores the old value of its destination in the HB
- When instruction is oldest and no exceptions/interrupts, the HB entry discarded
- When instruction is oldest and an exception needs to be handled, old values in the HB are written back into the architectural state from tail to head

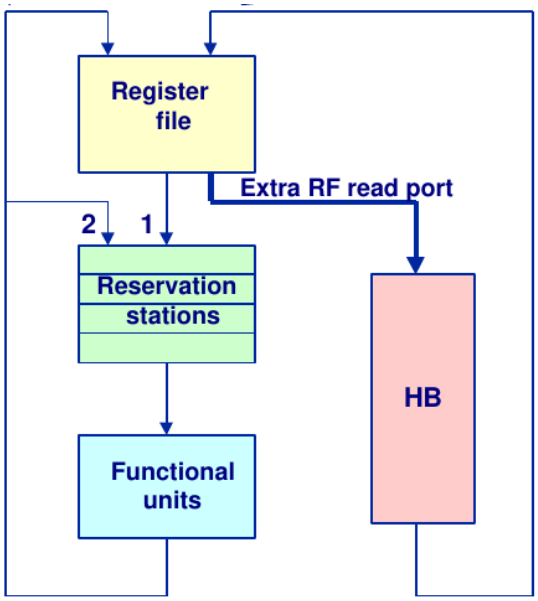
ROB הוא פסימי – חכה עד שההוראה תסתיים ולאחר מכן עדכן את RF.
HB הוא אופטימי – עדכן מייד את RF אך אם יש חריגה מחק את העדכונים.

History Buffer



- Advantage:
  - Register file contains up-to-date values for incoming instructions → History buffer access **not on critical path**
- Disadvantage:
  - Need to read the old value of the destination register
  - Need to unwind the history buffer upon an exception → increased exception/interrupt handling latency

הסבר נוסף על החסרונות



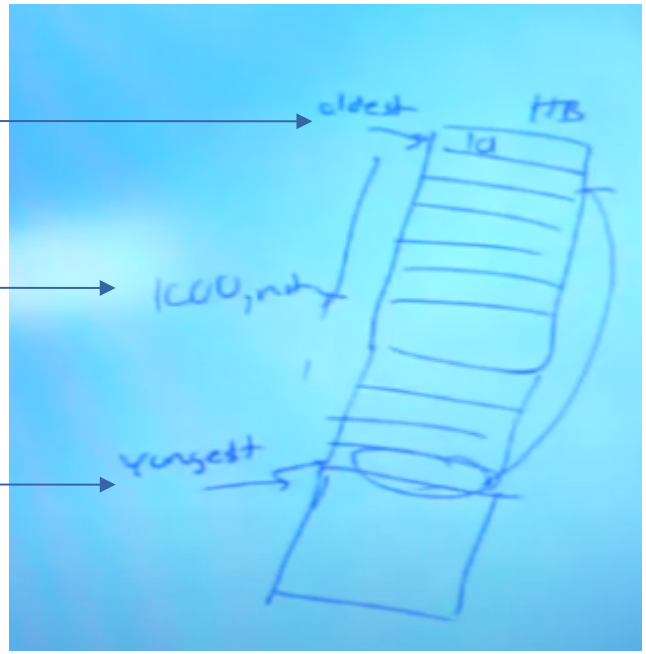
1) צורך בפורט נוסף בין ה-HB לבין ה- RF

2) במקרה של חריגה יש צורך בניקוי ה- HB. ראו **תרשים** (לכתוב חזרה לפי הסדר מלמטה למעלה)

המצביע הותיק

ביטול של 1000 הוראות לוקח זמן...

המצביע הצעיר



HB

Write back all the values

Comparison of Two Approaches

- Reorder buffer – מעדכן את הרגיסטר פייל מאוחר, עפ"י סדר התכנית
  - Pessimistic register file update
  - Update only with non-speculative values (in program order)
  - Leads to complexity/delay in accessing the new values

- History buffer – מעדכן את הרגיסטר פייל מיידית עם סיום הוראה
  - * Optimistic register file update
  - Update immediately, but log the old value for recovery
  - ** Leads to complexity/delay in logging old values (במקרה של recovery)

Can we get the best of both worlds?

להנות מ * מבלי לסבול מ- **

- Principle: Heterogeneity הפתרון – גם וגם
- Idea: **Have both types of register files**

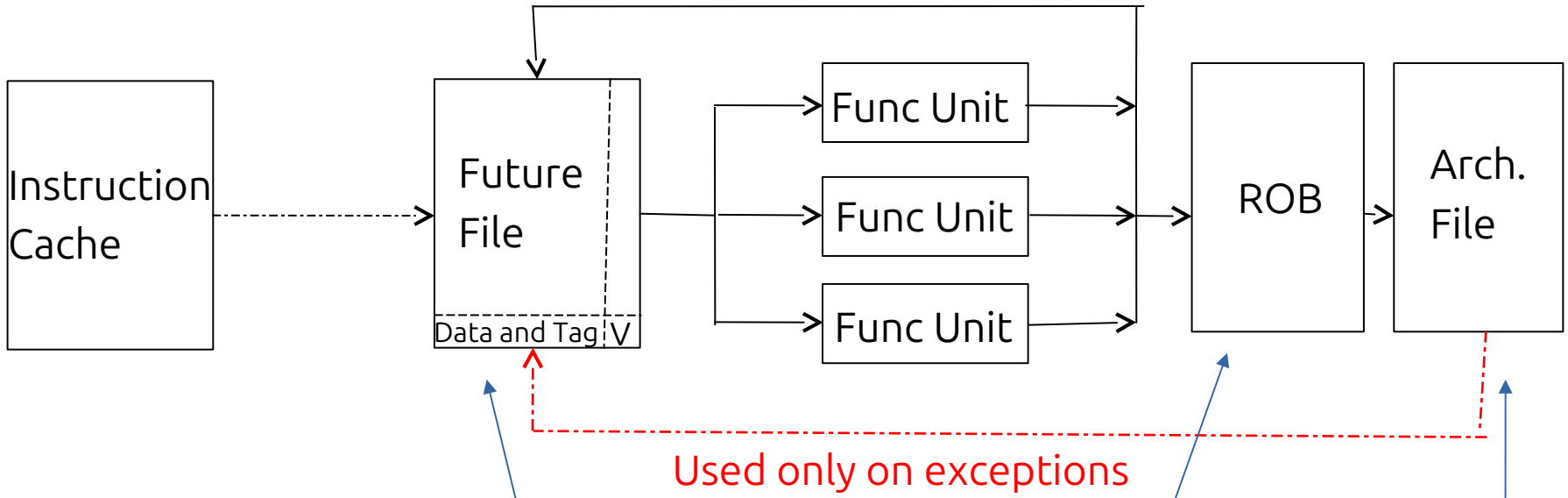
Solution III: Future File (FF) + ROB

- Idea: Keep **two register files** (speculative and architectural)
  - Arch reg file: Updated in program order for precise exceptions
    - Use a reorder buffer to ensure in-order updates
  - Future reg file: Updated as soon as an instruction completes (if the instruction is the youngest one to write to a register)

- Future file is used for fast access to latest register values (speculative state) **ספקולטיבי** זה לא לפי סדר התכנית ולכן
  - Frontend register file

- Architectural file is used for **state recovery** on exceptions (**architectural state**)
  - Backend register file

Future File



- Advantage
  - No need to read the new values from the ROB (no CAM or indirection) or the old value of destination register
- Disadvantage
  - Multiple register files
  - Need to copy arch. reg. file to future file on an exception

מכיל ערכים חדשים "ספקולטיבי"

עתה אין צורך בגישה אל ה- ROB

כאן נשמרים הרגיסטרים על פי הסדר שלהם בתכנית (לא ספקולטיביים)

צריך להעתיק את הרגיסטרים במקרה של חריגה. זמן מתכונתי למספר הרגיסטרים

לקריאה נוספת

Smith and Pleszkun [1988]

P&H CAQA Appendix C (C.5)

Implementing Precise Interrupts in Pipelined Processors

JAMES E. SMITH, MEMBER, IEEE, AND ANDREW R. PLESZKUN, MEMBER, IEEE

Abstract—**This paper describes and evaluates solutions to the precise interrupt problem in pipelined processors. An interrupt is *precise* if the saved process state corresponds with a sequential model of program execution where one instruction completes before the next begins. In a pipelined processor, precise interrupts are difficult to implement because an instruction may be initiated before its predecessors have completed.**

The precise interrupt problem is described, and five solutions are discussed in detail. The first solution forces instructions to complete and modify the process state in architectural order. The other four solutions allow instructions to complete in any order, but additional hardware is used so that a precise state can be restored when an interrupt occurs. All the methods are discussed in the context of a parallel pipeline structure. Simulation results based on the CRAY-1S scalar architecture are used to show that the first solution results in a performance degradation of at least 16 percent. The remaining four solutions offer better performance, and three of them result in as little as a 3 percent performance loss. Several extensions, including vector architectures, virtual memory, and linear pipeline structures, are briefly discussed.

Index Terms—**Performance simulation, pipelined computers, precise interrupts, process checkpointing, process recovery, virtual memory.**

I. Introduction

MOST computer architectures are based on a sequential model of program execution in which an architectural program counter sequences through instructions one-by-one, finishing one before starting the next. In contrast, a high-performance implementation may be pipelined, permitting several instructions to be in some phase of execution at the same time. The use of a sequential architecture and a pipelined implementation clash at the time of an interrupt; pipelined instructions may modify the process state in an order different from that defined by the sequential architectural model. At the time an interrupt condition is detected, the hardware may not be in a state that is consistent with any specific program counter value.

When an interrupt occurs, the state of an interrupted process is typically saved by the hardware, the software, or by a combination of the two. The process state generally consists of the program counter, registers, and memory. If the saved process state is consistent with the sequential architectural model, then the interrupt is *precise*. To be more specific, the saved state should reflect the following conditions.

1) All instructions preceding the instruction indicated by the saved program counter have been executed and have modified the process state correctly.

2) All instructions following the instruction indicated by the saved program counter are unexecuted and have not modified the process state.

3) If the interrupt is caused by an exception condition raised by an instruction in the program, the saved program counter points to the interrupted instruction. The interrupted instruction may or may not have been executed, depending on the definition of the architecture and the cause of the interrupt. Whichever is the case, the interrupted instruction has either completed, or has not started execution.

If the saved process state is inconsistent with the sequential architectural model and does not satisfy the above conditions, then the interrupt is *imprecise*.

This paper describes and compares ways of implementing precise interrupts in pipelined processors. The methods used are designed to modify the state of an executing process in a carefully controlled way. The simple methods force all instructions to update the process state in the architectural order. Other, more complex methods save portions of the process state so that the proper state may be restored by the hardware at the time an interrupt occurs.

A. Classification of Interrupts

We consider interrupts belonging to two classes.

1) *Program interrupts*, sometimes referred to as "traps," result from *exception conditions* detected during fetching and execution of specific instructions. These exceptions may be due to software errors such as trying to execute an illegal opcode, numerical errors such as overflow, or they may be part of normal program execution as with page faults.

2) *External interrupts* are not caused by specific instructions and are often caused by sources outside the currently executing process, sometimes completely unrelated to it. I/O interrupts and timer interrupts are examples.

For a specific architecture, all interrupts may be defined to be precise or only a proper subset. Virtually every architecture, however, has some types of interrupts that must be precise. There are a number of conditions under which precise interrupts are either necessary or desirable.

1) For I/O and timer interrupts, a precise process state makes restarting possible.

2) In virtual memory systems, precise interrupts allow a process to be correctly restarted after a page fault has been serviced.

Manuscript received March 23, 1986. This work is supported by the National Science Foundation under Grant ECS-8207277.

J. E. Smith is with the Department of Electrical and Computer Engineering, University of Wisconsin, Madison, WI 53706.

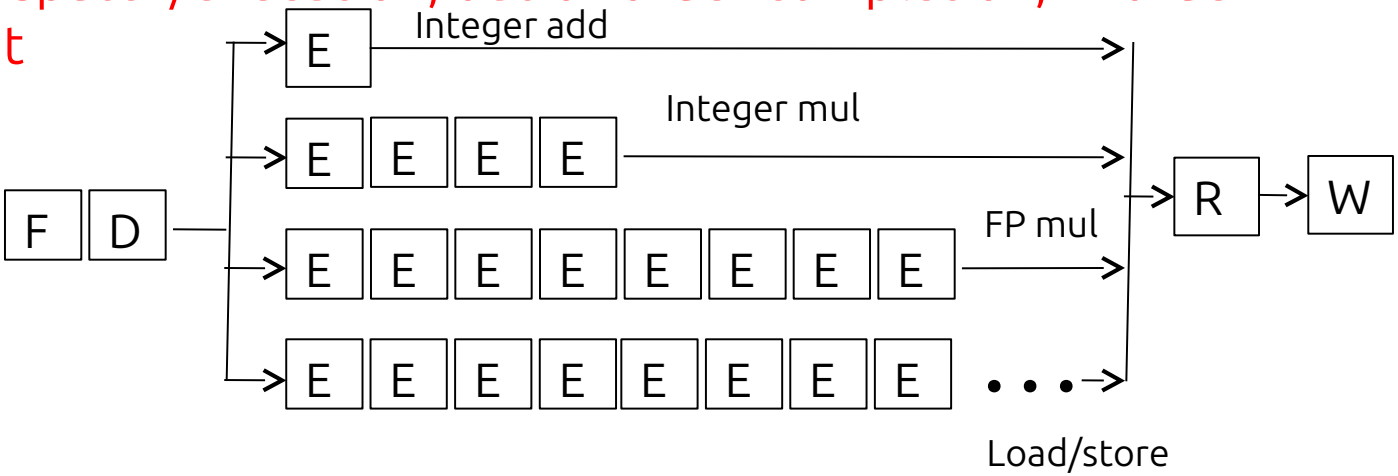
A. R. Pleszkun is with the Department of Computer Sciences, University of Wisconsin, Madison, WI 53706.

IEEE Log Number 8717725.

0018-9340/88/0500-0562$01.00 © 1988 IEEE

In-Order Pipeline with Future File and Reorder Buffer

- Decode (D): Access **future file**, allocate entry in ROB, check if instruction can execute, if so **dispatch** instruction
- Execute (E): Instructions can complete out-of-order
- Completion (R): Write result to **reorder buffer and future file**
- Retirement/Commit (W): Check for exceptions; if none, write result to architectural register file or memory; else, flush pipeline, copy architectural file to future file, and start from exception handler
- In-order dispatch/execution, out-of-order completion, in-order retirement



Can We Reduce the Overhead of Two Register Files?

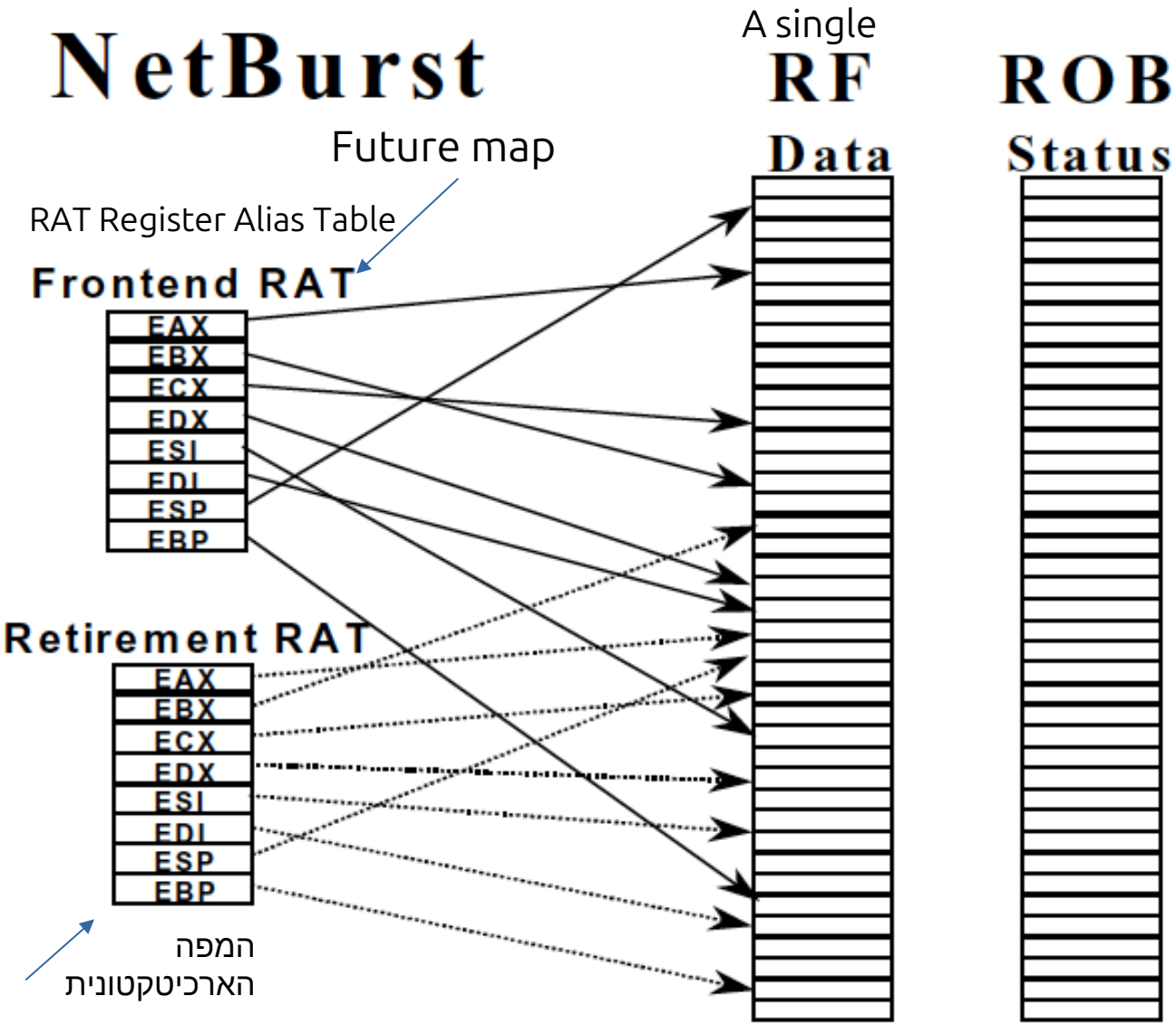
איך לצמצם תקורות עקב שני רגיסטר פיילס?

- Idea: Use pointers to data in front-end and retirement
  - Have a **single storage** that stores register data values
  - Keep two register maps (speculative and architectural); also called **register alias tables** (RATs)

- Future map used for fast access to latest register values (speculative state)
  - Front-end register map

- Architectural map is used for state recovery on exceptions (architectural state)
  - Back-end register map

Future Map in Intel Pentium 4



Boggs et al., "The Microarchitecture of the Pentium 4 Processor," Intel Technology Journal, 2001.

Many modern processors are similar:
-MIPS R10K
-Alpha 21264

זכרון יחיד ושתי מפות

מתעדכן כל פעם שמסתיימת הוראה

NetBurst

RF Data

ROB Status

Frontend RAT

EAX
EBX
ECX
EDX
ESI
EDI
ESP
EBP

אם לדוגמה בשתי המפות EAX פונה לאותו מקום באחסון זה אומר שאין כתיבה לרגיסטר הזה

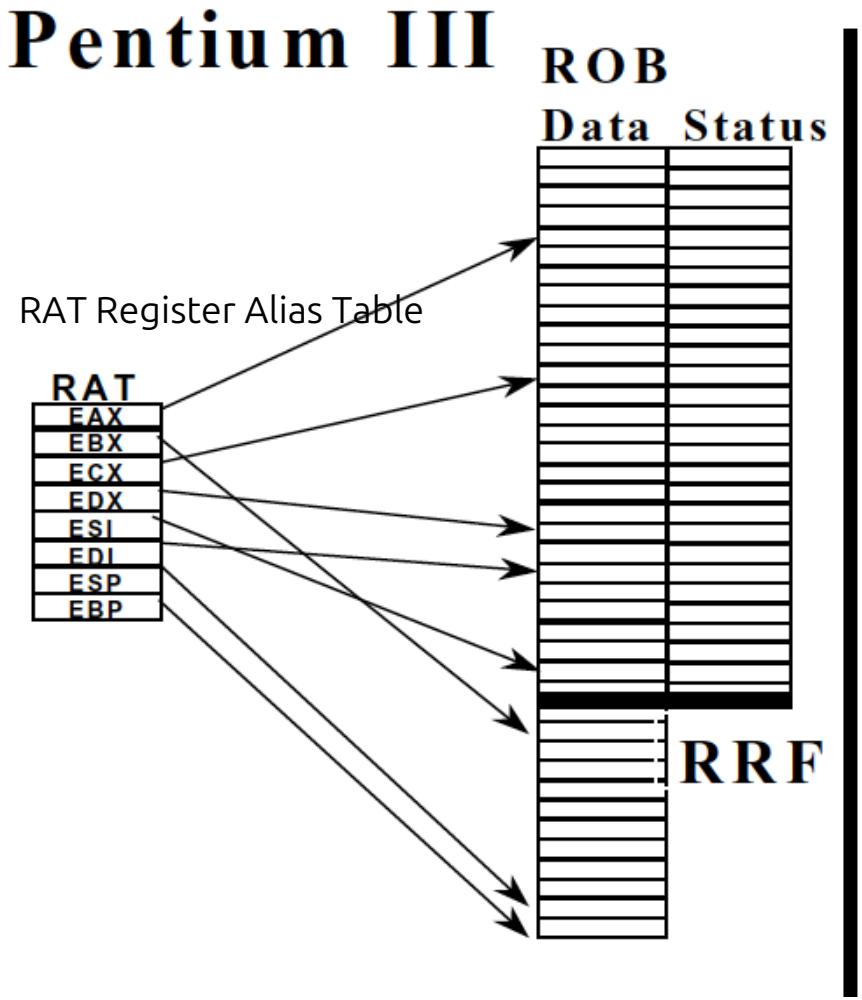
Retirement RAT

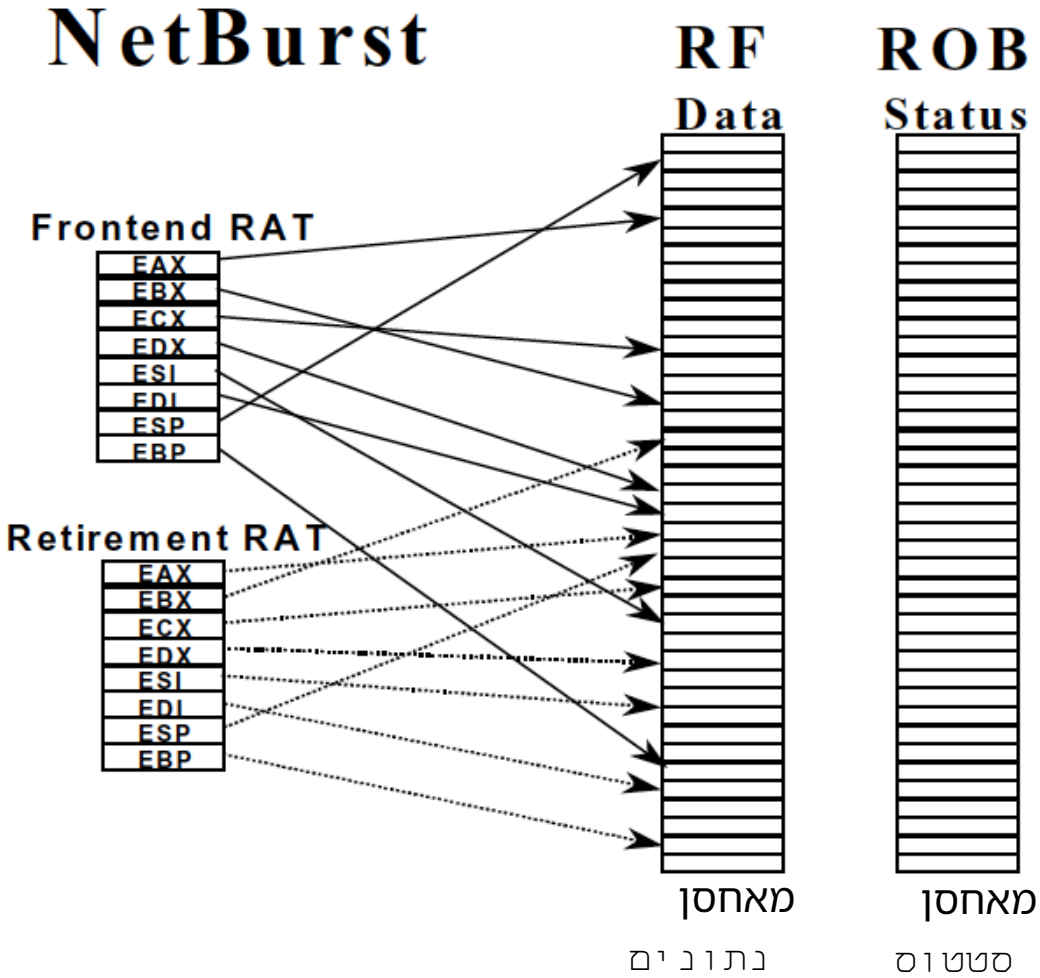
EAX
EBX
ECX
EDX
ESI
EDI
ESP
EBP

מתעדכן בסוף עפ"י סדר התכנית

Reorder Buffer vs. Future Map Comparison

הזכרון עבר פיצול
(צמצום)





מפה אחת, שני אחסונים

2 מיפויים, אחסון אחד

RRF=Retirement RF

Before We Get to Checkpointing …

- Let’s cover what happens on exceptions
- And branch mispredictions

Checking for and Handling Exceptions in Pipelining

- When the oldest instruction ready-to-be-retired is detected to have caused an exception, the control logic:
  - Recovers architectural state (register file, IP, and memory)
  - Flushes all younger instructions in the pipeline
  - Saves IP and registers (as specified by the ISA)
  - Redirects the fetch engine to the exception handling routine
    - Vectored exceptions = כשיש מידע מה סוג החריגה לדילוג אל ה- exception handler

<u>בדיקה וטיפול בחריגות בפייפליין</u>

כאשר ההוראה הכי ותיקה מוכנה לפרוש ומתגלית חריגה בהוראה זו, הלוגיקה של הבקרה תפעל באופן הבא:

א. שחזור המצב הארכיטקטוני (RF, PC וזיכרון).

ב. מחיקה של כל ההוראות הצעירות בפייפליין.

ג. שמירת ה-PC (IP) והרגיסטרים כפי שנדרש על-ידי ה-ISA.

ד. הפניית מנגנון ה-FETCH לרוטינה המטפלת בחריגה.

חריגה וקטורית – כאשר לא רק יש רוטינה כללית אלא גם מידע מה הסוג של החריגה (מה היה הגורם)

Pipelining Issues: Branch Mispredictions

- A branch misprediction resembles an "exception"
  - Except it is not visible to software (i.e., it is microarchitectural)

- What about branch misprediction recovery?
  - Similar to exception handling except can be initiated before the branch is the oldest instruction (not architectural)
  - All three state recovery methods can be used

הכוונה ל-FF, History Buffer, ROB

- Difference between exceptions and branch mispredictions?
  - Branch mispredictions are much more common than exceptions

→ need fast state recovery to minimize performance impact of mispredictions

How Fast Is State Recovery?

- Latency of state recovery affects
  - Exception service latency
  - Interrupt service latency
  - Latency to supply the correct data to instructions fetched after a branch misprediction
- Which ones above need to be fast?
- How do the three state maintenance methods fare in terms of recovery latency?
  - Reorder buffer
  - History buffer
  - Future file

Branch State Recovery Actions and Latency

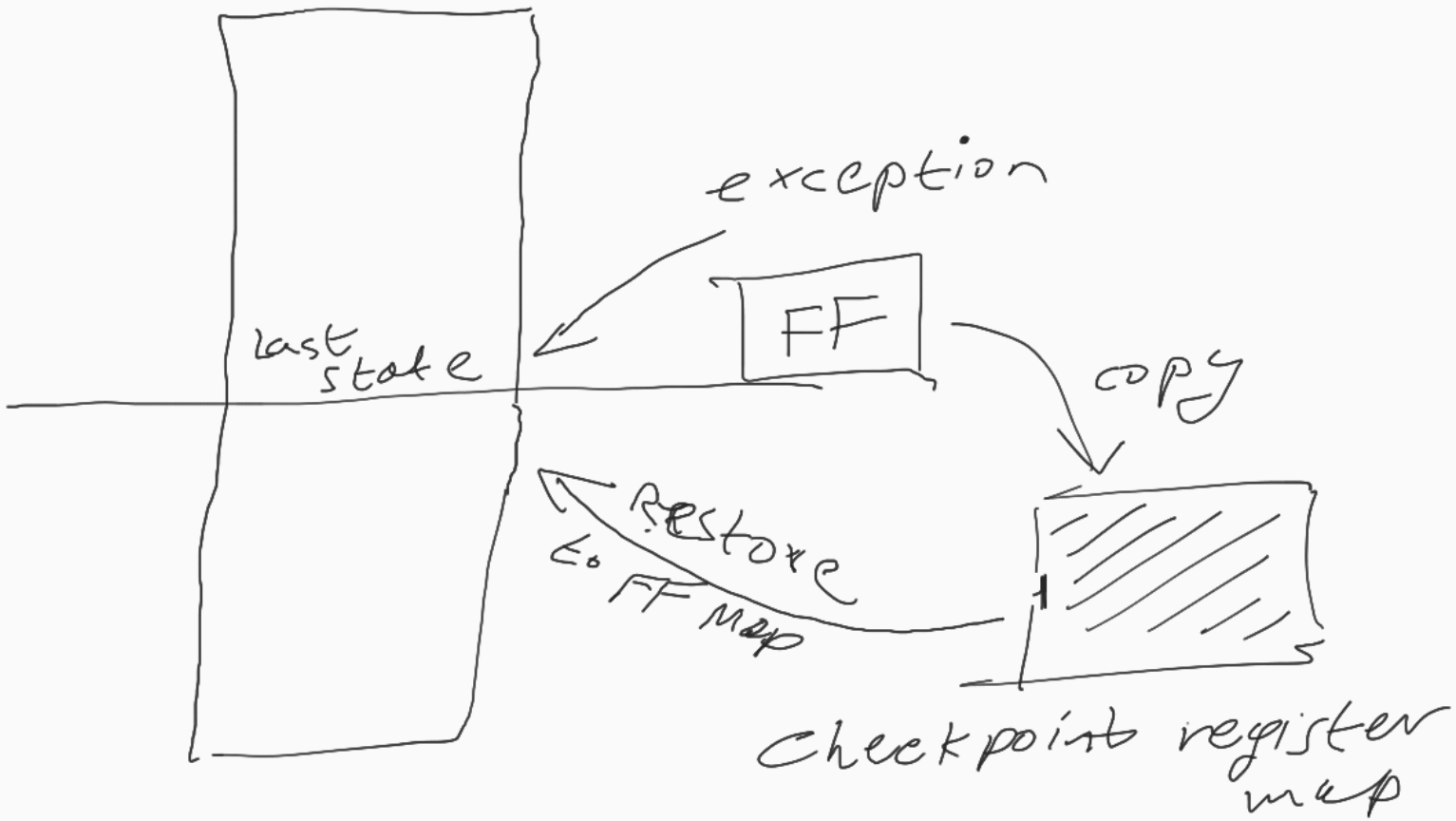
- Reorder Buffer
  - Flush instructions in pipeline younger than the branch
  - Finish all instructions in the reorder buffer
- History buffer
  - Flush instructions in pipeline younger than the branch
  - Undo all instructions after the branch by rewinding from the tail of the history buffer until the branch & restoring old values one by one into the register file
- Future file
  - Wait until branch is the oldest instruction in the machine
  - Copy arch. reg. file to future file
  - Flush entire pipeline

Can We Do Better?

- Goal: Restore the frontend state (future file) such that the correct next instruction after the branch can execute right away after the branch misprediction is resolved

- Idea: Checkpoint the frontend register state/map at the time a branch is decoded and keep the checkpointed state updated with results of instructions older than the branch
  - Upon branch misprediction, restore the checkpoint associated with the branch

- Hwu and Patt, "Checkpoint Repair for Out-of-order Execution Machines," ISCA 1987.


exception
FF
last state
copy
restore
to FF map
checkpoint register map

Checkpointing

- When a branch is decoded
  - Make a copy of the future file/map and associate it with the branch
- When an instruction produces a register value
  - All future file/map checkpoints that are younger than the instruction are updated with the value
- When a branch misprediction is detected
  - Restore the checkpointed future file/map for the mispredicted branch when the branch misprediction is resolved
  - Flush instructions in pipeline younger than the branch
  - Deallocate checkpoints younger than the branch

Checkpointing

- Advantages
  - Correct frontend register state available right after checkpoint restoration → Low state recovery latency
  - …
- Disadvantages
  - Storage overhead
  - Complexity in managing checkpoints
  - …

Many Modern Processors Use Checkpointing

- MIPS R10000
- Alpha 21264
- Pentium 4

- Yeager, "The MIPS R10000 Superscalar Microprocessor," IEEE Micro, April 1996

- Kessler, "The Alpha 21264 Microprocessor," IEEE Micro, March-April 1999.

- Boggs et al., "The Microarchitecture of the Pentium 4 Processor," Intel Technology Journal, 2001.

Summary: Maintaining Precise State

- Reorder buffer
- History buffer
- Future register file

3 פתרונות לחריגות

- Checkpointing

האצת ההתאוששות

- Readings
  - Smith and Plezskun, "Implementing Precise Interrupts in Pipelined Processors," IEEE Trans on Computers 1988 and ISCA 1985.
  - Hwu and Patt, "Checkpoint Repair for Out-of-order Execution Machines," ISCA 1987.

Computer Architecture

Simulators

Course: 361-1-4201

Dr. Guy Tel-Zur

במצגת זו מתוארים כלי סימולציה, חופשיים, אשר יכולים לסייע בהבנה של חומרי הלימוד באמצעות תרגול עצמי.

המצגת תתעדכן מעת לעת. עדכון אחרון: ~~11/4/2024~~ 11/1/2026

In 3 Parts...

~~Part 1: MIPS~~

Part 2: RISC-V

Part 3: Additional tools

Part 1: MIPS

Starting from 2026B we totally migrate to RISC-V

MARS

Java based MIPS simulator

Home page:

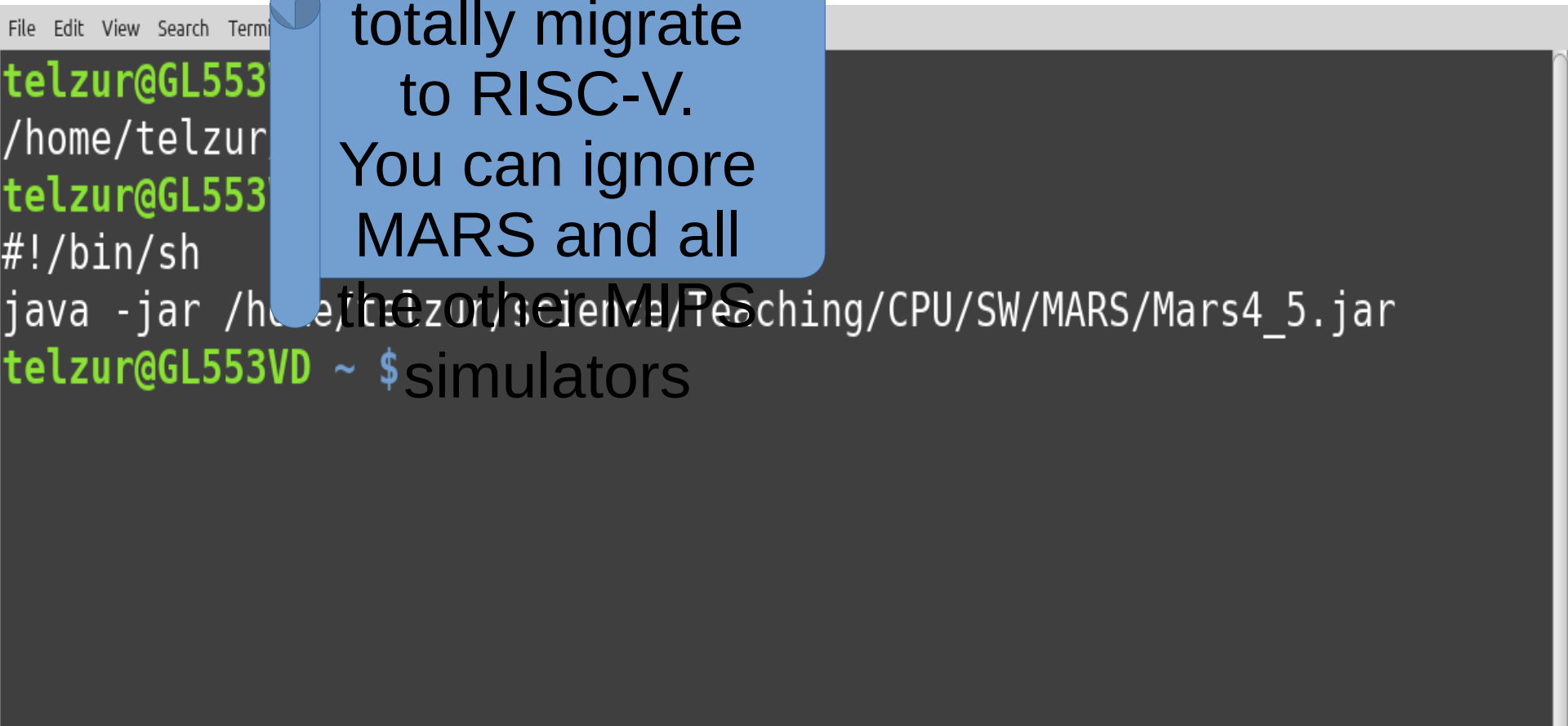
http://co e
du/KenV

Starting from 2026B we totally migrate to RISC-V

MARS

Invocation (Linux):

Starting from 2026B we totally migrate to RISC-V. You can ignore MARS and all the other MIPS simulators



MARS

File Edit Run Settings Tools Help

Run speed at max (no interaction)

Edit | Execute

Text Segment

Bkpt	Address	Code	Basic	Source
	0x00400000	0x24020004	addiu $2,$0,4	10: li $v0, 4
	0x00400004	0x3c011001	lui $1,4097	11: la $a0, msg0
	0x00400008	0x34240000	ori $4,$1,0	
	0x0040000c	0x0000000c	syscall	12: syscall
	0x00400010	0x3c011001	lui $1,4097	li $s0, 0x10010000 # begining data segment
	0x00400014	0x34300000	ori $16,$1,0	
	0x00400018	0x82080000	lb $8,0($16)	18: lb $t0,0x00($s0) # load byte from Mem to Reg
	0x0040001c	0x82090001	lb $9,1($16)	19: $t1,0x01($s0)
	0x00400020	0x820a0002	lb $10,2($16)	20: $t2,0x02($s0)
	0x00400024	0x820b0003	lb $11,3($16)	21: 2,0x03($s0)
	0x00400028	0x3c011001	lui $1,4097	24: 0x10010040
	0x0040002c	0x34310040	ori $17,$1,64	
	0x00400030	0xa2280000	sb $8,0($17)	
	0x00400034	0xa2290001	sb $9,1($17)	
	0x00400038	0xa22a0002	sb $10,2($17)	
	0x0040003c	0xa22b0003	sb $11,3($17)	
	0x00400040	0x24020004	addiu $2,$0,4	

Data Segment

Address	(0)					
0x10010000	440324					
0x10010020	6807					
0x10010040	0			0	0	0
0x10010060				0	0	0
0x10010080				0	0	0
0x100100a0				0	0	0
0x100100c0				0	0	0
0x100100e0						
0x10010100						
0x10010120		0	0			
0x10010140		0	0			
0x10010160		0	0	0		
0x10010180		0	0	0		
0x100101a0		0	0	0		
0x100101c0		0	0			

0x10010000 (.data) ☑ Hexadecimal Addresses ☐ Hexadecimal Values ☐ ASCII

Mars Messages | Run I/O

```
Invalid language element: @function
Error in /mnt/038d7f89-5713-462a-a96b-aa9afa3f6f93/Teaching/CPU/lectures/03/code/slide123/slide123_mips.asm line 5 column 10:   .module fp=xx
Invalid language element: fp=xx
Error in /mnt/038d7f89-5713-462a-a96b-aa9afa3f6f93/Teaching/CPU/lectures/03/... line 20 column 14:
Invalid language element: @function
Error in /mnt/038d7f89-5713-462a-a96b-aa9afa3f6f93/Teaching/CPU/lectures/03/code/slide123/slide123_mips.asm
Invalid language element: @function
Assemble: operation completed with errors.

Assemble: assembling /mnt/038d7f89-5713-462a-a96b-aa9afa3f6f93/Teaching/CPU/lectures/03/code/simple_demos/lbsb.asm

Assemble: operation completed successfully.
```

Clear

Registers | Coproc 1 | Coproc 0

Name	Number	Value
$zero	0	0
$at	1	0
$v0	2	0
$v1	3	0
$a0	4	0
$a1	5	0
$a2	6	0
$a3	7	0
$t0	8	0
$t1	9	0
$t2	10	0
$t3	11	0
$t4	12	0
$t5	13	0
$t6	14	0
$t7	15	0
$s0	16	0
$s1	17	0
$s2	18	0
$s3	19	0
$s4	20	0
$s5	21	0
$s6	22	0
$s7	23	0
$t8	24	0
$t9	25	0
$k0	26	0
$k1	27	0
$gp	28	268468224
$sp	29	2147479548
$fp	30	0
$ra	31	0
pc		4194304
hi		0
lo		0

Starting from 2026B we totally migrate to RISC-V

2) Go

The code

Memory

Terminal

The registers

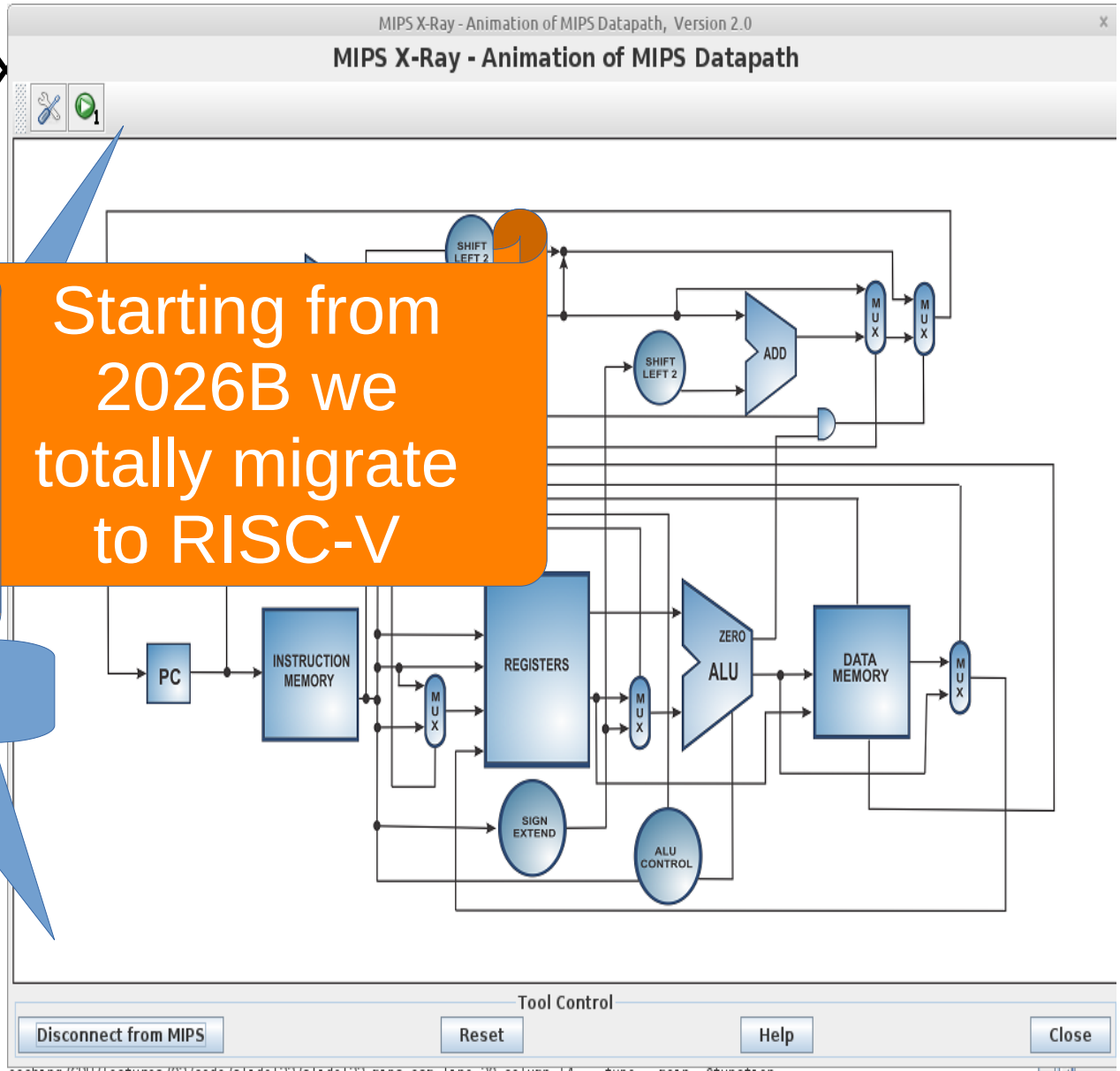
MARS

Tools → X
Ray

2) Execute

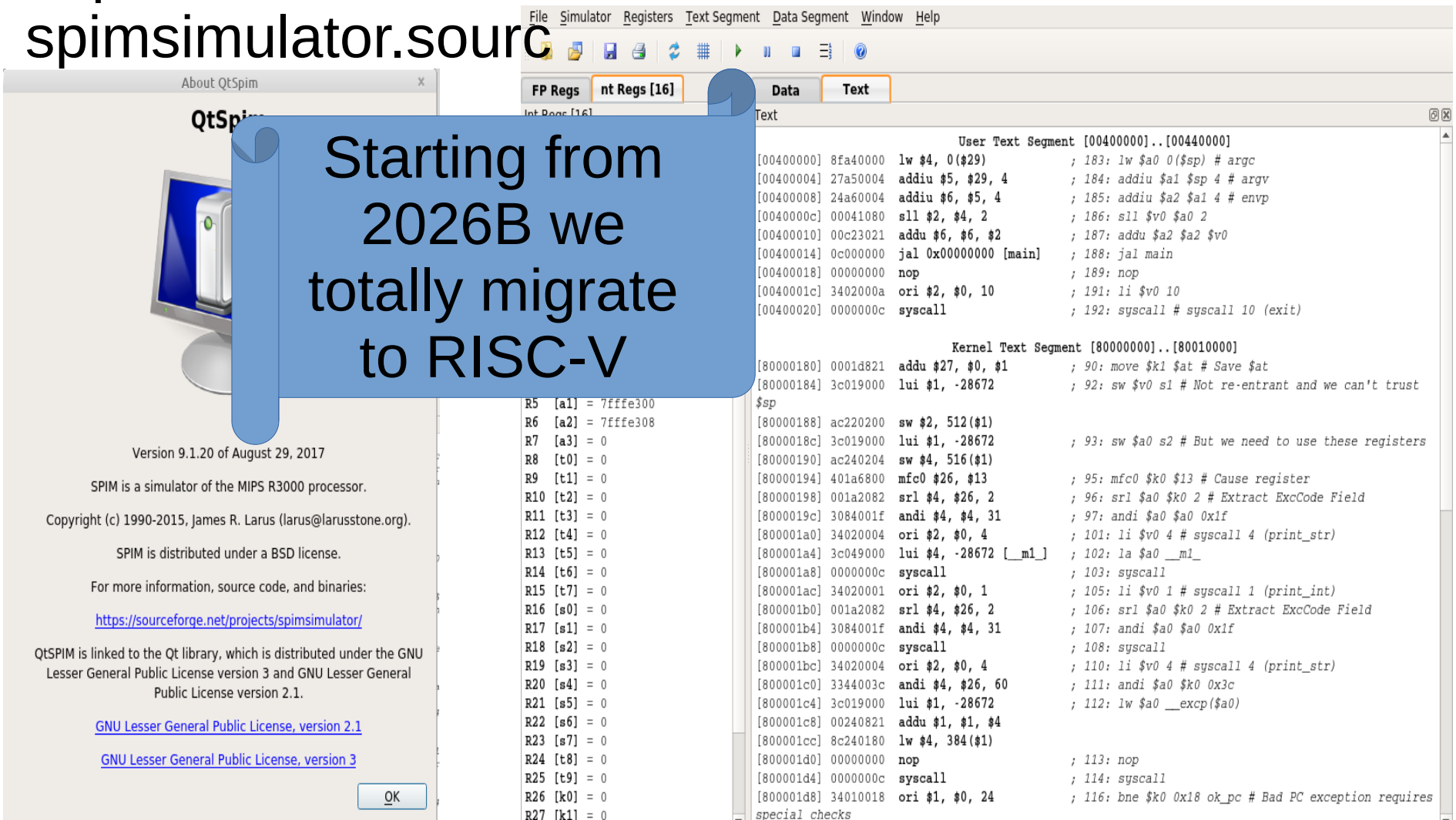
1) Connect to MIPS

Starting from 2026B we totally migrate to RISC-V



SPIM / QtSPIM

https://
spimsimulator.sourc



Starting from 2026B we totally migrate to RISC-V

QtSPIM

The code is loaded into the simulator
To execu
click on
"Simulat

→ Run (F5) or Single Step (F10)

Starting from 2026B we totally migrate to RISC-V

```
telzur@GL553VD ~/tmp/R                    e ./qtspim_example.s
    .text
    .globl main
main:
    li $t0, 0x2 # $t0  0x2
    li $t1, 0x3 # $t1  0x3
    addu $t2  $t0, $t1 # $t2  ADD($t0, $t1)
telzur@GL553VD ~/tmp/RISC-V $
```

QtSpim

File Simulator Registers Text Segment Data Segment Window Help

FP Regs | nt Regs [16] | Data | Text

Int Regs [16]

```
PC       = 400030
EPC      = 0
Cause    = 0
BadVAddr = 0
Status   = 3000ff10

HI       = 0
LO       = 0

R0  [r0] = 0
R1  [at] = 0
R25 [t9] = 0
R26 [k0] = 0
R27 [k1] = 0
```

Text

```
                    User Text Segment [00400000]..[00440000]
[00400000] 8fa40000  lw $4, 0($29)            ; 183: lw $a0 0($sp) # argc
[00400004] 27a50004  addiu $5, $29, 4         ; 184: addiu $a1 $sp 4 # argv
[00400008] 24a60004  addiu $6, $5, 4          ; 185: addiu $a2 $a1 4 # envp
[0040000c] 00041080  sll $2, $4, 2            ; 186: sll $v0 $a0 2
[00400010] 00c23021  addu $6, $6, $2          ; 187: addu $a2 $a2 $v0
[00400014] 0c100009  jal 0x00400024 [main]    ; 188: jal main
[00400018] 00000000  nop                      ; 189: nop
[0040001c] 3402000a  ori $2, $0, 10           ; 191: li $v0 10
[00400020] 0000000c  syscall                  ; 192: syscall # syscall 10 (exit)
[00400024] 34080002  ori $8, $0, 2            ; 4: li $t0, 0x2 # $t0 0x2
[00400028] 34090003  ori $9, $0, 3            ; 5: li $t1, 0x3 # $t1 0x3
[0040002c] 01095021  addu $10, $8, $9         ; 6: addu $t2, $t0, $t1 # $t2 ADD($t0, $t1)

                    Kernel Text Segment [80000000]..[80010000]
[80000000] 001d821   addu $27, $0, $1         ; 90: move $k1 $at # Save $at
[80000004] 3c019000  lui $1, -28672           ; 92: sw $v0 s1 # Not re-entrant and we can't trust
[80000008] ac220200  sw $2, 512($1)
[8000000c] 3c019000  lui $1, -28672           ; 93: sw $a0 s2 # But we need to use these registers
[80000010] ac240204  sw $4, 516($1)
[80000014] 401a6800  mfc0 $26, $13            ; 95: mfc0 $k0 $13 # Cause register
[80000018] 001a2082  srl $4, $26, 2           ; 96: srl $a0 $k0 2 # Extract ExcCode Field
[8000001c] 3084001f  andi $4, $4, 31          ; 97: andi $a0 $a0 0x1f
[800001a0] 34020004  ori $2, $0, 4            ; 101: li $v0 4 # syscall 4 (print_str)
[800001a4] 3c049000  lui $4, -28672 [__m1_]   ; 102: la $a0 __m1_
[800001a8] 0000000c  syscall                  ; 103: syscall
[800001ac] 34020001  ori $2, $0, 1            ; 105: li $v0 1 # syscall 1 (print_int)
[800001b0] 001a2082  srl $4, $26, 2           ; 106: srl $a0 $k0 2 # Extract ExcCode Field
[800001b4] 3084001f  andi $4, $4, 31          ; 107: andi $a0 $a0 0x1f
[800001b8] 0000000c  syscall                  ; 108: syscall
[800001bc] 34020004  ori $2, $0, 4            ; 110: li $v0 4 # syscall 4 (print_str)
[800001c0] 3344003c  andi $4, $26, 60         ; 111: andi $a0 $k0 0x3c
[800001c4] 3c019000  lui $1, -28672           ; 112: lw $a0 __excp($a0)
[800001c8] 00240821  addu $1, $1, $4
[800001cc] 8c240180  lw $4, 384($1)
[800001d0] 00000000  nop                      ; 113: nop
```

```
Attempt to execute non-instruction at 0x00400030
Attempt to execute non-instruction at 0x00400030
Attempt to execute non-instruction at 0x00400030
Attempt to execute non-instruction at 0x00400030
```

EduMIPS

https://edumips.org/

Starting from 2026B we totally migrate to RISC-V

EduMIPS64 v. 1.2.7.1 - MIPS64 Processor Simulator

File Execute Configure Tools Window Help

Cycles

Home screen

Registers

R0	($zero)	0000000000000000	F0	0000000000000000
R1	($at)	0000000000000000	F1	0000000000000000
R2	($v0)	0000000000000000	F2	0000000000000000
R3	($v1)	0000000000000000	F3	0000000000000000
R4	($a0)	0000000000000000	F4	0000000000000000
R5	($a1)	0000000000000000	F5	0000000000000000
R6	($a2)	0000000000000000	F6	0000000000000000
R7	($a3)	0000000000000000	F7	0000000000000000
R8	($t0)	0000000000000000	F8	0000000000000000
R9	($t1)	0000000000000000	F9	0000000000000000
R10	($t2)	0000000000000000	F10	0000000000000000
R11	($t3)	0000000000000000	F11	0000000000000000
			F12	0000000000000000
			F13	0000000000000000
			F14	0000000000000000
			F15	0000000000000000
			F16	0000000000000000
			F17	0000000000000000

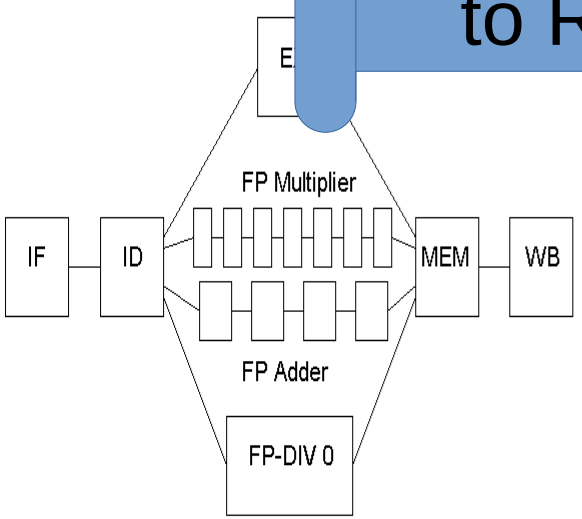
Statistics

Execution
0 Cycles
0 Instructions

Stalls
0 RAW Stalls
0 WAW Stalls
0 WAR Stalls
0 Structural Stalls (Divider not available)
0 Structural Stalls (Memory not available)
0 Branch Taken Stalls
0 Branch Misprediction Stalls
Code size
0 Bytes
Floating point unit
FCSR register
FCC Cause EnablFlag RM
7654321 0 VZOUIVZOUIVZOUI
00000000000000000000111100000001

Starting from 2026B we totally migrate to RISC-V

Pipeline



Data

0010	0000000000000000
0018	0000000000000000
0020	0000000000000000
0028	0000000000000000
0030	0000000000000000
0038	0000000000000000
0040	0000000000000000
0048	0000000000000000
0050	0000000000000000
0058	0000000000000000
0060	0000000000000000
0068	0000000000000000
0070	0000000000000000
0078	0000000000000000
0080	0000000000000000

Code

Address	Representation	Label	Instruction
0000			
0004			
0008			
000C			
0010			
0014			
0018			
001C			
0020			
0024			
0028			
002C			
0030			
0034			
0038			
003C			
0040			

Welcome to EduMIPS64 1.2.7.1

EduMIPS64 v. 1.2.7.1 - MIPS64 Processor Simulator - guy1.asm

File Execute Configure Tools Window Help

Cycles

daddi r5,r0,format_
sw r5,fs_addr(r0)
daddi r2,r0,s1

IF	ID	EX	MEM	WB
	IF	ID	RAW	RAW
		IF		

Registers

R0	($zero)	0000000000000000	F0	0000000000000000
R1	($at)	0000000000000000	F1	0000000000000000
R2	($v0)	0000000000000000	F2	0000000000000000
R3	($v1)	0000000000000000	F3	0000000000000000
R4	($a0)	0000000000000000	F4	0000000000000000
R5	($a1)	0000000000000000	F5	0000000000000000
R6	($a2)	0000000000000000	F6	0000000000000000
R7	($a3)	0000000000000000	F7	0000000000000000
R8	($t0)	0000000000000000	F8	0000000000000000
R9	($t1)	0000000000000000	F9	0000000000000000
R10	($t2)	0000000000000000	F10	0000000000000000
R11	($t3)	0000000000000000	F11	0000000000000000
			F12	0000000000000000
			F13	0000000000000000
			F14	0000000000000000
			F15	0000000000000000
			F16	0000000000000000
			F17	0000000000000000

Statistics

Execution
5 Cycles
0 Instructions

Stalls
2 RAW Stalls
0 WAW Stalls
0 WAR Stalls
0 Structural Stalls (Divider not available)
0 Structural Stalls (Memory not available)
0 Branch Taken Stalls
0 Branch Misprediction Stalls

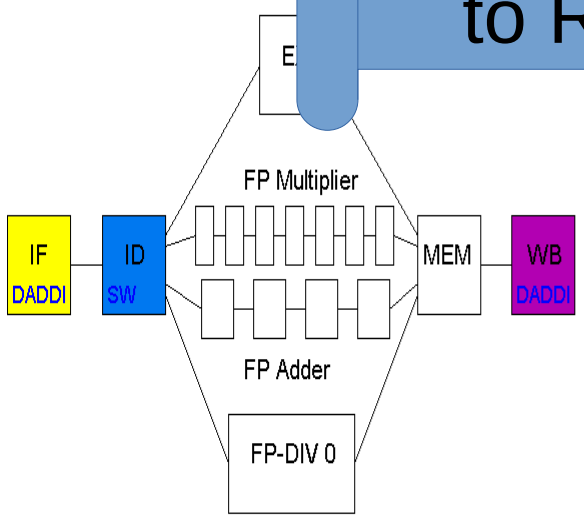
Code size
36 Bytes

Floating point unit
FCSR register
FCC Cause EnablFlag RM
7654321 0 VZOUIVZOUIVZOUI
00000000000000000000111100000001

While running a code

Starting from 2026B we totally migrate to RISC-V

Pipeline



				Data
			str	.asciiz "%dth of %s
0010	25206E6F69737265			
0018	2073692069252E69			
0020	657420676E696562			
0028	0000002164657473			
0030	00000000656E754A	s1		.asciiz "June"
0038	365350494D756445	s2		.asciiz "EduMIPS64"
0040	0000000000000034			
0048	0000000000000000	fs_addr		.space 4
0050	0000000000000005			.word 5
0058	0000000000000000	s1_addr		.space 4
0060	0000000000000000	s2_addr		.space 4
0068	0000000000000000			.word 0
0070	0000000000000005			.word 5
0078	0000000000000000	test		
0080	0000000000000000			

Code

Address	Representation	Label	Instruction
0000	60050000		daddi r5,r0,format_
0004	AC050048		sw r5,fs_addr(r0)
0008	60020030		daddi r2,r0,s1
000C	60030038		daddi r3,r0,s2
0010	FC020058		sd r2,s1_addr(r0)
0014	FC030060		sd r3,s2_addr(r0)
0018	600E0048		daddi r14,r0,fs_add
001C	0000014C		syscall 5
0020	0000000C		syscall 0
0024			
0028			
002C			
0030			
0034			
0038			
003C			
0040			

Welcome to EduMIPS64 1.2.7.1 RUNNING

Dr. MIPS

https://brunonova.github.io/drmips/
https://github.com/brunonova/drmips

https://w atch?v=dyDSV-3v1Ns

Starting from 2026B we totally migrate to RISC-V

ation (CISPEE)

rchitecture Teaching
ing

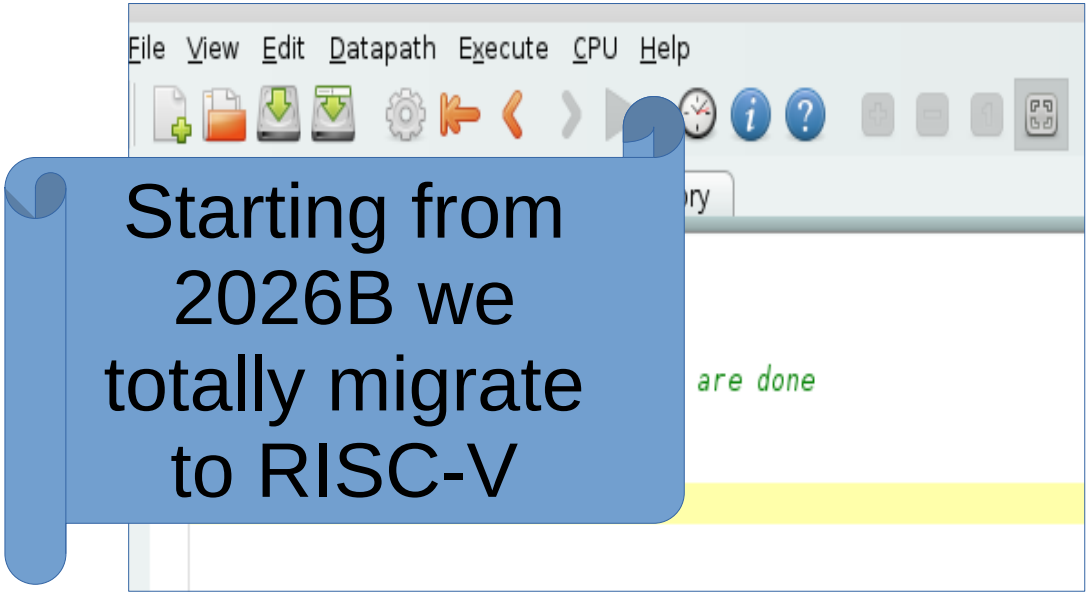
Bruno Nova[1], João C. Ferreira[2], António Araújo[2]
[1]Faculdade de Engenharia da Universidade do Porto, Porto, Portugal
[2]INESC TEC and Faculdade de Engenharia da Universidade do Porto, Porto, Portugal

***Abstract*—Computer architecture is an important subject for informatics and electrical engineering courses. However, students display some difficulties in this subject, mainly due to the lack of educational tools that are intuitive, versatile and graphical. Existing tools are not adequate enough or are very specific. In this paper, an educational MIPS simulator, DrMIPS, is described. This tool simulates the execution of an assembly program on the CPU and displays the datapath graphically. Registers, data memory and assembled code are also displayed and a "performance mode" is also provided. Both unicycle and**

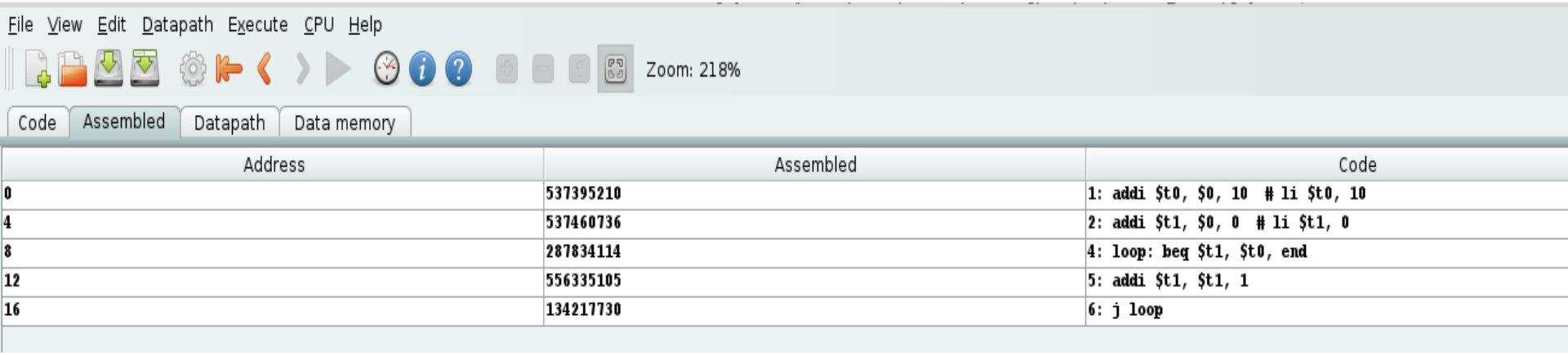
B. Objectives

The main objective of the work presented in this paper was to create a tool to support computer architecture teaching and learning. This educational tool is a simulator the MIPS processor [1], which is a well-known processor in the computer architecture academic community and also one of the most used processors for teaching computer architecture courses in universities [2]. This simulator was developed under the

Dr. MIPS

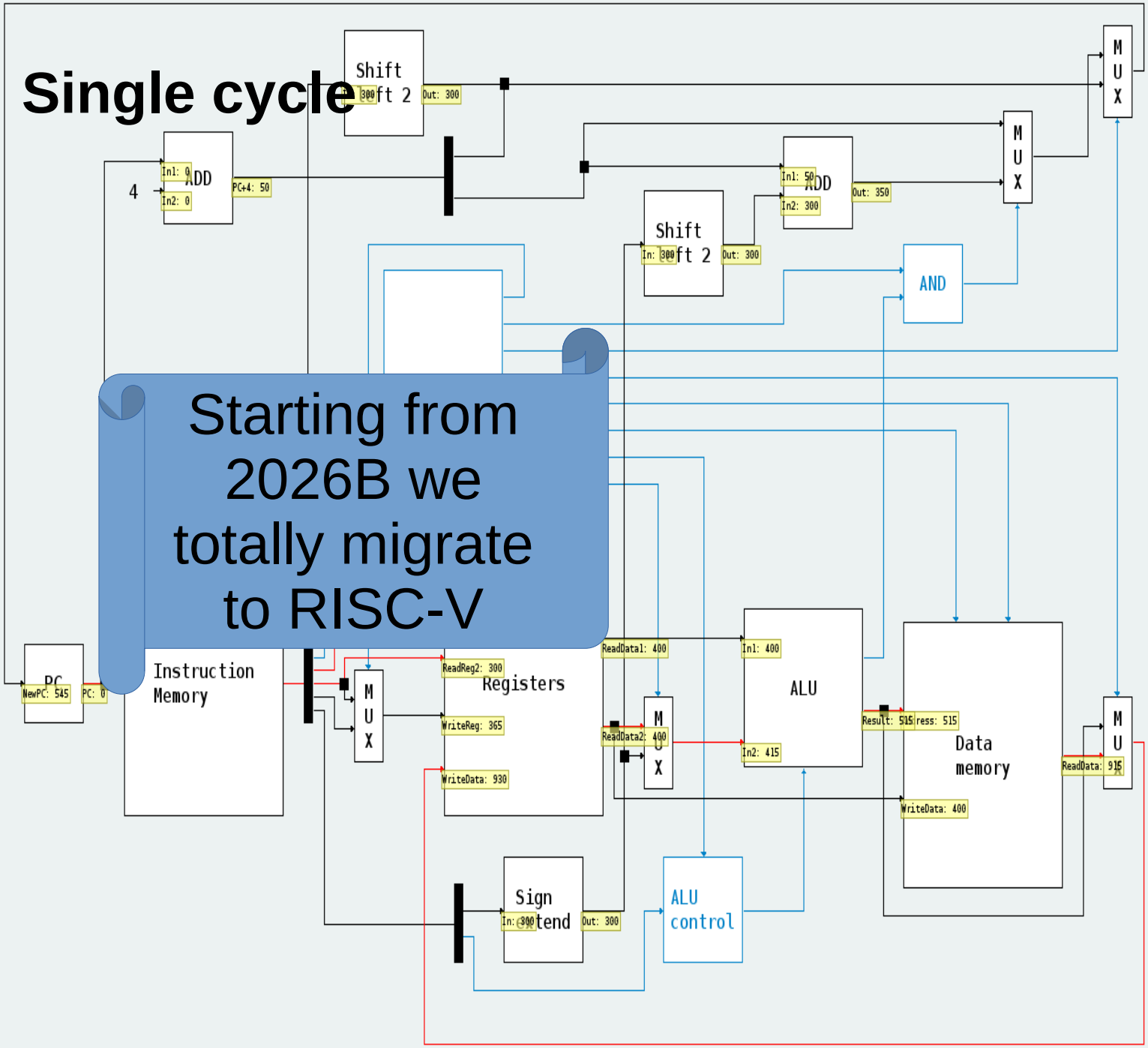


Starting from 2026B we totally migrate to RISC-V



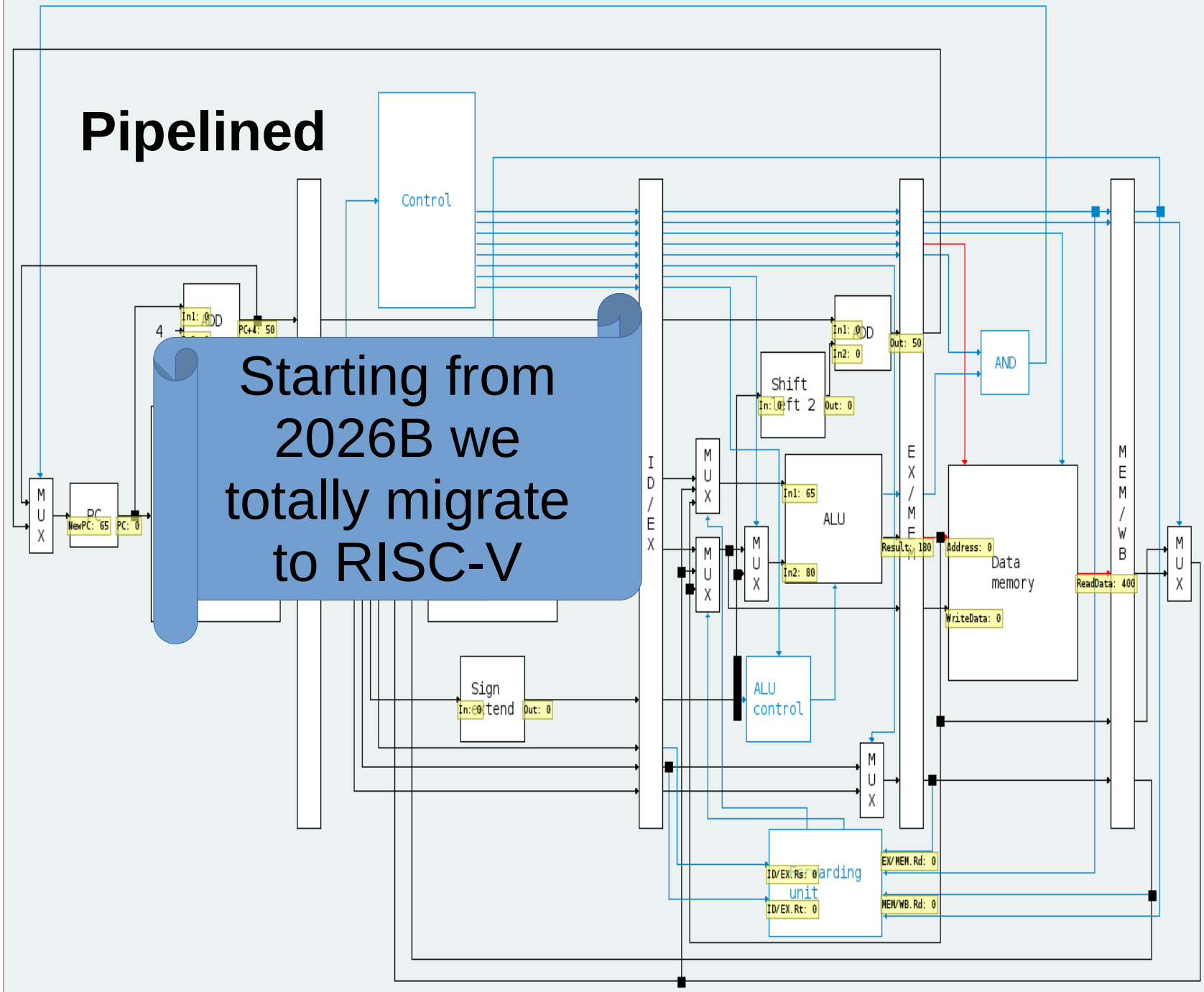
Address	Assembled	Code
0	537395210	1: addi $t0, $0, 10 # li $t0, 10
4	537460736	2: addi $t1, $0, 0 # li $t1, 0
8	287834114	4: loop: beq $t1, $t0, end
12	556335105	5: addi $t1, $t1, 1
16	134217730	6: j loop

Single cycle



Starting from 2026B we totally migrate to RISC-V

Pipelined

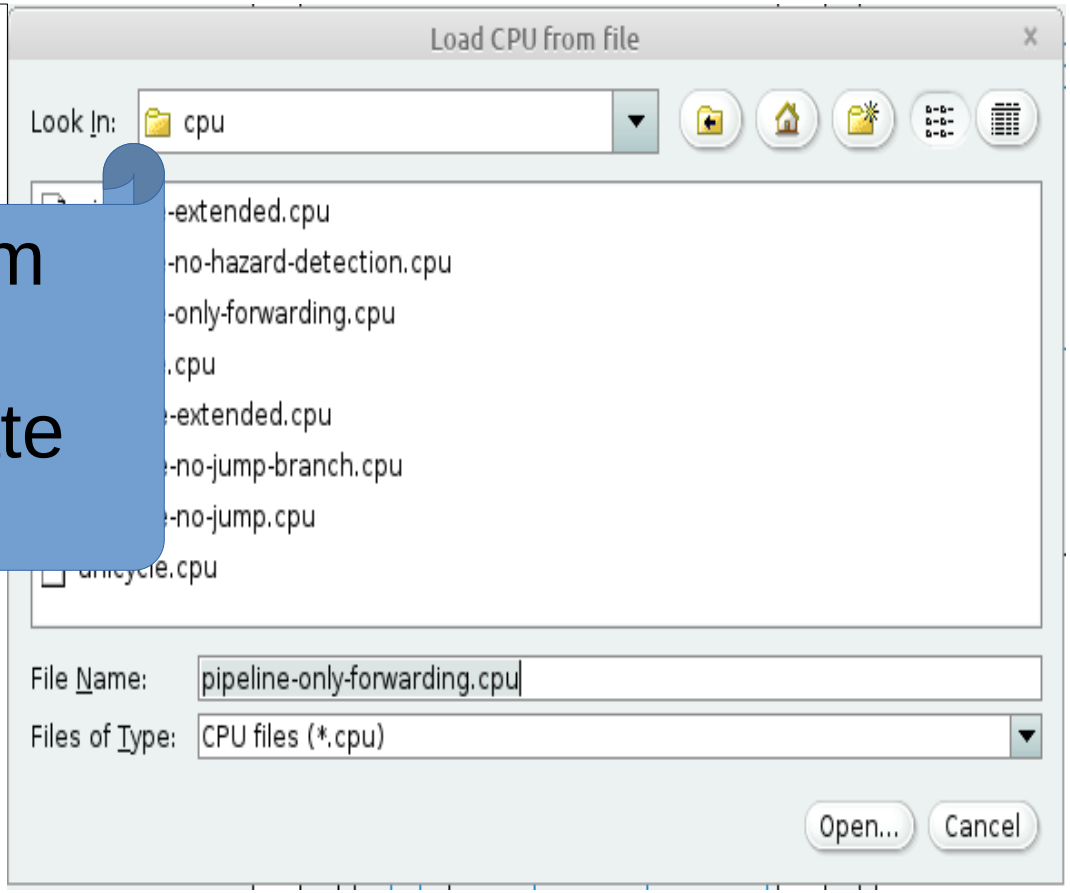


Starting from 2026B we totally migrate to RISC-V

Dr. MIPS

There is a support for several micro-archite
(but t
suppor
many c

Starting from 2026B we totally migrate to RISC-V



QtMIPS

This is an advanced MIPS simulation tool!

Home

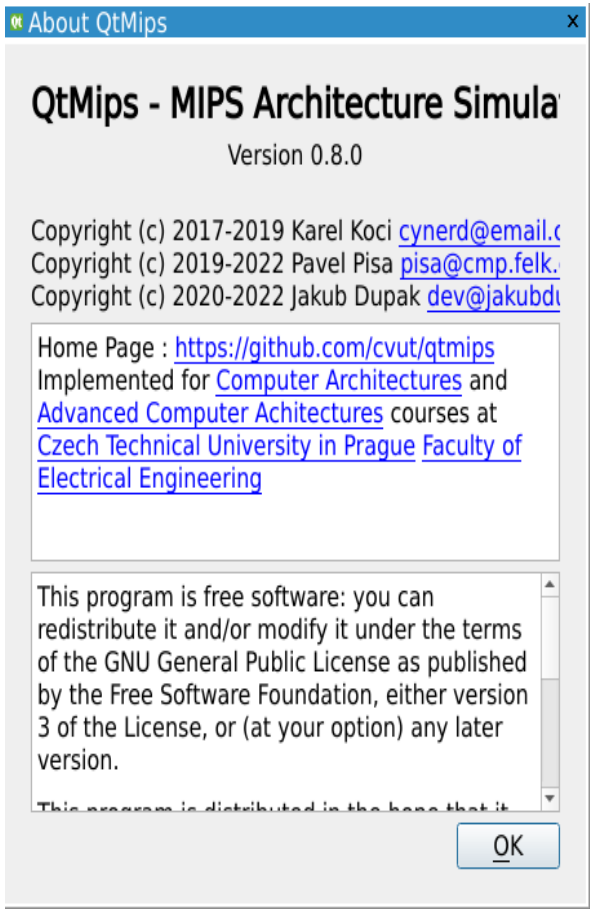
https:// cz/qtmips/a

Starting from 2026B we totally migrate to RISC-V

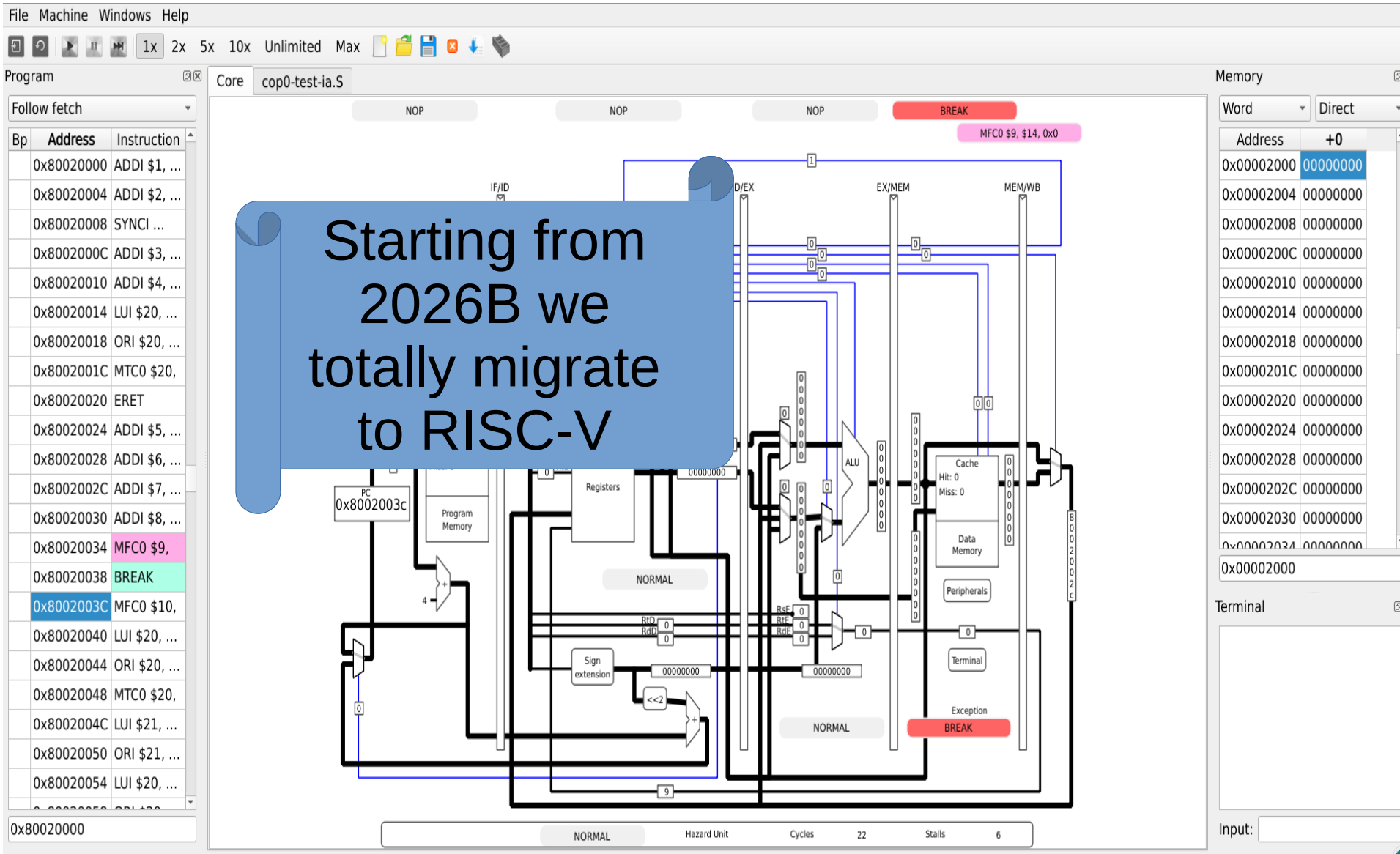
See a screen capture in the next slide

Tutorial video:

https://www.youtube.com/watch?v=6xH72UBvnaY



QtMIPS



Starting from 2026B we totally migrate to RISC-V

Part 2: RISC-V

Starting from 2026B RISC-V is the course architecture

RARS

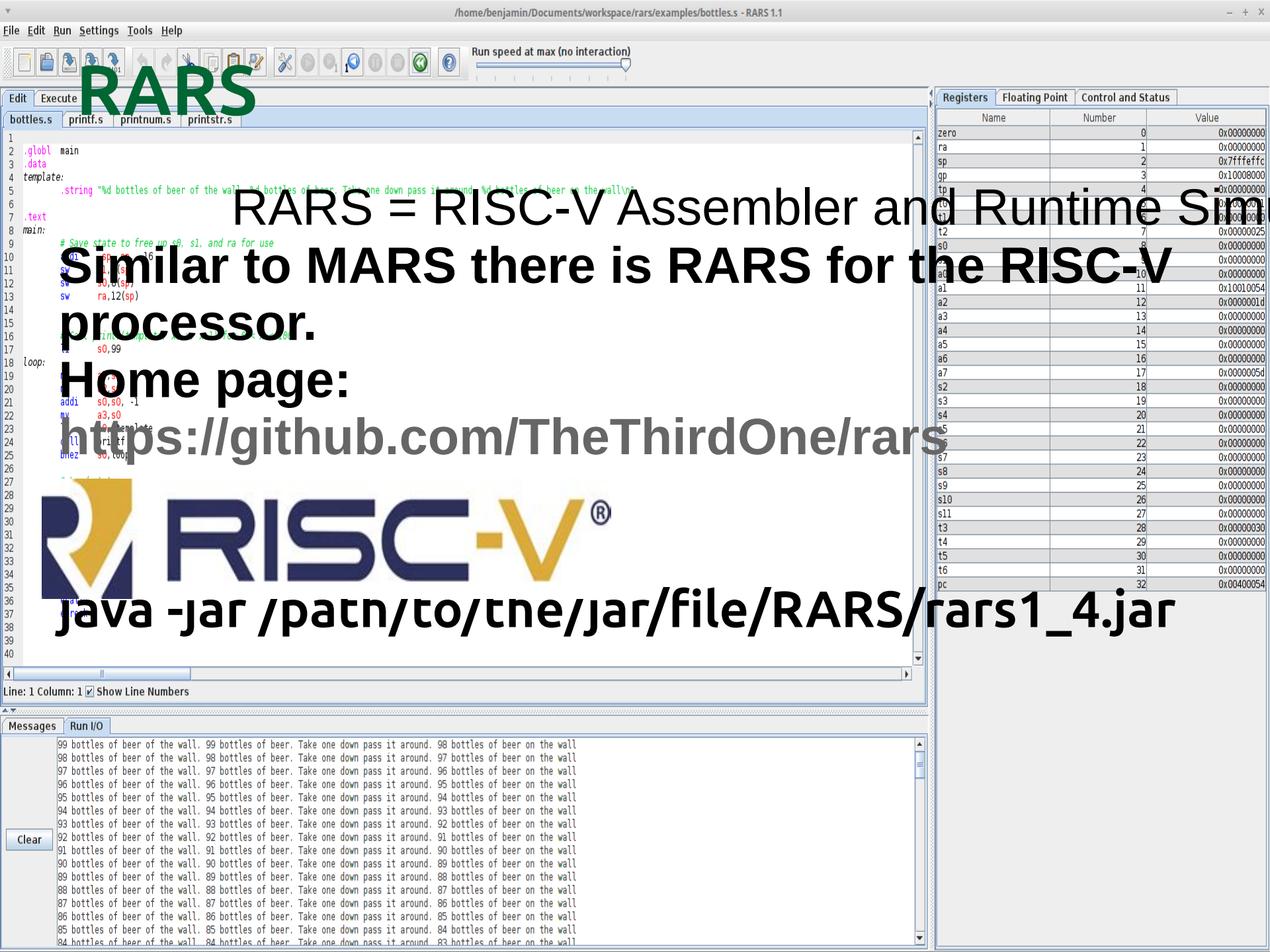
RARS = RISC-V Assembler and Runtime Simu

Similar to MARS there is RARS for the RISC-V processor.

Home page:

https://github.com/TheThirdOne/rars

java -jar /path/to/the/jar/file/RARS/rars1_4.jar



RISC-V

Home page:
https://riscv-programming.org/ale/#home
(it was tested on Google Chrome)

This is a portal for executing RISC-V code.

<u>RISC-V emulation is done here in two stages:</u>
1) build: This is done on the local (Linux) machine with the RISC-V tool chain.
2) Upload and execute the code in the portal.

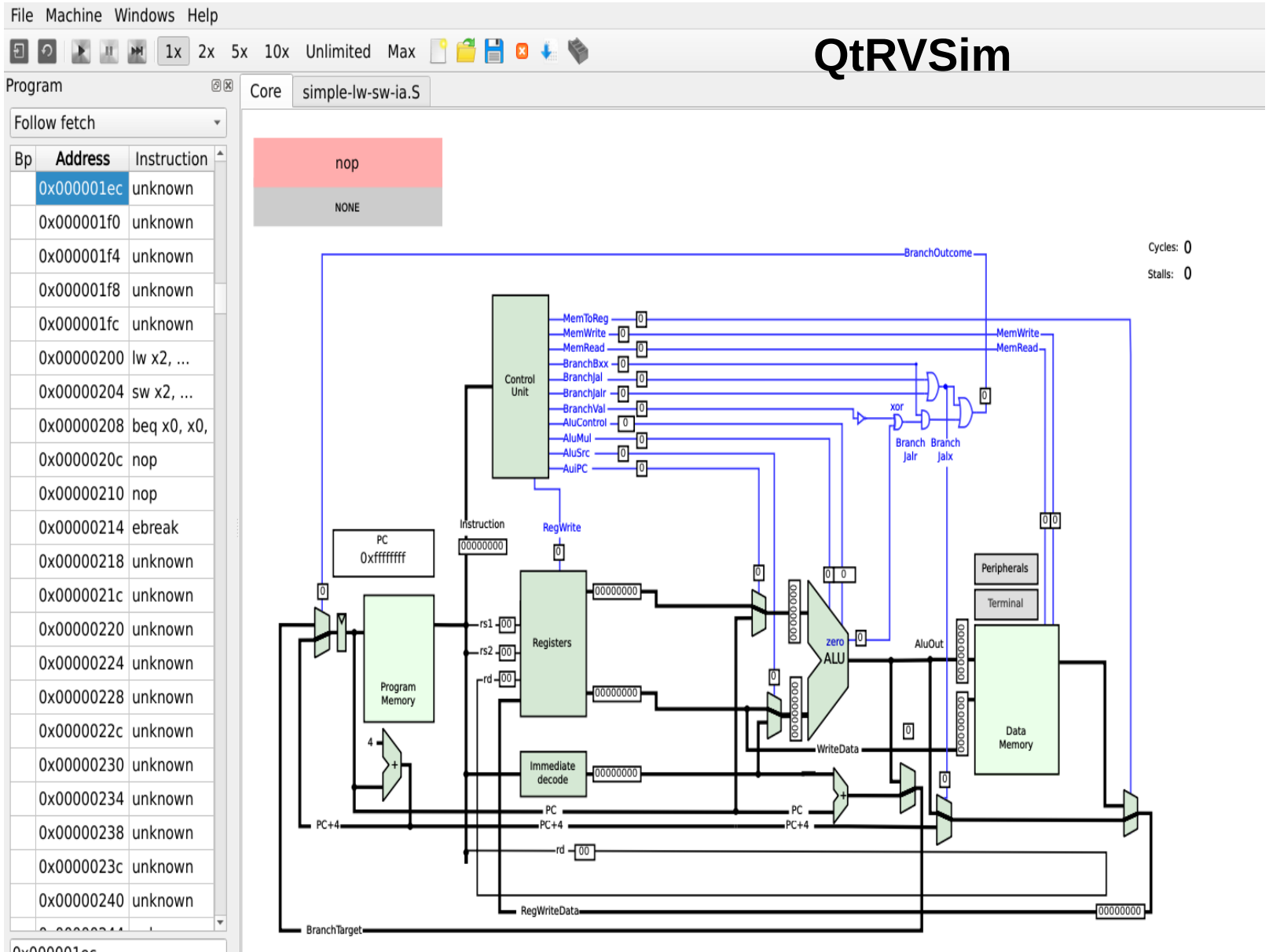
RISC-V

QtRVSi
m
QtRVSim is the counter part to the QtMIPS simulator

Tutorial are available at:
https://comparch.edu.cvut.cz/slides/ewc22-qtrvsim.pdf
and:
https://drive.google.com/file/d/1Rxd6-Qm6LhkDjd9740IAkY_kX1roLOeO/view

The simulator is available at:
https://github.com/cvut/qtrvsim

RISC-V


File Machine Windows Help
1x 2x 5x 10x Unlimited Max
QtRVSim
Program
Follow fetch
Bp Address Instruction
0x000001ec unknown
0x000001f0 unknown
0x000001f4 unknown
0x000001f8 unknown
0x000001fc unknown
0x00000200 lw x2, ...
0x00000204 sw x2, ...
0x00000208 beq x0, x0,
0x0000020c nop
0x00000210 nop
0x00000214 ebreak
0x00000218 unknown
0x0000021c unknown
0x00000220 unknown
0x00000224 unknown
0x00000228 unknown
0x0000022c unknown
0x00000230 unknown
0x00000234 unknown
0x00000238 unknown
0x0000023c unknown
0x00000240 unknown
Core
simple-lw-sw-ia.S
nop
NONE
Cycles: 0
Stalls: 0
BranchOutcome
Control Unit
MemToReg
MemWrite
MemRead
BranchBxx
BranchJal
BranchJalr
BranchVal
AluControl
AluMul
AluSrc
AuiPC
xor
Branch Jalr
Branch Jalx
PC
0xffffffff
Instruction
RegWrite
Program Memory
rs1
rs2
rd
Registers
Immediate decode
zero
ALU
AluOut
WriteData
Peripherals
Terminal
Data Memory
PC+4
PC
RegWriteData
BranchTarget

Ripes

A graphical processor simulator and assembly editor for the RISC-V ISA

Homepage:
https://github.com/mortbopet/Ripes/

Documentation:
https://github.com/mortbopet/Ripes/tree/master/docs

Easy install:
In Linux as AppImage just do
chmod u+x Ripes-v2.2.6-linux-

File Edit View Help

100 ms

Editor

Processor

Cache

Memory

I/O

Input type: Assembly C Executable code

View mode: Binary Disassembled

```
# This example shows an implementation of the mathematical
# factorial function (! function) to find the factorial value of !7 = 5040.

.data
argument: .word   7
str1:     .string "Factorial value of "
str2:     .string " is "

.text
main:
        lw  a0, argument   # Load argument from static data
        jal ra, fact       # Jump-and-link to the 'fact' label

        # Print the result to console
        mv  a1, a0
        lw  a0, argument
        jal ra, printResult

        # Exit program
        li a7, 10
        ecall

fact:
        addi sp, sp, -16
        sw   ra, 8(sp)
        sw   a0, 0(sp)
        addi t0, a0, -1
        bge  t0, zero, nfact

        addi a0, zero, 1
        addi sp, sp, 16
        jr x1

nfact:
        addi a0, a0, -1                                   EX
        jal  ra, fact                                     ID
        addi t1, a0, 0                                    IF
```

```
    0:      10000517      auipc x10 0x10000
    4:      00052503      lw x10 0 x10
    8:      01c000ef      jal x1 28 <fact>
    c:      00050593      addi x11 x10 0
    10:      10000517      auipc x10 0x10000
    14:      ff052503      lw x10 -16 x10
    18:      04c000ef      jal x1 76 <printResult>
    1c:      00a00893      addi x17 x0 10
    20:      00000073      ecall

00000024 <fact>:
    24:      ff010113      addi x2 x2 -16
    28:      00112423      sw x1 8 x2
    2c:      00a12023      sw x10 0 x2
    30:      fff50293      addi x5 x10 -1
    34:      0002d863      bge x5 x0 16 <nfact>
    38:      00100513      addi x10 x0 1
    3c:      01010113      addi x2 x2 16
    40:      00008067      jalr x0 x1 0

00000044 <nfact>:
    44:      fff50513      addi x10 x10 -1                EX
    48:      fddff0ef      jal x1 -36 <fact>              ID
    4c:      00050313      addi x6 x10 0                  IF
    50:      00012502      lw x10 0 x2
    54:      00812083      lw x1 8 x2
    58:      01010113      addi x2 x2 16
    5c:      02650533      mul x10 x10 x6
    60:      00008067      jalr x0 x1 0

00000064 <printResult>:
    64:      00050293      addi x5 x10 0
    68:      00058313      addi x6 x11 0
    6c:      10000517      auipc x10 0x10000
    70:      f9850513      addi x10 x10 -104
    74:      00400893      addi x17 x0 4
    78:      00000073      ecall
```

GPR

Name	Alias	Value
x0	zero	0x00000000
x1	ra	0x0000004c
x2	sp	0x7fffff90
x3	gp	0x10000000
x4	tp	0x00000000
x5	t0	0x00000001
x6	t1	0x00000000
x7	t2	0x00000000
x8	s0	0x00000000
x9	s1	0x00000000
x10	a0	0x00000002
x11	a1	0x00000000
x12	a2	0x00000000
x13	a3	0x00000000
x14	a4	0x00000000
x15	a5	0x00000000
x16	a6	0x00000000
x17	a7	0x00000000
x18	s2	0x00000000
x19	s3	0x00000000
x20	s4	0x00000000
x21	s5	0x00000000
x22	s6	0x00000000
x23	s7	0x00000000
x24	s8	0x00000000
x25	s9	0x00000000

Hex

Console

```
Factorial value of 7 is 5040
Program exited with code: 0
```

The assembly code

The disassembled code

The registers

sor: 5-stage processor ISA: RV32IM

File Edit View Help

100 ms

Editor

Processor

Cache

Memory

I/O

5-Stage RISC-V Processor

addi x6 x10 0 | jal x1 -36 <fact> | addi x10 x10 -1 | nop (flush) | nop (flush)

4

2

+

PC

enable

Instr. memory

addr

instr

Compressed decoder

IFID

enable

clear

Decode

opcode

Registers

Wr data

Wr idx

R1 idx

R2 idx

Wr En

Reg 1

Reg 2

Imm.

IDEX

enable

clear

Op 1

ALU

Res

Op 2

Branch

Branch taken

EX/ MEM

Addr.

Data in

Wr en

Data memory

Read out

MEM/ WB

Registers

GPR

Name	Alias	Value
x0	zero	0x00000000
x1	ra	0x0000004c
x2	sp	0x7fffff90
x3	gp	0x10000000
x4	tp	0x00000000
x5	t0	0x00000001
x6	t1	0x00000000
x7	t2	0x00000000
x8	s0	0x00000000
x9	s1	0x00000000
x10	a0	0x00000002
x11	a1	0x00000000
x12	a2	0x00000000
x13	a3	0x00000000
x14	a4	0x00000000
x15	a5	0x00000000
x16	a6	0x00000000
x17	a7	0x00000000
x18	s2	0x00000000

Display type: Hex

Console

```
Factorial value of 7 is 5040
Program exited with code: 0
```

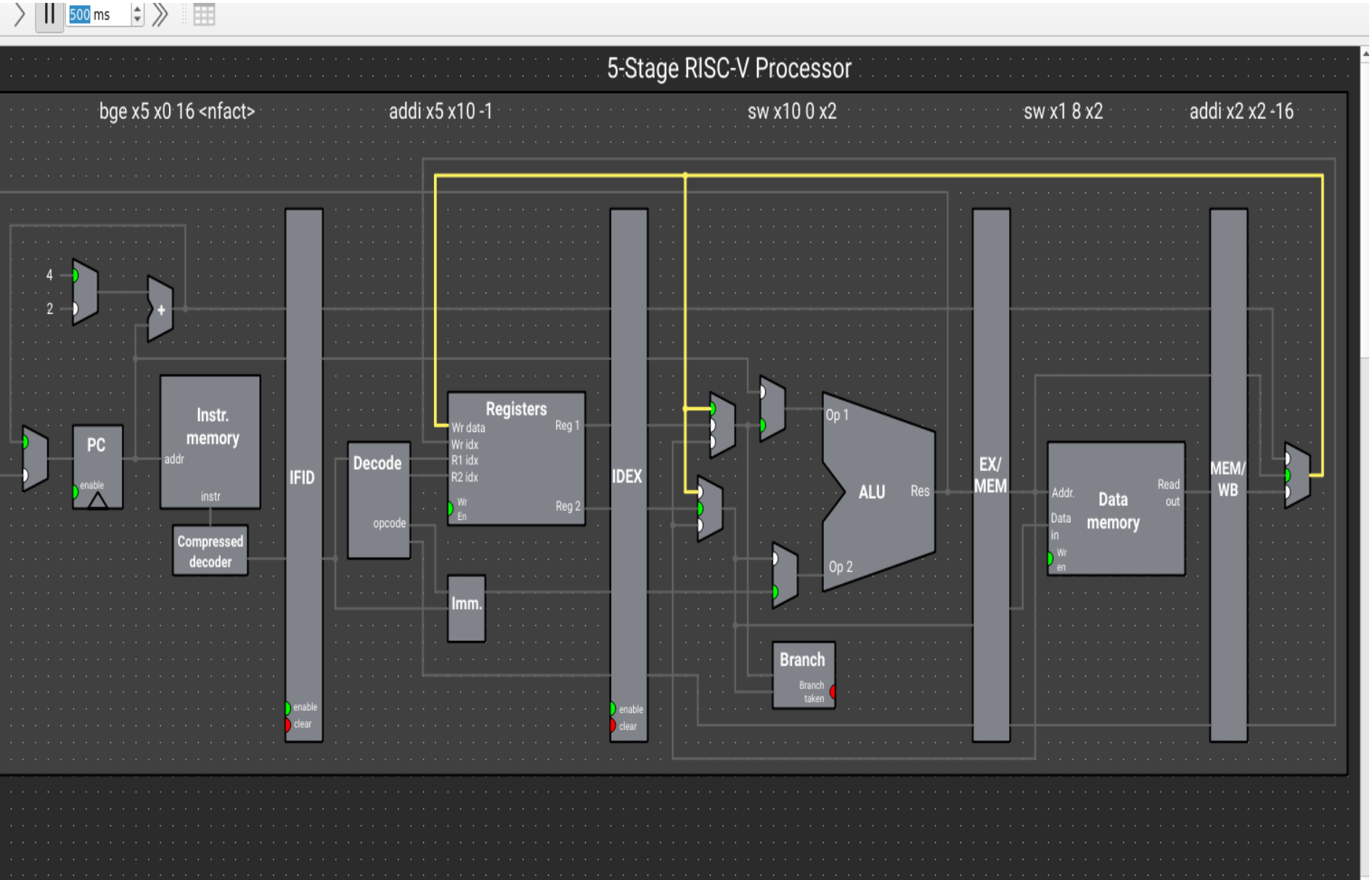
Execution info

Cycles:	69
Instrs. retired:	43
CPI:	1.6
IPC:	0.623
Clock rate:	1.05 Hz

Instruction memory

BP	Addr	Stage	Instruction
	0x3c		addi x2 x2 16
	0x40		jalr x0 x1 0
	0x44	EX	addi x10 x10 -1
	0x48	ID	jal x1 -36 <fact>
	0x4c	IF	addi x6 x10 0
	0x50		lw x10 0 x2

Processor: 5-stage processor ISA: RV32IM


500 ms
5-Stage RISC-V Processor
bge x5 x0 16 <nfact>
addi x5 x10 -1
sw x10 0 x2
sw x1 8 x2
addi x2 x2 -16
4
2
+
PC
enable
Instr.
memory
addr
instr
Compressed
decoder
IFID
enable
clear
Decode
opcode
Registers
Wr data
Wr idx
R1 idx
R2 idx
Wr
En
Reg 1
Reg 2
Imm.
IDEX
enable
clear
Op 1
ALU
Res
Op 2
Branch
Branch
taken
EX/
MEM
Addr.
Data
in
Wr
en
Data
memory
Read
out
MEM/
WB

Venus

https://venus.kvakil.me/

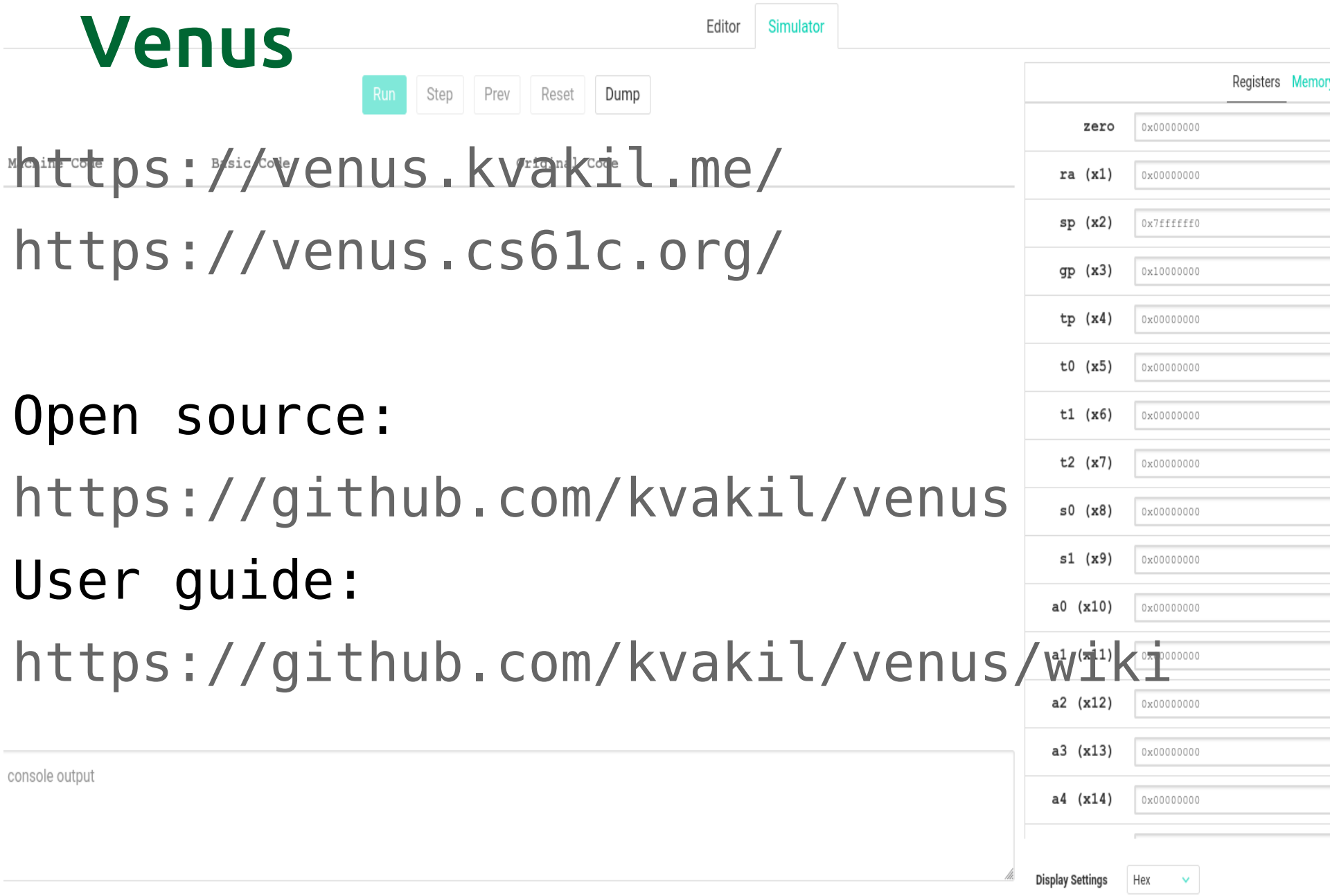
https://venus.cs61c.org/

Open source:

https://github.com/kvakil/venus

User guide:

https://github.com/kvakil/venus/wiki



Venus

The Editor tab:

Active File: null Save Close

```
1 li t0, 3
2 addi t1, t0, 5
```

The Editor tab:

The Simulator tab:

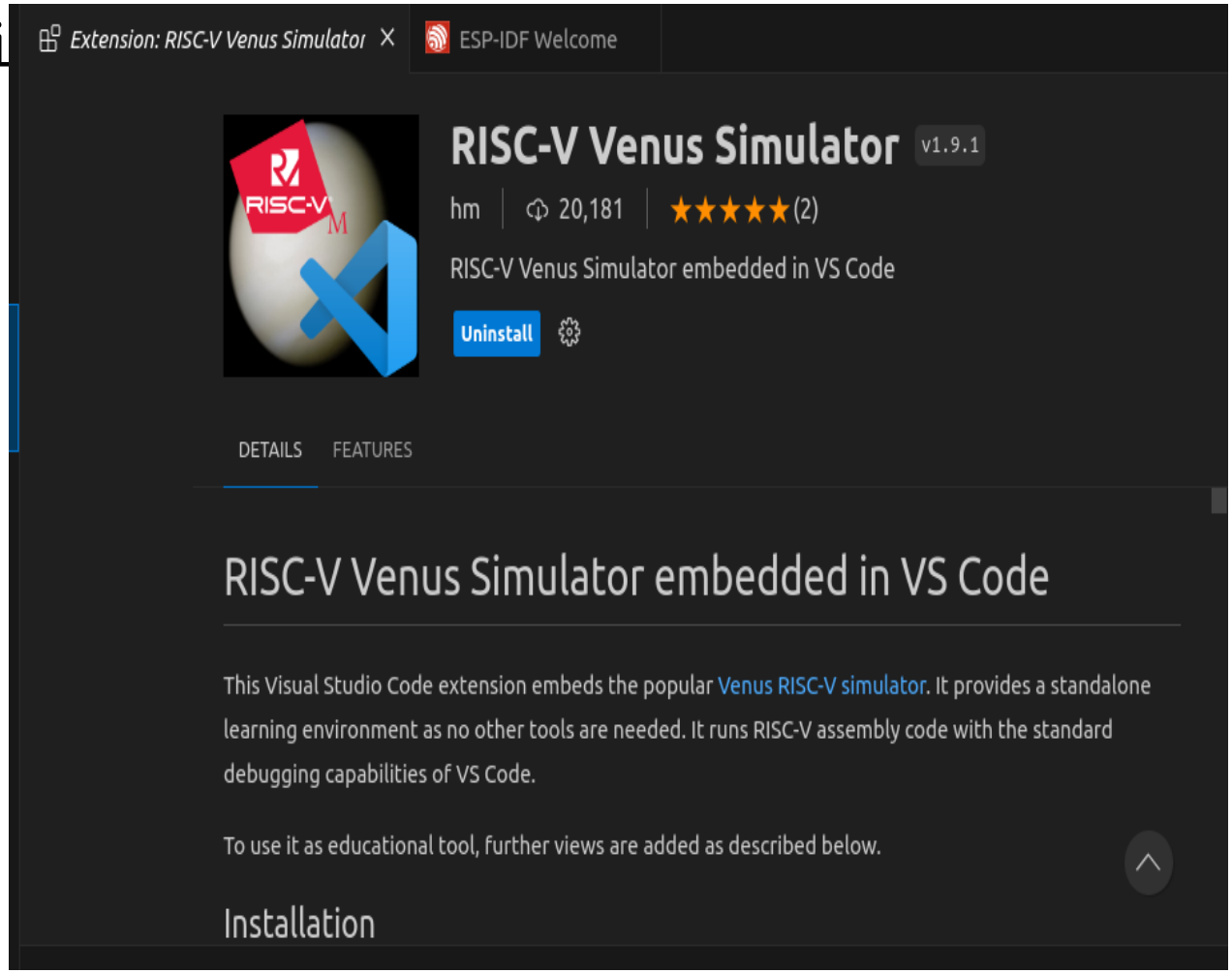
Venus Editor Sim

Run Step Prev Reset Dump Trace Re-assemble from Editor

PC	Machine Code	Basic Code	Original Code
0x0	0x00300293	addi x5 x0 3	li t0, 3
0x4	0x00528313	addi x6 x5 5	addi t1, t0, 5

Venus as a VS-Code Extension

```
ext install
hm.ri
```



emulsiV

https://eseo-tech.github.io/emulsiV/

Select an example | Animation | Speed 10 | Reset | Run | Step | Fetch | Decode | ALU | Compare | Mem/Reg | PC

Memory

Address	0	1	2	3	Instructions
00000000	93	00	00	02	addi x1, x0, 32
00000004	37	01	00	c0	lui x2, 0xc0000
00000008	83	c1	00	00	lbu x3, 0(x1)
0000000c	63	88	01	00	beq x3, x0, +16
00000010	23	00	31	00	sb x3, 0(x2)
00000014	93	80	10	00	addi x1, x1, 1
00000018	6f	f0	1f	ff	jal x0, -16
0000001c	6f	00	00	00	jal x0, 0
00000020	48	65	6c	6c	–
00000024	6f	00	00	00	jal x0, 0
00000028	00	00	00	00	–
0000002c	00	00	00	00	–
00000030	00	00	00	00	–
00000034	00	00	00	00	–
00000038	00	00	00	00	–
0000003c	00	00	00	00	–

Program counter

pc	00000000
mepc	00000000
pc+4	00000004

ALU

op	–
a	–
b	–
r	–

General-purpose regs

x0	00000000
x1	00000000
x2	00000000
x3	00000000
x4	00000000
x5	00000000
x6	00000000
x7	00000000
x8	00000000
x9	00000000
x10	00000000
x11	00000000
x12	00000000
x13	00000000
x14	00000000
x15	00000000
x16	00000000
x17	00000000

x[rs1] x[rs2] x[rd]

Bus

addr	–
data	–
irq	false

Instruction reg.

instr	–
fn	–
rs1	–
rs2	–
rd	–
imm	–

Comparator

op	–
a	–
b	–
taken	–

Text I/O

	0	1
b0000000 (ctrl, data)	00	00

c0000000 (data) 00

General-purpose I/O

	0	1	2	3
d0000000 (dir)	ff	ff	ff	ff
d0000004 (ien)	00	00	00	00
d0000008 (rev)	00	00	00	00
d000000c (fev)	00	00	00	00
d0000010 (val)	00	00	00	00

0
1

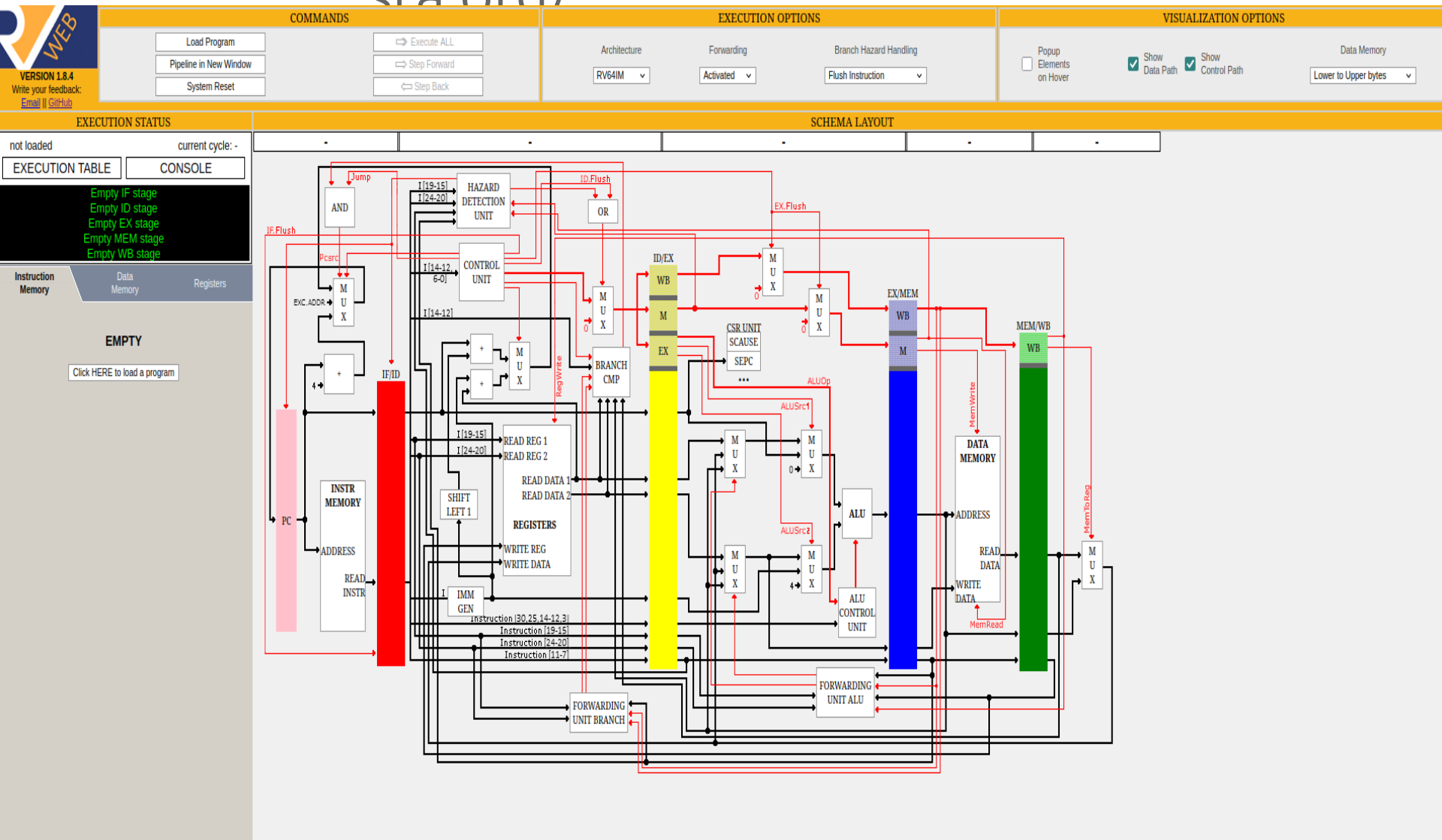
Bitmap output (00000C00–00000FFF)

About emulsiV
Read the license
Report an issue
Get the source code

eseo INNOVATION MAKES SENSE

WebRISC-V

https://webriscv.altervista.org/



Interactive RISC-V Simulator

RISC-V Simulator | Editor | Datapath | Memory | Cache | 1000ms | Show Output 6

https://riscv-simulator-five.vercel.app

Code Editor

Import | Export | Quicksort | Assemble

```
# Quicksort Algorithm Implementation
# This program demonstrates the quicksort algorithm on an array of integers

.data
array:    .word 64, 34, 25, 12, 22, 11, 90, 45    # Array to be sorted
length:   .word 8                                 # Length of the array
msg1:     .asciz "Original array: "
msg2:     .asciz "\nSorted array: "
space:    .asciz " "

.text
.globl main
main:
    # Print original array message
    la a0, msg1
    li a7, 4                    # Print string
    ecall

    # Print original array
    la s0, array                # s0 = array address
    la s1, length               # s1 = length address
    lw s1, 0(s1)                # s1 = length value
    jal ra, print_array

    # Call quicksort
    la a0, array                # a0 = array address
    li a1, 0                    # a1 = start index (0)
    addi a2, s1, -1             # a2 = end index (length-1)
    jal ra, quicksort

    # Print sorted array message
    la a0, msg2
    li a7, 4                    # Print string
    ecall

    # Print sorted array
    la s0, array                # s0 = array address
    jal ra, print_array

    # Exit program
    li a7, 10                   # Exit
```

Result

Export Code | Symbols

Address	Code	Basic	Source
0x00000064	0x02928863	beq t0, s1, 0x00000094	beq t0, s1, print_done
0x00000068	0x00229313	slli t1, t0, 2	slli t1, t0, 2
0x0000006c	0x00640333	add t1, s0, t1	add t1, s0, t1
0x00000070	0x00032503	lw a0, 0(t1)	lw a0, 0(t1)
0x00000074	0x00100893	addi a7, x0, 1	li a7, 1
0x00000078	0x00000073	ecall	ecall
0x0000007c	0x10000537	lui a0, 0x10000	la a0, space
0x00000080	0x04650513	addi a0, a0, 70	la a0, space
0x000000…	0x00400893	addi a7, x0, 4	li a7, 4
0x00000088	0x00000073	ecall	ecall
0x0000008c	0x00128293	addi t0, t0, 1	addi t0, t0, 1
0x00000090	0xfd5ff06f	jal x0, 0x00000064	j print_loop
<print_done>:			
0x00000094	0x00008067	jalr zero, ra, 0	jalr zero, ra, 0
<quicksort>:			
0x00000098	0x06c5d463	bge a1, a2, 0x00000100	bge a1, a2, quicksort_done
0x0000009c	0xfec10113	addi sp, sp, -20	addi sp, sp, -20
0x000000a0	0x00112823	sw ra, 16(sp)	sw ra, 16(sp)
0x000000a4	0x00812623	sw s0, 12(sp)	sw s0, 12(sp)
0x000000a8	0x00912423	sw s1, 8(sp)	sw s1, 8(sp)
0x000000ac	0x01212223	sw s2, 4(sp)	sw s2, 4(sp)
0x000000b0	0x01312023	sw s3, 0(sp)	sw s3, 0(sp)
0x000000b4	0x00050413	addi s0, a0, 0	mv s0, a0

Register

Hexadecima

Name	Number	Value
zero	0	0x00000000
ra	1	0x00000028
sp	2	0x7ffffff0
gp	3	0x10000000
tp	4	0x00000000
t0	5	0x00000002
t1	6	0x10000008
t2	7	0x00000000
s0	8	0x10000000
s1	9	0x00000008
a0	10	0x10000046
a1	11	0x00000000
a2	12	0x00000000
a3	13	0x00000000
a4	14	0x00000000
a5	15	0x00000000
a6	16	0x00000000
a7	17	0x00000001
s2	18	0x00000000
s3	19	0x00000000
s4	20	0x00000000
s5	21	0x00000000
s6	22	0x00000000
s7	23	0x00000000
s8	24	0x00000000
s9	25	0x00000000
s10	26	0x00000000
s11	27	0x00000000
t3	28	0x00000000
t4	29	0x00000000
t5	30	0x00000000
t6	31	0x00000000

RISC-V Functional Simulator

https://risc-v-cpu-visualizer.vercel.app/simulator

RISC-V

ycle Simulator | Five-Stage Simulator | Converter | Contact Us

RISC-V Functional Simulator

Input your RISC-V code here:

```
.main:
    # write your code here

.end:
    ebreak
```

Reset | Run / Continue | Fast Run | Step | Initialize Memory

Execution Log

How to Use the Simulator

Registers

Init Value	Register	Decimal	Hex	Binary
0	x0 (zero)	0	0x00000000	00000000000000000000000000000000
0	x1 (ra)	0	0x00000000	00000000000000000000000000000000
0	x2 (sp)	0	0x00000000	00000000000000000000000000000000
0	x3 (gp)	0	0x00000000	00000000000000000000000000000000
0	x4 (tp)	0	0x00000000	00000000000000000000000000000000
0	x5 (t0)	0	0x00000000	00000000000000000000000000000000
0	x6 (t1)	0	0x00000000	00000000000000000000000000000000
0	x7 (t2)	0	0x00000000	00000000000000000000000000000000
0	x8 (s0/fp)	0	0x00000000	00000000000000000000000000000000
0	x9 (s1)	0	0x00000000	00000000000000000000000000000000
0	x10 (a0)	0	0x00000000	00000000000000000000000000000000
0	x11 (a1)	0	0x00000000	00000000000000000000000000000000
0	x12 (a2)	0	0x00000000	00000000000000000000000000000000

ret

https://ret.futo.org/riscv/

RISC-V ▾ Assemble Disassemble Assemble and Run Format Hex

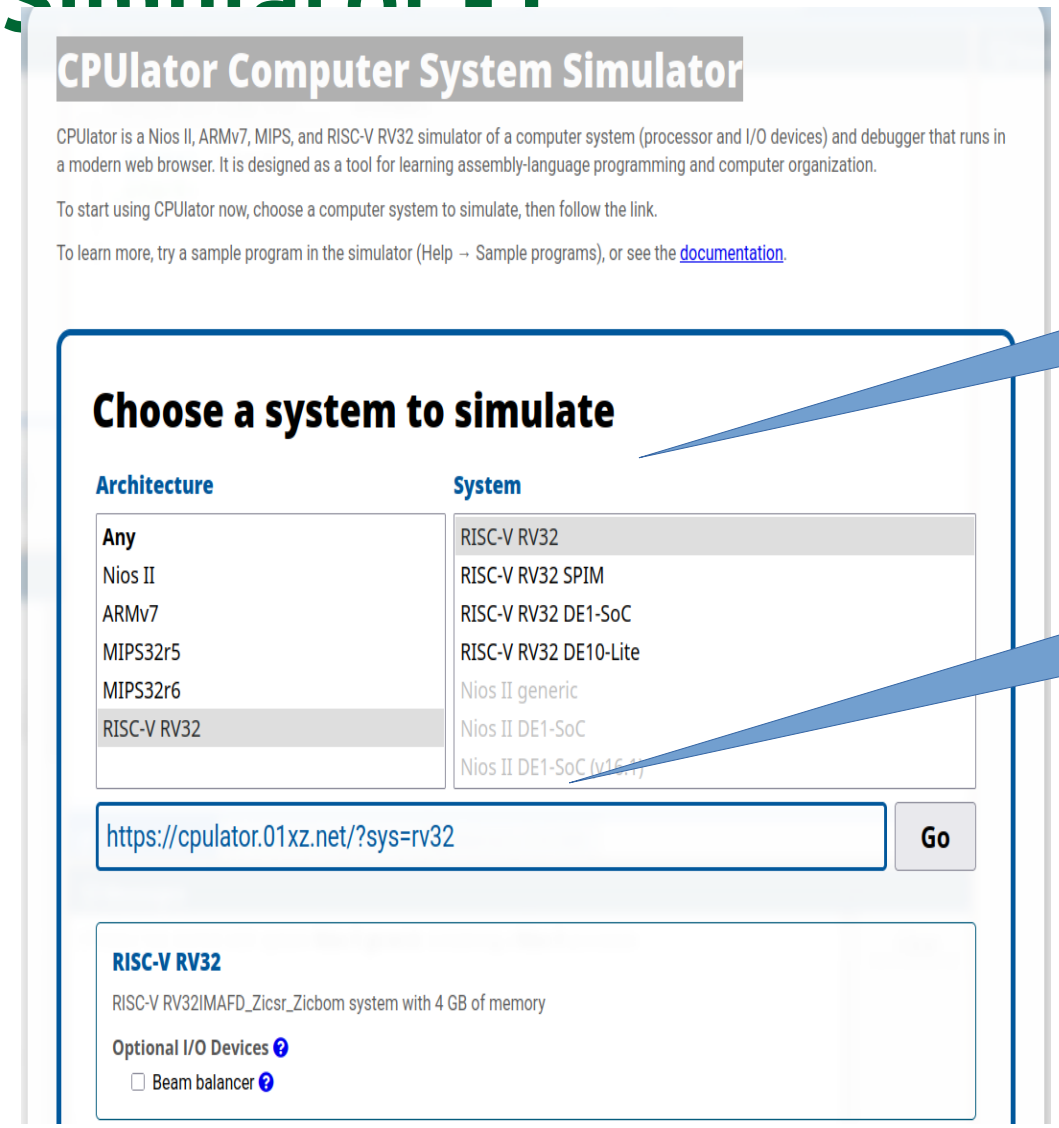
ISC-V Options Base address: 0x0

```
la a0, string
li a2, 0x9000000 // UART_DR, write here to print characters

top:
lbu a1, 0(a0)
sw a1, 0(a2)
addi a0, a0, 1
bne a1, zero, top

j skip
string: .string "Hello World"
.align 1
skip:
```

Save as binary file

```
Ret - Reverse-Engineering Tool
Click the top left button to switch architecture.
Click 'Examples' to get started.
```

CPUlator Computer System Simulator - I



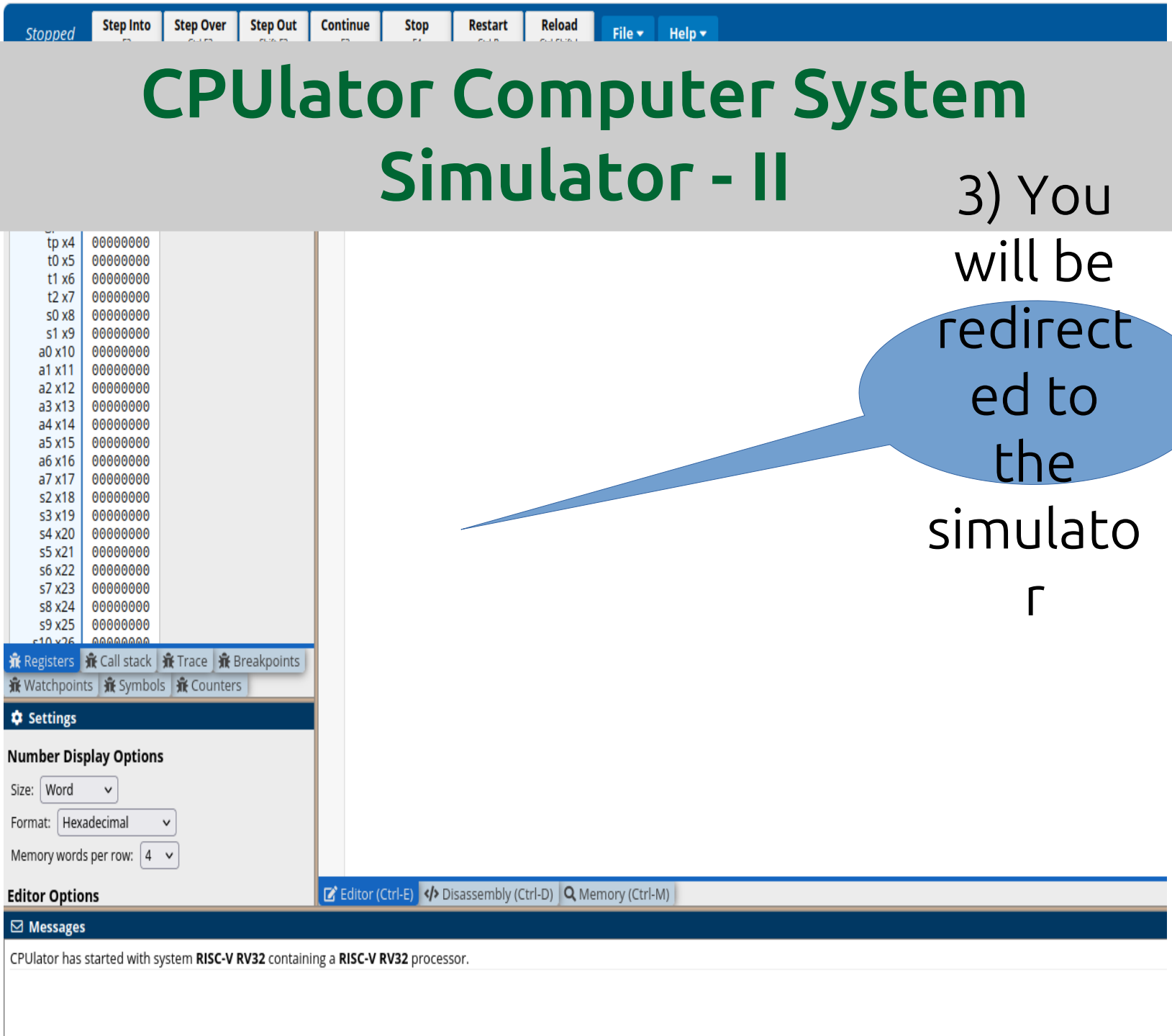
1) Select the system

2) Check the link

https://cpulator.01xz.net/

CPUlator Computer System Simulator - II

3) You will be redirected to the simulator



QEMU

RISC-V 64 Bit Emulator

https://www.qemu.org/download/#windows

Build from the source:

git clone git clone

https://gitlab.com/qemu-project/qemu.git

+ The RISC-V Toolchain

Spike

https://github.com/riscv-soft

Spike is the official or golden-reference–style RISC-V Instruction Set Simulator (ISS)

```
telzur@TUF:~/science/Teaching/CPU/SW/spike/guy$
riscv64-unknown-elf-gcc -o hello ./hello.c
```

```
telzur@TUF:~/science/Teaching/CPU/SW/spike/guy$
sp
He
```

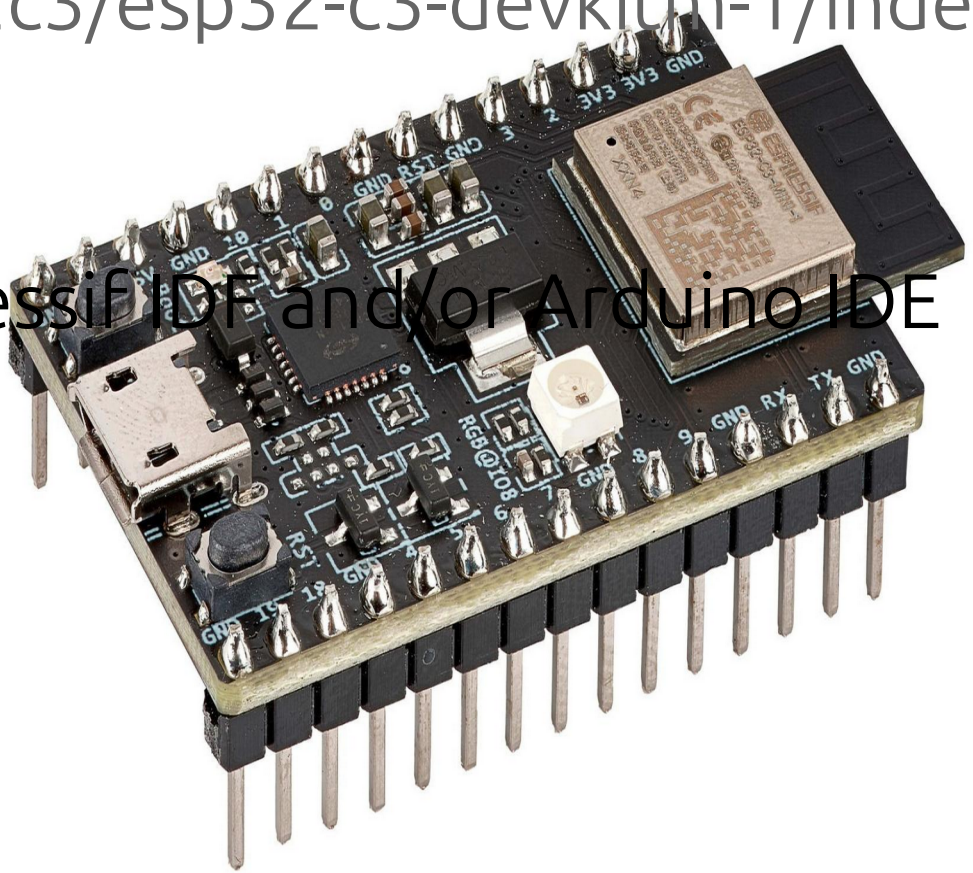
Spike is best suited for:

- ✔️ Reference ISA- level correctness
- ✔️ Early software development (bare- metal & small programs)
- ✔️ Debugging and inspecting architectural state
- ✔️ Verification and compliance testing
- ✔️ Exploring ISA extensions and configurations
- ✔️ Running RISC- V code when hardware is not available

Get a RISC-V core!

https://docs.espressif.com/projects/esp-dev-kits/en/latest/esp32c3/esp32-c3-devkitm-1/index.html

Install Espressif IDF and/or Arduino IDE



Part 3: Additional simulators

Compiler Explorer

Reorder Buffer

Tomasulo Algorithm

Compiler Explorer

https://

C source #1

```c
int bar(int,int);

int foo(int a, int b){
    if (a <= b)
      a = 0;
    else
      b = 0;
    return bar(a, b);
}
```

mips gcc 11.2.0 (Editor #1)

mips gcc 11.2.0 -O3 -march=mips4 -fno-delayed-branch

```mips
foo:
        slt     $2,$5,$4
        movn    $5,$0,$2
        movz    $4,$0,$2
        j       bar
        nop
```

x86-64 clang 14.0.0 (Editor #1)

x86-64 clang 14.0.0 -O3 -fomit-frame-pointer

```x86
foo:                                    # @foo
        xor     eax, eax
        cmp     edi, esi
        cmovle  edi, eax
        cmovg   esi, eax
        jmp     bar                     # TAILCALL
```

Output of mips gcc 11.2.0 (Compiler #1)

```
Compiler returned: 0
```

Reorder Buffer

https://www.ecs.umass.edu/ece/koren/architecture/ROB/rob_simulator.htm

האתר הזה חצי

חי

(האתר לא

מאובטח SSL)

Simulation Control

+1 Clock Cycle

+5 Clock Cycles

current clock:

Simulation Settings

START SIMULATION

CONFIGURATION

HELP

Reorder Buffer Simulator

Configuration Window

Number of **ROB** entries: 6

FUs	Number	Ex. Cycles	# of RSs
FP-Adder	1	2	3
FP-Multiplier	1	10	1
FP-Divider	1	40	1

Number of **CDBs**: 1

Number of Load Buffers: 2

Latency of Load: 2 clock cycle(s)

change # of Instructions to 6

current # of Instructions: 6

Examples

Instructions

#	Opcode	Operands			causes an exception
		Dest.	Op1	Op2	
1	L.D	F6	F0	F0	
2	L.D	F2	F0	F0	
3	MUL.D	F0	F2	F4	
4	SUB.D	F8	F6	F2	
5	DIV.D	F10	F0	F6	
6	ADD.D	F6	F8	F2	

Tomasulo Algorithm

http://nathantypanski.github.io/tomasulo-simulator/

Tomasulo algorithm simulator (protoype)

This simulates Tomasulo's algorithm for a floating-point MIPS-like instruction pipeline, demonstrating out-of-order execution. The source is on GitHub.

Click instructions on the right to issue and execute them. **Instructions will only execute if all of their data dependencies have been resolved**, but they may issue in any order (though at least issuing them in order is recommended). Currently, loads have two-step execution and still require a writeback cycle. Regs[x] is the value at location x from the register file.

Color codes are as follows: *destination* *source* *occupied source* *occupied destination*.

Reservation stations

Name	Busy	Op	Vj	Vk	Qj	Qk	A	Result
Load0	false							
Load1	false							
Add0	false							
Add1	false							
Add2	false							
Mult0	false							
Mult1	false							
Store0	false							
Store1	false							

Instruction Status

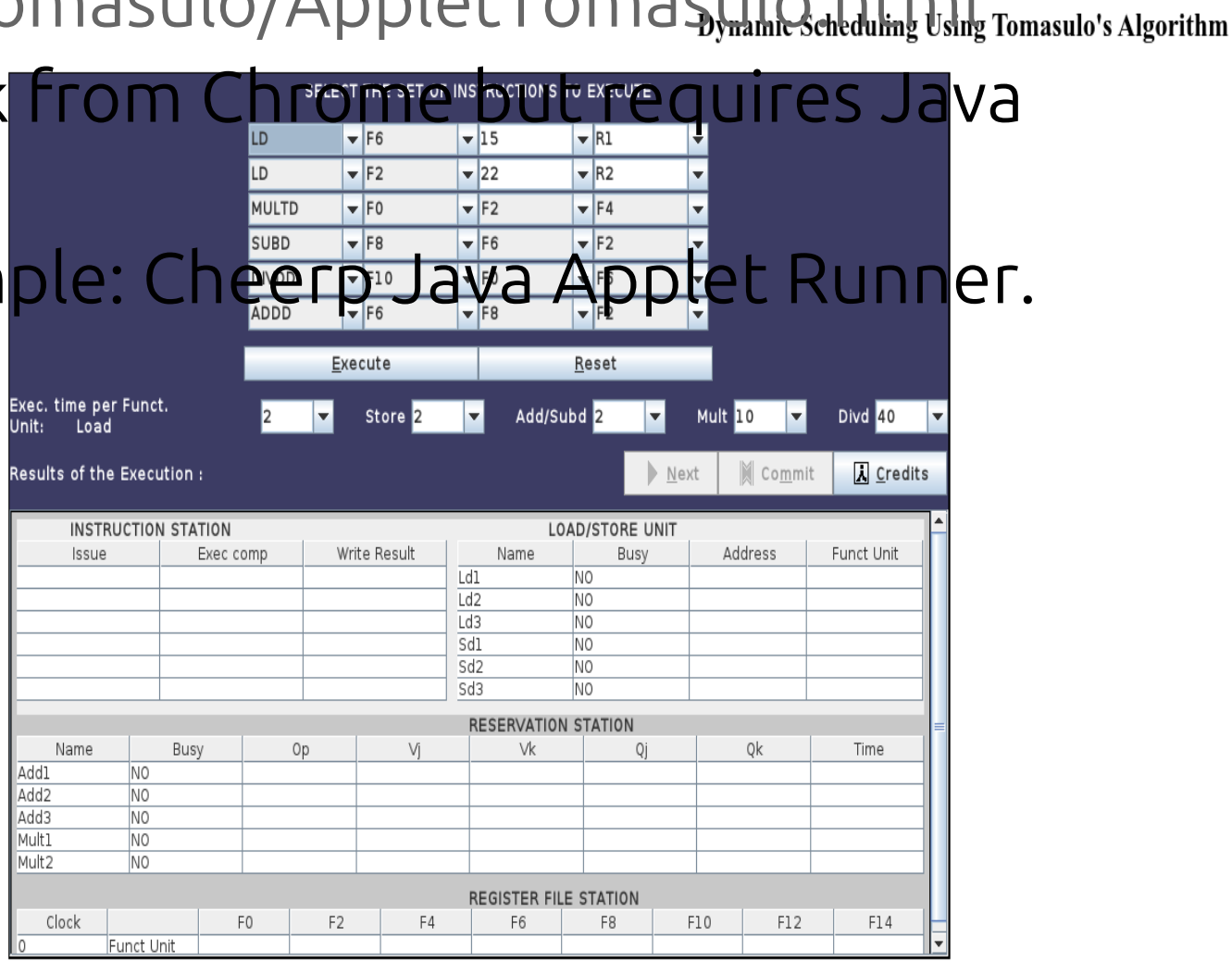
Instruction	Issue	Execute	Write result
L.D F6,32(R2)			
L.D F2,44(R3)			
MUL.D F0,F2,F4			
ADD.D F10,F12,F8			
SUB.D F8,F2,F6			
DIV.D F10,F0,F6			
ADD.D F6,F8,F2			
S.D F6,32(R2)			

Tomasulo Algorithm

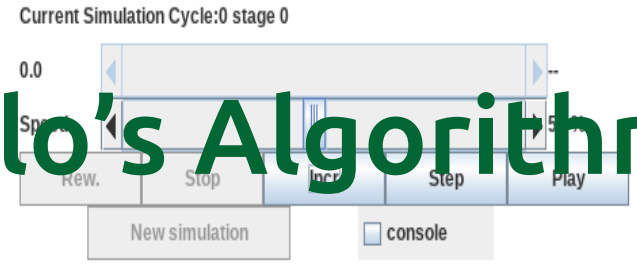
https://www.ecs.umass.edu/ece/koren/architecture/Tomasulo/AppletTomasulo.html

Can work from Chrome but requires Java Plug-in.

For example: Cheerp Java Applet Runner.



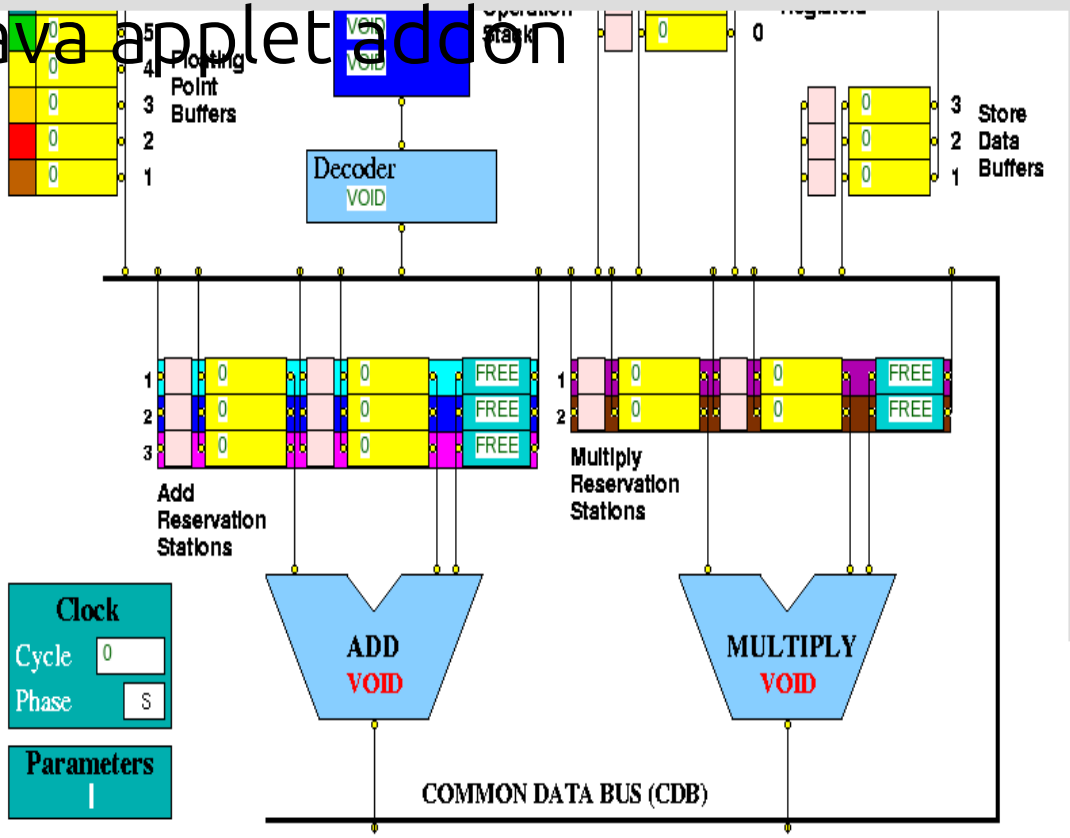
Tomasulo’s Algorithm V1.5



https://www.icsa.inf.ed.ac.uk/cgi-bin/hase/ident.pl?Tomasulo

works on Chrome with CheerpJ

java applet addon



לא פועל בגלל מניעת תמיכה בג’אווה בדפדפנים מודרניים.

FreeSS

סימולטור חדש!

WCAE'25 - Workshop on Computer Architecture Education, June 21--25, 2025, Tokyo, Japan

arXiv:2506.07665v1 [cs.AR] 9 Jun 2025

FREESS: An Educational Simulator of a RISC-V-Inspired Superscalar Processor Based on Tomasulo's Algorithm

Roberto Giorgi
giorgi@unisi.it
University of Siena
Siena, Italy

Abstract

FREESS is a free, interactive simulator that illustrates instruction-level parallelism in a RISC-V-inspired superscalar processor. Based on an extended version of Tomasulo's algorithm, FREESS is intended as a hands-on educational tool for Advanced Computer Architecture courses. It enables students to explore dynamic, out-of-order instruction execution, emphasizing how instructions are issued as soon as their operands become available.

The simulator models key microarchitectural components, including the Instruction Window (IW), Reorder Buffer (ROB), Register Map (RM), Free Pool (FP), and Load/Store Queues. FREESS allows users to dynamically configure runtime parameters, such as the superscalar issue width, functional unit types and latencies, and the sizes of architectural buffers and queues.

To simplify learning, the simulator uses a minimal instruction set inspired by RISC-V (ADD, ADDI, BEQ, BNE, LW, MUL, SW), which is sufficient to demonstrate key pipeline stages: fetch, register renaming, out-of-order dispatch, execution, completion, commit, speculative branching, and memory access. FREESS includes three step-by-step, illustrated examples that visually demonstrate how multiple instructions can be issued and executed in parallel within a single cycle. Being open source, FREESS encourages students and educators to experiment freely by writing and analyzing their own instruction-level programs and superscalar architectures.

CCS Concepts

• **Computer systems organization** → **Superscalar architectures**; *Reduced instruction set computing*; • **General and reference** → *Evaluation*; • **Applied computing** → **Computer-assisted instruction**.

Keywords

Superscalar, Simulator, RISC-V, Tomasulo

ACM Reference Format:

ional Simulator of a RISC-V-Inspired
sulo's Algorithm. In *Proceedings of*
cation (WCAE '25). ACM, New York,
XXXX.XXXXXXX

l all or part of this work for personal or
d that copies are not made or distributed
opies bear this notice and the full citation
s of this work owned by others than the
credit is permitted. To copy otherwise, or
to lists, requires prior specific permission
issions@acm.org.

s). Publication rights licensed to ACM.

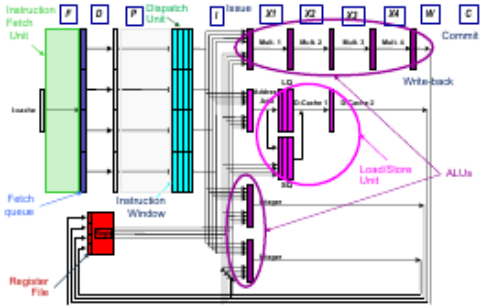


Figure 1: Structure of an Out-Of-Order Processor in FREESS.

1 Introduction

Most modern microprocessors in medium- and high-end systems adopt *superscalar* architectures, building upon dynamic scheduling mechanisms such as *Thornton's scoreboard* [10] and *Tomasulo's algorithm* [11]. In this paper, we focus on Tomasulo's to illustrate how machine instructions can be executed *in parallel*, transparently to the user. These techniques are commonly known as *dynamic scheduling*, *out-of-order execution*, or *restricted dataflow* [4].

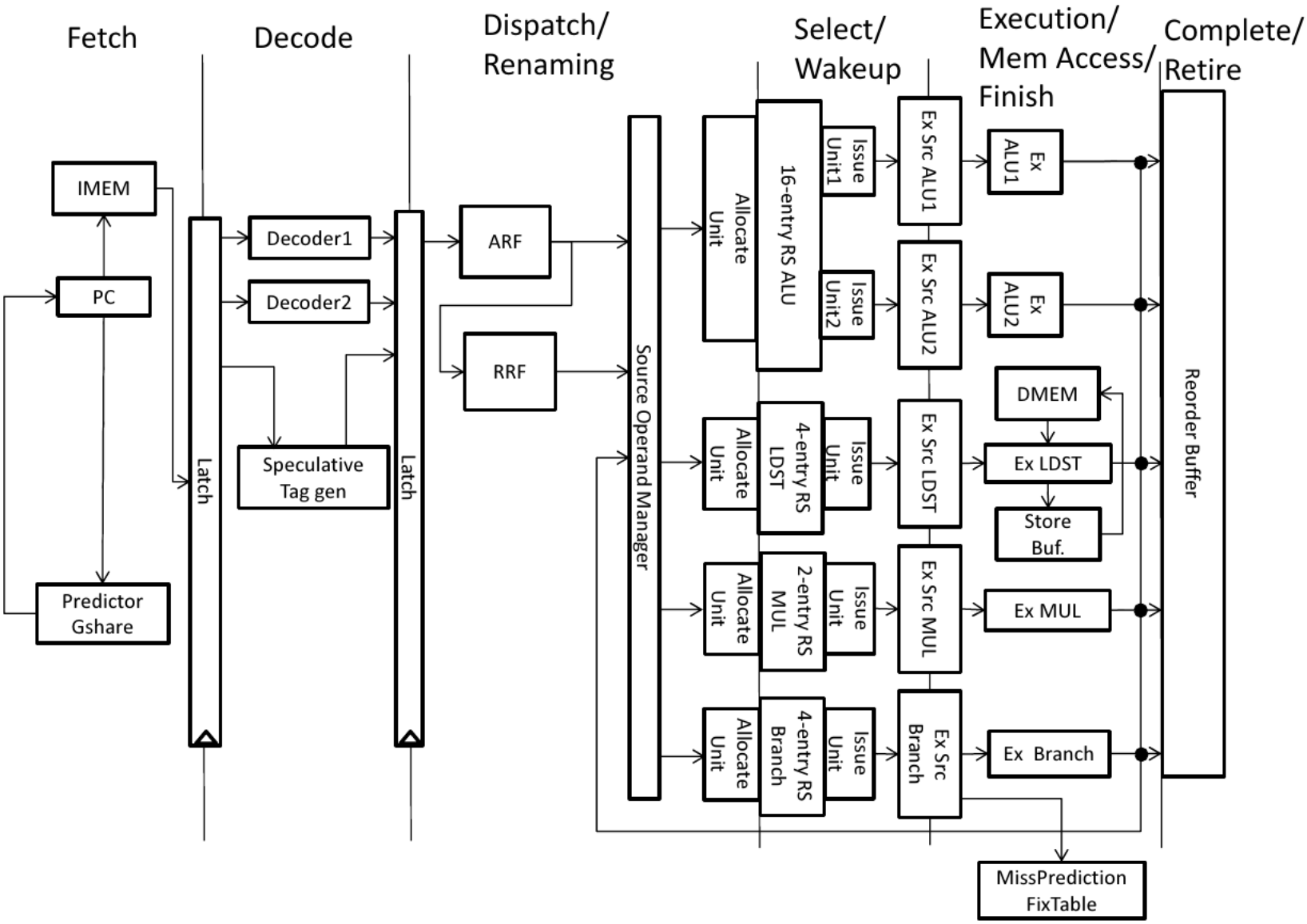
This topic is therefore a cornerstone for *Advanced Computer Architecture* courses, where *Instruction-Level Parallelism (ILP)* plays a key role in achieving *High-Performance Computing* and in accelerating *single-threaded* execution.

Superscalar architectures provide an elegant hardware-based solution to the problem of tracking control and data dependencies among instructions. While thread– and data-level parallelism are useful, executing multiple instructions per cycle can also significantly boost performance. Normally, one instruction takes several cycles to complete. A first performance improvement comes from *pipelining*, but a single pipeline can still issue at most one instruction per cycle, also known as *Flynn's bottleneck*. This limitation is overcome by issuing multiple instructions to independent functional units. This goal is achieved in *superscalar processors* (via hardware scheduling) or *VLIW architectures* (via software scheduling).

The superscalar approach based on Tomasulo's extended algorithm augments the pipeline with several hardware structures that dynamically track data dependencies and execute instructions according to the *dataflow* principle: firing them as soon as their operands are ready. The most relevant structures involved in the tracking dependencies include: *Physical Registers (Px)*, the *Free Pool (FP)*, the *Register Map (RM)*, the *Instruction Window (IW)*, the *Reorder Buffer (ROB)*, and the *Load/Store Queues (LSQs)* (see Fig. 1).

paper:https://arxiv.org/abs/2506.07665
code:
https://github.com/robgiorgi/freess

RIDECORE: RIsc-v Dynamic Exection CORE



RIDECORE's microarchitecture is based on "Modern Processor Design: Fundamentals of Superscalar Processors"
https://github.com/ridecore/ridecore/tree/master

References

Roberto Giorgi, “FREESS: An Educational Simulator of a RISC-V-Inspired Superscalar Processor Based on Tomasulo’s Algorithm”, Proceedings of per cycle, also known as Flynn’s bottleneck. This limitation is over-Workshop on Computer Architecture Education (WCAE ’25).

שימו לב – 2 השקפים הבאים:
2 עמודי תקציר הוראות RISC-V
אשר יצורפו לשאלון הבחינה.

(ההמרה למצגת עיוותה אותם.
אל דאגה. במבחן זה יהיה כמו
במקור. בכל מקרה,
דפים אלה ניתנים להורדה
מהמודל).

Free & Open RISC-V Reference Card ①

Base Integer Instructions: RV32I, RV64I, and RV128I

Category	Name	Fmt	RV32I Base	+RV{64,128}
Loads	Load Byte	I	`LB rd,rs1,imm`	
	Load Halfword	I	`LH rd,rs1,imm`	
	Load Word	I	`LW rd,rs1,imm`	`L{D|Q} rd,rs1,imm`
	Load Byte Unsigned	I	`LBU rd,rs1,imm`	
	Load Half Unsigned	I	`LHU rd,rs1,imm`	`L{W|D}U rd,rs1,imm`
Stores	Store Byte	S	`SB rs1,rs2,imm`	
	Store Halfword	S	`SH rs1,rs2,imm`	
	Store Word	S	`SW rs1,rs2,imm`	`S{D|Q} rs1,rs2,imm`
Shifts	Shift Left	R	`SLL rd,rs1,rs2`	`SLL{W|D} rd,rs1,rs2`
	Shift Left Immediate	I	`SLLI rd,rs1,shamt`	`SLLI{W|D} rd,rs1,shamt`
	Shift Right	R	`SRL rd,rs1,rs2`	`SRL{W|D} rd,rs1,rs2`
	Shift Right Immediate	I	`SRLI rd,rs1,shamt`	`SRLI{W|D} rd,rs1,shamt`
	Shift Right Arithmetic	R	`SRA rd,rs1,rs2`	`SRA{W|D} rd,rs1,rs2`
	Shift Right Arith Imm	I	`SRAI rd,rs1,shamt`	`SRAI{W|D} rd,rs1,shamt`
Arithmetic	ADD	R	`ADD rd,rs1,rs2`	`ADD{W|D} rd,rs1,rs2`
	ADD Immediate	I	`ADDI rd,rs1,imm`	`ADDI{W|D} rd,rs1,imm`
	SUBtract	R	`SUB rd,rs1,rs2`	`SUB{W|D} rd,rs1,rs2`
	Load Upper Imm	U	`LUI rd,imm`	
	Add Upper Imm to PC	U	`AUIPC rd,imm`	
Logical	XOR	R	`XOR rd,rs1,rs2`	
	XOR Immediate	I	`XORI rd,rs1,imm`	
	OR	R	`OR rd,rs1,rs2`	
	OR Immediate	I	`ORI rd,rs1,imm`	
	AND	R	`AND rd,rs1,rs2`	
	AND Immediate	I	`ANDI rd,rs1,imm`	
Compare	Set <	R	`SLT rd,rs1,rs2`	
	Set < Immediate	I	`SLTI rd,rs1,imm`	
	Set < Unsigned	R	`SLTU rd,rs1,rs2`	
	Set < Imm Unsigned	I	`SLTIU rd,rs1,imm`	
Branches	Branch =	SB	`BEQ rs1,rs2,imm`	
	Branch ≠	SB	`BNE rs1,rs2,imm`	
	Branch <	SB	`BLT rs1,rs2,imm`	
	Branch ≥	SB	`BGE rs1,rs2,imm`	
	Branch < Unsigned	SB	`BLTU rs1,rs2,imm`	
	Branch ≥ Unsigned	SB	`BGEU rs1,rs2,imm`	
Jump & Link	J&L	UJ	`JAL rd,imm`	
	Jump & Link Register	UJ	`JALR rd,rs1,imm`	
Synch	Synch thread	I	`FENCE`	
	Synch Instr & Data	I	`FENCE.I`	
System	System CALL	I	`SCALL`	
	System BREAK	I	`SBREAK`	
Counters	ReaD CYCLE	I	`RDCYCLE rd`	
	ReaD CYCLE upper Half	I	`RDCYCLEH rd`	
	ReaD TIME	I	`RDTIME rd`	
	ReaD TIME upper Half	I	`RDTIMEH rd`	
	ReaD INSTR RETired	I	`RDINSTRET rd`	
	ReaD INSTR upper Half	I	`RDINSTRETH rd`	

RV Privileged Instructions

Category	Name	RV mnemonic
CSR Access	Atomic R/W	`CSRRW rd,csr,rs1`
	Atomic Read & Set Bit	`CSRRS rd,csr,rs1`
	Atomic Read & Clear Bit	`CSRRC rd,csr,rs1`
	Atomic R/W Imm	`CSRRWI rd,csr,imm`
	Atomic Read & Set Bit Imm	`CSRRSI rd,csr,imm`
	Atomic Read & Clear Bit Imm	`CSRRCI rd,csr,imm`
Change Level	Env. Call	`ECALL`
	Environment Breakpoint	`EBREAK`
	Environment Return	`ERET`
Trap Redirect	to Supervisor	`MRTS`
	Redirect Trap to Hypervisor	`MRTH`
	Hypervisor Trap to Supervisor	`HRTS`
Interrupt	Wait for Interrupt	`WFI`
MMU	Supervisor FENCE	`SFENCE.VM rs1`

Optional Compressed (16-bit) Instruction Extension: RVC

Category	Name	Fmt	RVC	RVI equivalent
Loads	Load Word	CL	`C.LW rd',rs1',imm`	`LW rd',rs1',imm*4`
	Load Word SP	CI	`C.LWSP rd,imm`	`LW rd,sp,imm*4`
	Load Double	CL	`C.LD rd',rs1',imm`	`LD rd',rs1',imm*8`
	Load Double SP	CI	`C.LDSP rd,imm`	`LD rd,sp,imm*8`
	Load Quad	CL	`C.LQ rd',rs1',imm`	`LQ rd',rs1',imm*16`
	Load Quad SP	CI	`C.LQSP rd,imm`	`LQ rd,sp,imm*16`
Stores	Store Word	CS	`C.SW rs1',rs2',imm`	`SW rs1',rs2',imm*4`
	Store Word SP	CSS	`C.SWSP rs2,imm`	`SW rs2,sp,imm*4`
	Store Double	CS	`C.SD rs1',rs2',imm`	`SD rs1',rs2',imm*8`
	Store Double SP	CSS	`C.SDSP rs2,imm`	`SD rs2,sp,imm*8`
	Store Quad	CS	`C.SQ rs1',rs2',imm`	`SQ rs1',rs2',imm*16`
	Store Quad SP	CSS	`C.SQSP rs2,imm`	`SQ rs2,sp,imm*16`
Arithmetic	ADD	CR	`C.ADD rd,rs1`	`ADD rd,rd,rs1`
	ADD Word	CR	`C.ADDW rd,rs1`	`ADDW rd,rd,imm`
	ADD Immediate	CI	`C.ADDI rd,imm`	`ADDI rd,rd,imm`
	ADD Word Imm	CI	`C.ADDIW rd,imm`	`ADDIW rd,rd,imm`
	ADD SP Imm * 16	CI	`C.ADDI16SP x0,imm`	`ADDI sp,sp,imm*16`
	ADD SP Imm * 4	CIW	`C.ADDI4SPN rd',imm`	`ADDI rd',sp,imm*4`
	Load Immediate	CI	`C.LI rd,imm`	`ADDI rd,x0,imm`
	Load Upper Imm	CI	`C.LUI rd,imm`	`LUI rd,imm`
	MoVe	CR	`C.MV rd,rs1`	`ADD rd,rs1,x0`
	SUB	CR	`C.SUB rd,rs1`	`SUB rd,rd,rs1`
Shifts	Shift Left Imm	CI	`C.SLLI rd,imm`	`SLLI rd,rd,imm`
Branches	Branch=0	CB	`C.BEQZ rs1',imm`	`BEQ rs1',x0,imm`
	Branch≠0	CB	`C.BNEZ rs1',imm`	`BNE rs1',x0,imm`
Jump	Jump	CJ	`C.J imm`	`JAL x0,imm`
	Jump Register	CR	`C.JR rd,rs1`	`JALR x0,rs1,0`
Jump & Link	J&L	CJ	`C.JAL imm`	`JAL ra,imm`
	Jump & Link Register	CR	`C.JALR rs1`	`JALR ra,rs1,0`
System	Env. BREAK	CI	`C.EBREAK`	`EBREAK`

32-bit Instruction Formats

	31	30	25 24	21	20	19	15 14	12 11	8	7	6	0
R	funct7			rs2		rs1	funct3		rd		opcode	
I	imm[11:0]					rs1	funct3		rd		opcode	
S	imm[11:5]			rs2		rs1	funct3		imm[4:0]		opcode	
SB	imm[12]	imm[10:5]		rs2		rs1	funct3		imm[4:1]	imm[11]	opcode	
U	imm[31:12]								rd		opcode	
UJ	imm[20]	imm[10:1]			imm[11]	imm[19:12]			rd		opcode	

16-bit (RVC) Instruction Formats

	15 14 13	12	11 10 9	8 7	6 5	4 3	2	1 0
CR	funct4		rd/rs1		rs2			op
CI	funct3	imm	rd/rs1		imm			op
CSS	funct3	imm			rs2			op
CIW	funct3	imm				rd′		op
CL	funct3	imm	rs1′		imm	rd′		op
CS	funct3	imm	rs1′		imm	rs2′		op
CB	funct3	offset	rs1′		offset			op
CJ	funct3	jump target						op

RISC-V Integer Base (RV32I/64I/128I), privileged, and optional compressed extension (RVC). Registers x1-x31 and the pc are 32 bits wide in RV32I, 64 in RV64I, and 128 in RV128I (x0=0). RV64I/128I add 10 instructions for the wider formats. The RVI base of <50 classic integer RISC instructions is required. Every 16-bit RVC instruction matches an existing 32-bit RVI instruction. See risc.org.

Free & Open RISC-V Reference Card (riscv.org) ②

Optional Multiply-Divide Instruction Extension: RVM

Category	Name	Fmt	RV32M (Multiply-Divide)		+RV{64,128}	
Multiply	MULtiply	R	MUL	rd,rs1,rs2	MUL{W|D}	rd,rs1,rs2
	MULtiply upper Half	R	MULH	rd,rs1,rs2		
	MULtiply Half Sign/Uns	R	MULHSU	rd,rs1,rs2		
	MULtiply upper Half Uns	R	MULHU	rd,rs1,rs2		
Divide	DIVide	R	DIV	rd,rs1,rs2	DIV{W|D}	rd,rs1,rs2
	DIVide Unsigned	R	DIVU	rd,rs1,rs2		
Remainder	REMainder	R	REM	rd,rs1,rs2	REM{W|D}	rd,rs1,rs2
	REMainder Unsigned	R	REMU	rd,rs1,rs2	REMU{W|D}	rd,rs1,rs2

Optional Atomic Instruction Extension: RVA

Category	Name	Fmt	RV32A (Atomic)		+RV{64,128}	
Load	Load Reserved	R	LR.W	rd,rs1	LR.{D|Q}	rd,rs1
Store	Store Conditional	R	SC.W	rd,rs1,rs2	SC.{D|Q}	rd,rs1,rs2
Swap	SWAP	R	AMOSWAP.W	rd,rs1,rs2	AMOSWAP.{D|Q}	rd,rs1,rs2
Add	ADD	R	AMOADD.W	rd,rs1,rs2	AMOADD.{D|Q}	rd,rs1,rs2
Logical	XOR	R	AMOXOR.W	rd,rs1,rs2	AMOXOR.{D|Q}	rd,rs1,rs2
	AND	R	AMOAND.W	rd,rs1,rs2	AMOAND.{D|Q}	rd,rs1,rs2
	OR	R	AMOOR.W	rd,rs1,rs2	AMOOR.{D|Q}	rd,rs1,rs2
Min/Max	MINimum	R	AMOMIN.W	rd,rs1,rs2	AMOMIN.{D|Q}	rd,rs1,rs2
	MAXimum	R	AMOMAX.W	rd,rs1,rs2	AMOMAX.{D|Q}	rd,rs1,rs2
	MINimum Unsigned	R	AMOMINU.W	rd,rs1,rs2	AMOMINU.{D|Q}	rd,rs1,rs2
	MAXimum Unsigned	R	AMOMAXU.W	rd,rs1,rs2	AMOMAXU.{D|Q}	rd,rs1,rs2

Three Optional Floating-Point Instruction Extensions: RVF, RVD, & RVQ

Category	Name	Fmt	RV32{F|D|Q} (HP/SP,DP,QP Fl Pt)		+RV{64,128}	
Move	Move from Integer	R	FMV.{H|S}.X	rd,rs1	FMV.{D|Q}.X	rd,rs1
	Move to Integer	R	FMV.X.{H|S}	rd,rs1	FMV.X.{D|Q}	rd,rs1
Convert	Convert from Int	R	FCVT.{H|S|D|Q}.W	rd,rs1	FCVT.{H|S|D|Q}.{L|T}	rd,rs1
	Convert from Int Unsigned	R	FCVT.{H|S|D|Q}.WU	rd,rs1	FCVT.{H|S|D|Q}.{L|T}U	rd,rs1
	Convert to Int	R	FCVT.W.{H|S|D|Q}	rd,rs1	FCVT.{L|T}.{H|S|D|Q}	rd,rs1
	Convert to Int Unsigned	R	FCVT.WU.{H|S|D|Q}	rd,rs1	FCVT.{L|T}U.{H|S|D|Q}	rd,rs1
Load	Load	I	FL{W,D,Q}	rd,rs1,imm		
Store	Store	S	FS{W,D,Q}	rs1,rs2,imm		
Arithmetic	ADD	R	FADD.{S|D|Q}	rd,rs1,rs2		
	SUBtract	R	FSUB.{S|D|Q}	rd,rs1,rs2		
	MULtiply	R	FMUL.{S|D|Q}	rd,rs1,rs2		
	DIVide	R	FDIV.{S|D|Q}	rd,rs1,rs2		
	SQuare RooT	R	FSQRT.{S|D|Q}	rd,rs1		
Mul-Add	Multiply-ADD	R	FMADD.{S|D|Q}	rd,rs1,rs2,rs3		
	Multiply-SUBtract	R	FMSUB.{S|D|Q}	rd,rs1,rs2,rs3		
	Negative Multiply-SUBtract	R	FNMSUB.{S|D|Q}	rd,rs1,rs2,rs3		
	Negative Multiply-ADD	R	FNMADD.{S|D|Q}	rd,rs1,rs2,rs3		
Sign Inject	SiGN source	R	FSGNJ.{S|D|Q}	rd,rs1,rs2		
	Negative SiGN source	R	FSGNJN.{S|D|Q}	rd,rs1,rs2		
	Xor SiGN source	R	FSGNJX.{S|D|Q}	rd,rs1,rs2		
Min/Max	MINimum	R	FMIN.{S|D|Q}	rd,rs1,rs2		
	MAXimum	R	FMAX.{S|D|Q}	rd,rs1,rs2		
Compare	Compare Float =	R	FEQ.{S|D|Q}	rd,rs1,rs2		
	Compare Float <	R	FLT.{S|D|Q}	rd,rs1,rs2		
	Compare Float ≤	R	FLE.{S|D|Q}	rd,rs1,rs2		
Categorization	Classify Type	R	FCLASS.{S|D|Q}	rd,rs1		
Configuration	Read Status	R	FRCSR	rd		
	Read Rounding Mode	R	FRRM	rd		
	Read Flags	R	FRFLAGS	rd		
	Swap Status Reg	R	FSCSR	rd,rs1		
	Swap Rounding Mode	R	FSRM	rd,rs1		
	Swap Flags	R	FSFLAGS	rd,rs1		
	Swap Rounding Mode Imm	I	FSRMI	rd,imm		
	Swap Flags Imm	I	FSFLAGSI	rd,imm		

RISC-V Calling Convention

Register	ABI Name	Saver	Description
x0	zero	---	Hard-wired zero
x1	ra	Caller	Return address
x2	sp	Callee	Stack pointer
x3	gp	---	Global pointer
x4	tp	---	Thread pointer
x5-7	t0-2	Caller	Temporaries
x8	s0/fp	Callee	Saved register/frame pointer
x9	s1	Callee	Saved register
x10-11	a0-1	Caller	Function arguments/return values
x12-17	a2-7	Caller	Function arguments
x18-27	s2-11	Callee	Saved registers
x28-31	t3-t6	Caller	Temporaries
f0-7	ft0-7	Caller	FP temporaries
f8-9	fs0-1	Callee	FP saved registers
f10-11	fa0-1	Caller	FP arguments/return values
f12-17	fa2-7	Caller	FP arguments
f18-27	fs2-11	Callee	FP saved registers
f28-31	ft8-11	Caller	FP temporaries

RISC-V calling convention and five optional extensions: 10 multiply-divide instructions (RV32M); 11 optional atomic instructions (RV32A); and 25 floating-point instructions each for single-, double-, and quadruple-precision (RV32F, RV32D, RV32Q). The latter add registers f0-f31, whose width matches the widest precision, and a floating-point control and status register fcsr. Each larger address adds some instructions: 4 for RVM, 11 for RVA, and 6 each for RVF/D/Q. Using regex notation, `{}` *means set, so* `L{D|Q}` *is both* `LD` *and* `LQ`*. See risc.org. (8/21/15 revision)*

עד כאן מצגת זו