NATIONAL RESEARCH UNIVERSITY

Computer Architecture and Operating Systems
Lecture 11: Exceptions

Andrei Tatarnikov

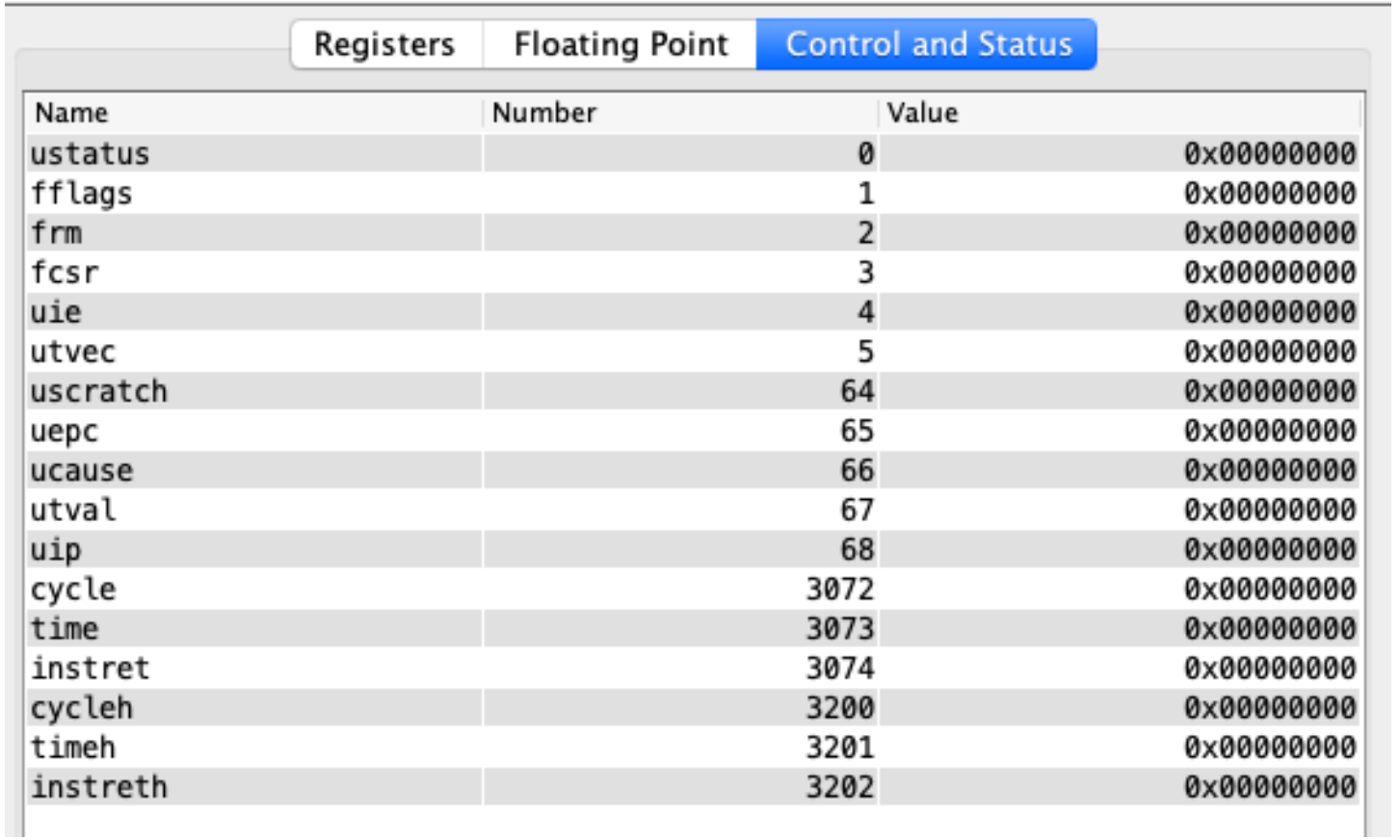
atatarnikov@hse.ru
@andrewt0301

Exceptions and Interrupts

- **Exception** is an unscheduled event that disrupts program execution
  - Arises within the CPU
    - e.g., undefined opcode, system call, …
- **Interrupt** is an exception that comes from outside of the processor
  - From an external I/O controller
- Some architectures use the term interrupt for all exceptions
- Exceptions require special system instructions and registers
- Dealing with exceptions without sacrificing performance is hard

Control and Status Registers

- **Control and Status Registers (CSRs)** are system registers provided by RISC-V to control monitor system states
- CSR's can be read, written and bits can be set/cleared
- Each CSR has a special name and is assigned a unique function.
- In this course, we focus on the user privilege level
- We will use user-level CSRs to handle user-level exceptions



Name	Number	Value
ustatus	0	0x00000000
fflags	1	0x00000000
frm	2	0x00000000
fcsr	3	0x00000000
uie	4	0x00000000
utvec	5	0x00000000
uscratch	64	0x00000000
uepc	65	0x00000000
ucause	66	0x00000000
utval	67	0x00000000
uip	68	0x00000000
cycle	3072	0x00000000
time	3073	0x00000000
instret	3074	0x00000000
cycleh	3200	0x00000000
timeh	3201	0x00000000
instreth	3202	0x00000000

Main CSR Registers

- User Trap Setup
  - **ustatus** – User status register
  - **uie** – User interrupt-enable register
  - **utvec** – User trap handler base address
- User Trap Handling
  - **uscratch** – Scratch register for user trap handlers
  - **uepc** – User exception program counter
  - **ucause** – User trap cause
  - **utval** – User bad address or instruction
  - **uip** – User interrupt pending

System Instructions

- **ebreak** – Pause execution (at a breakpoint)
- **ecall** – Execute a system call specified by value in a7
- **uret** – Return from handling an interrupt (to uepc)
- **wfi** – Wait for interrupt
- **csrrc**, **csrrci**, **csrrs**, **csrrsi**, **csrrw**, **csrrwi** – Read/write CSR

Handling Exceptions in CPU

- Save PC of offending (or interrupted) instruction
  - In RISC-V: User Exception Program Counter (UEPC)
- Save indication of the problem
  - In RISC-V: User Exception Cause Register (UCAUSE)
  - 32 bits, but most bits unused
    - Exception code field: 2 for undefined opcode, 12 for hardware malfunction, …
- Jump to handler
  - Assume at $1C09\ 0000_{hex}$

Handling Vectored Exceptions in CPU

- Alternate Mechanism: Vectored Interrupts
  - Handler address determined by the cause
- Exception vector address to be added to a vector table base register:
  - Undefined opcode 00 0100 0000_{two}
  - Hardware malfunction: 01 1000 0000_{two}
  - …: …
- Handler instructions either
  - Deal with the interrupt, or
  - Jump to real handler

Handler Actions

- Read cause, and transfer to relevant handler
- Determine action required
- If restartable
  - Take corrective action
  - Use UEPC to return to program
- Otherwise
  - Terminate program
  - Report error using UEPC, UCAUSE, …

Trivial Exception Handler

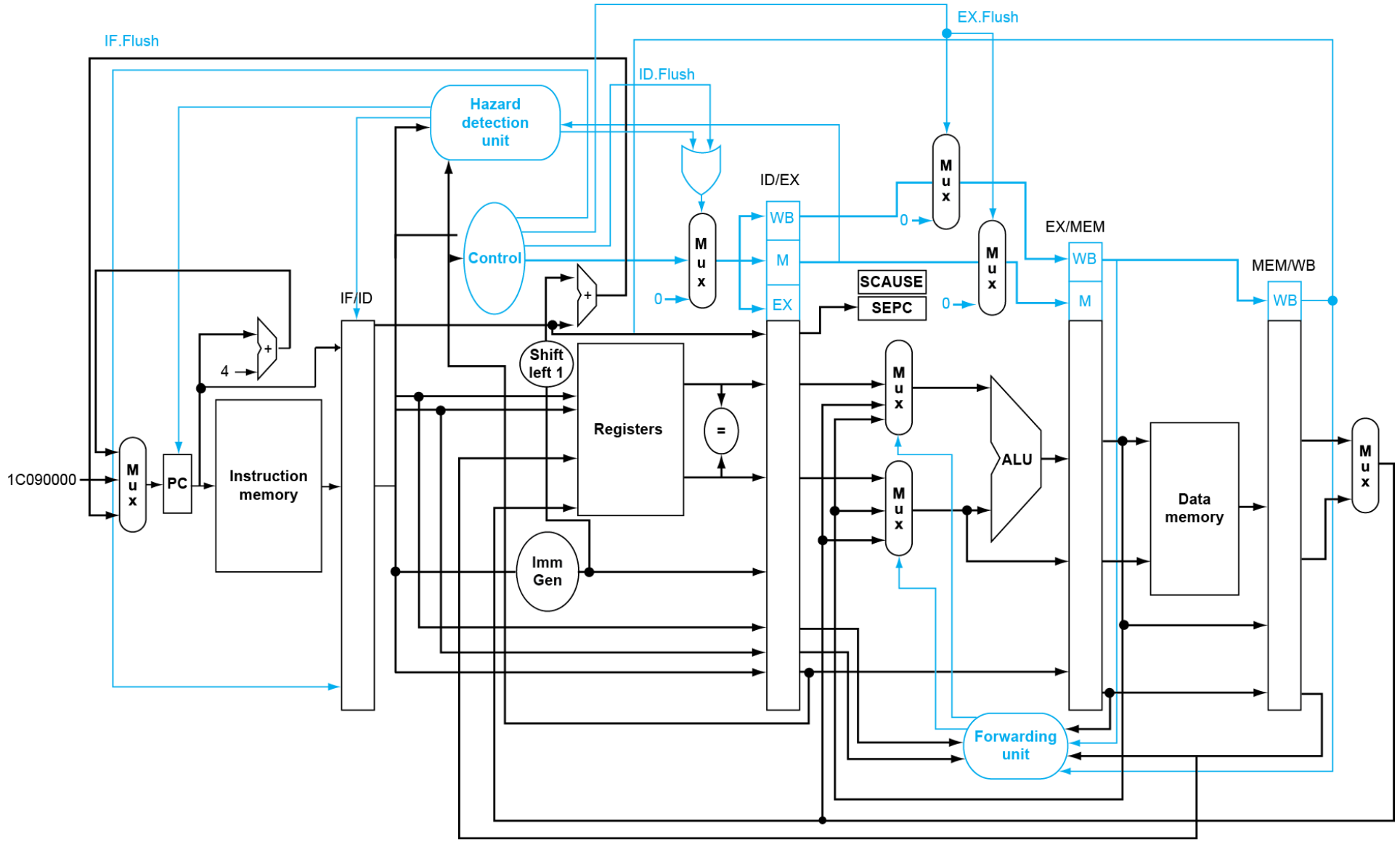
```
# Example with a trivial exception handler that just returns to the next instruction.
    .text
    j main
handler:
    # Just ignore it by moving uepc to the next instruction
    csrrw  t0, uepc, zero  # load exception PC into t0
    addi   t0, t0, 4       # increment t0
    csrrw  zero, uepc, t0  # update exception PC
    uret                   # return to uepc
main:
    la     t0, handler
    csrrw  zero, utvec, t0  # set utvec (5) to the handlers address
    csrrsi zero, ustatus, 1 # set interrupt enable bit in ustatus (0)
    lw     zero, 0(zero)    # trigger trap for Load access fault

    li     a7, 10
    ecall
```

Exceptions in a Pipeline

- Another form of control hazard
- Consider malfunction on add in EX stage

  `add x1, x2, x1`
  - Prevent x1 from being clobbered
  - Complete previous instructions
  - Flush `add` and subsequent instructions
  - Set SEPC and SCAUSE register values
  - Transfer control to handler
- Similar to mispredicted branch
  - Use much of the same hardware

Pipeline with Exceptions



Exception Properties

- Restartable exceptions
  - Pipeline can flush the instruction
  - Handler executes, then returns to the instruction
    - Refetched and executed from scratch
- PC saved in SEPC register
  - Identifies causing instruction

Exception Example

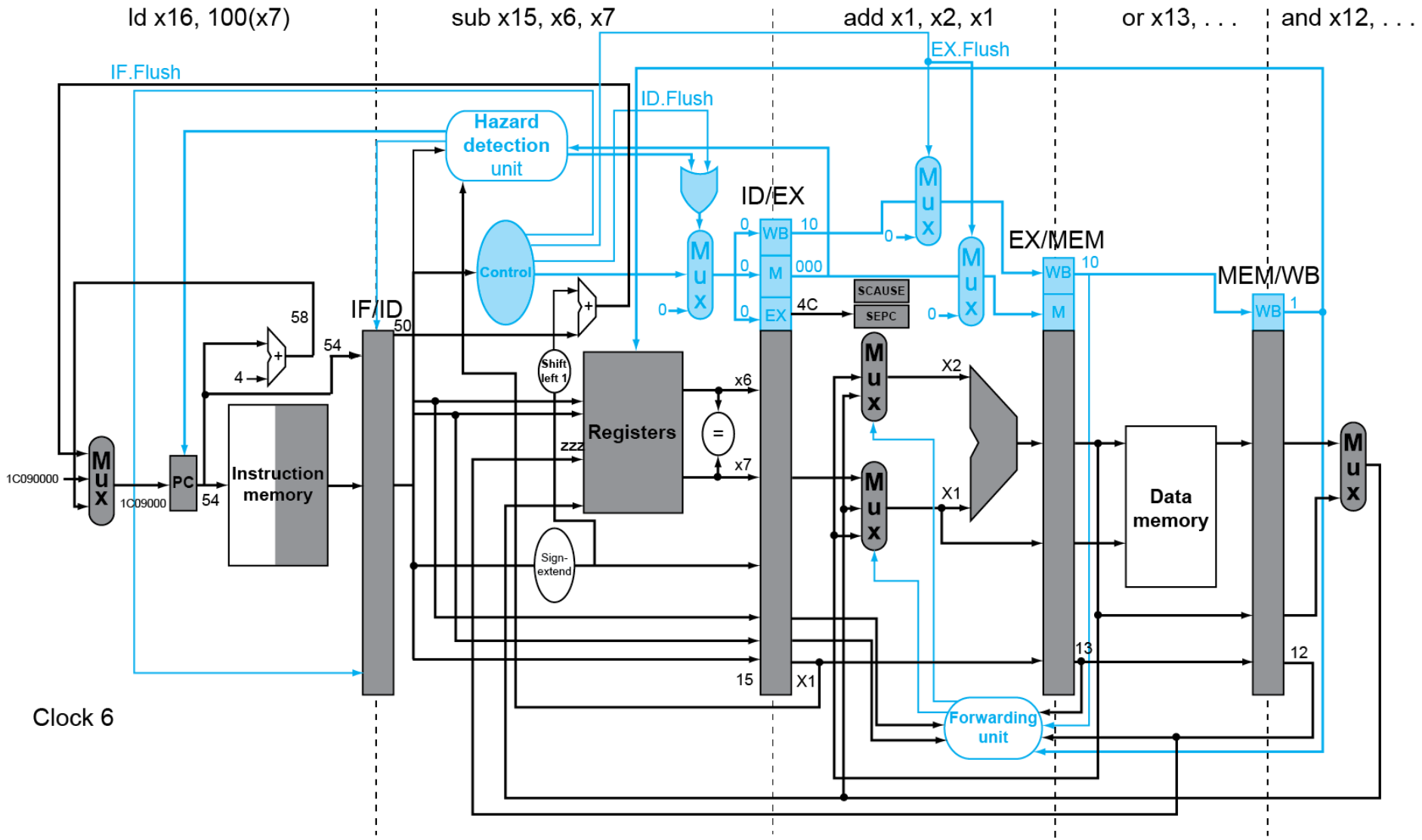
- Exception on `add` in

```
40    sub  x11, x2, x4
44    and  x12, x2, x5
48    orr  x13, x2, x6
4c    add  x1,  x2, x1
50    sub  x15, x6, x7
54    ld   x16, 100(x7)
…
```

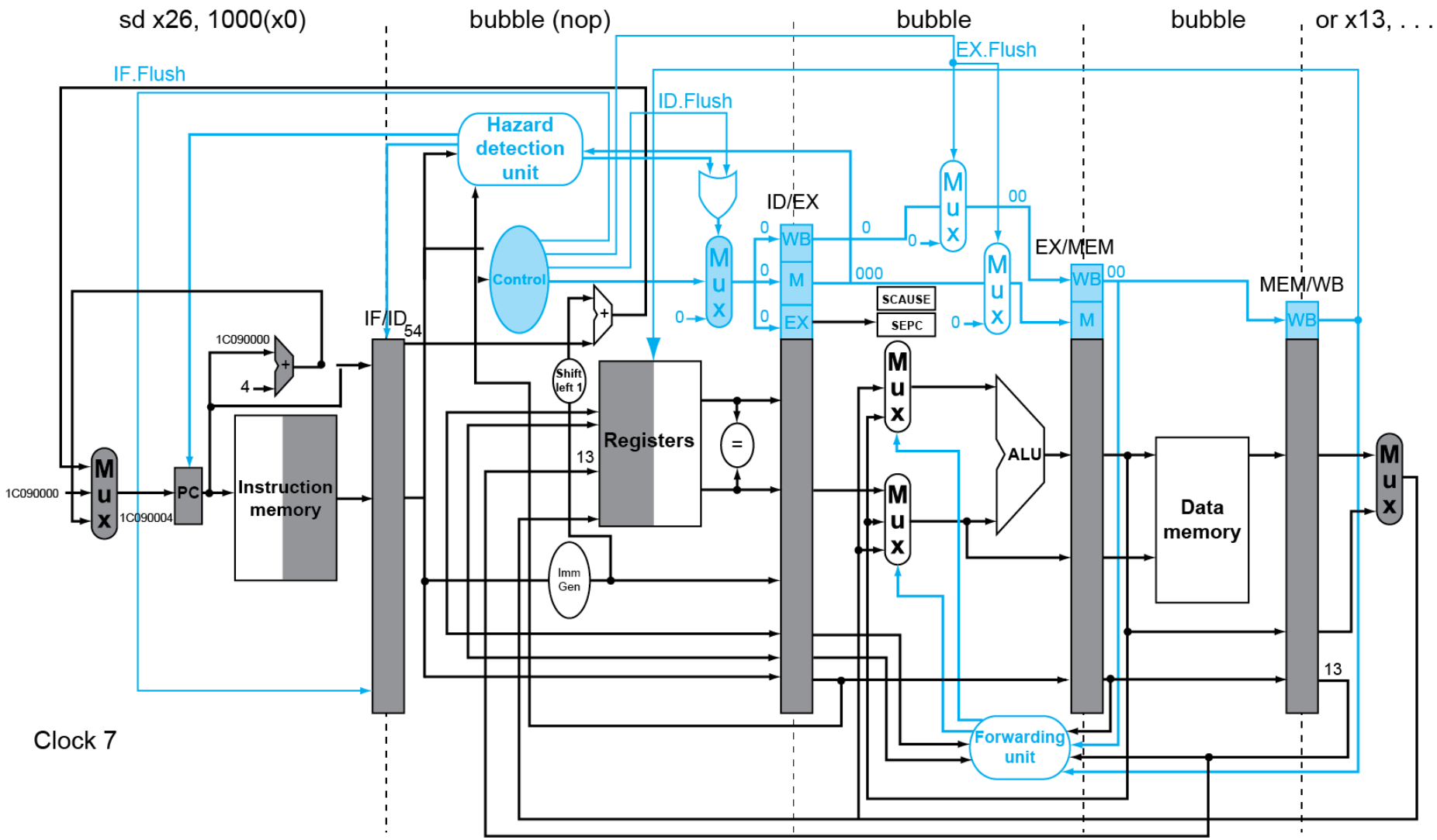
- Handler

```
1c090000 sd x26, 1000(x10)
1c090004 sd x27, 1008(x10)
…
```

Exception Example



Exception Example



Multiple Exceptions

- Pipelining overlaps multiple instructions
  - Could have multiple exceptions at once
- Simple approach: deal with exception from earliest instruction
  - Flush subsequent instructions
  - “Precise” exceptions
- In complex pipelines
  - Multiple instructions issued per cycle
  - Out-of-order completion
  - Maintaining precise exceptions is difficult!

Imprecise Exceptions

- Just stop pipeline and save state
  - Including exception cause(s)
- Let the handler work out
  - Which instruction(s) had exceptions
  - Which to complete or flush
    - May require “manual” completion
- Simplifies hardware, but more complex handler software
- Not feasible for complex multiple-issue out-of-order pipelines

Any Questions?

```
            .text
__start:    addi t1, zero, 0x18
            addi t2, zero, 0x21
cycle:      beq t1, t2, done
            slt t0, t1, t2
            bne t0, zero, if_less
            nop
            sub t1, t1, t2
            j cycle
            nop
if_less:    sub t2, t2, t1
            j cycle
done:       add t3, t1, zero
```