Simulation of a 5-stage Pipelined Nios II Solution

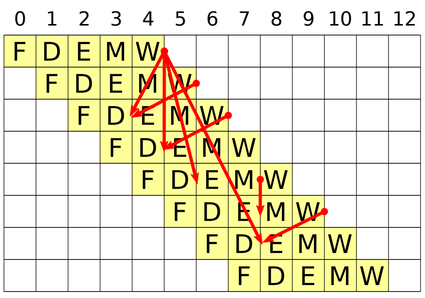
Learning Goal: Processor pipeline.

Requirements: Nios2Sim.

1 Exercise 1

1.1 Simulation

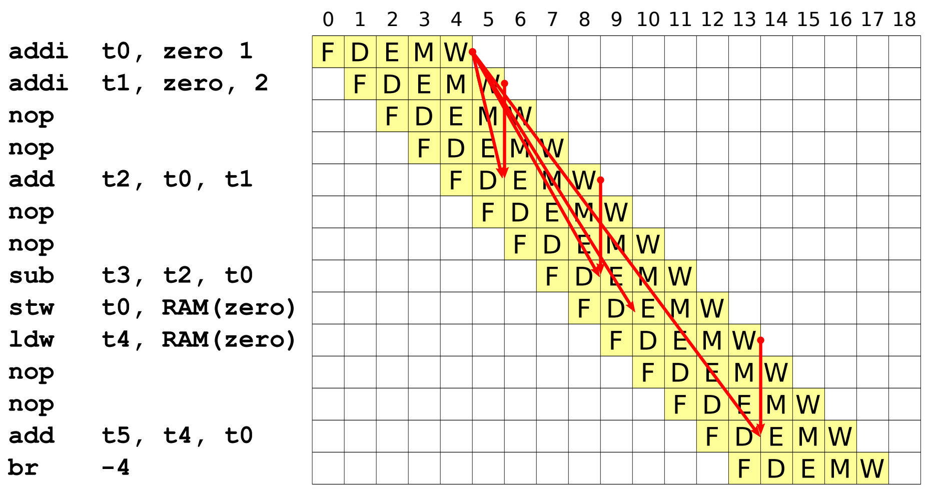
- The expected values of the registers at the end of a normal execution are: `t2` $= 3$, `t3` $= 2$, `t4` $= 1$ and `t5` $= 2$.
- In the simulation the correct values are not obtained because the stalling is disabled. When data hazards occur, the processor will not stop the execution to wait for the correct value. As the E→E and M→E forwarding paths are also disabled, most of these data hazards won't be eliminated.



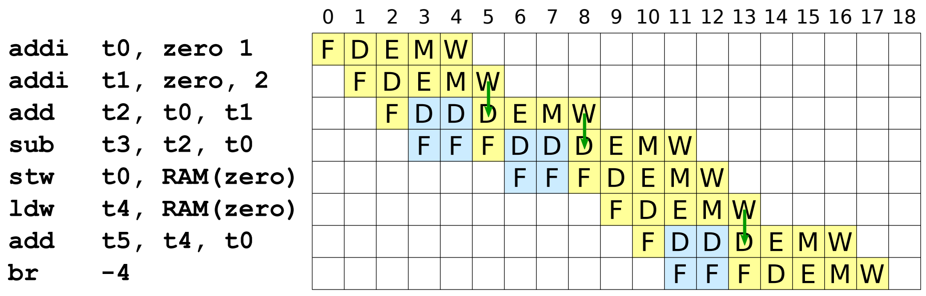
	0	1	2	3	4	5	6	7	8	9	10	11	12
`addi t0, zero 1`	F	D	E	M	W								
`addi t1, zero, 2`		F	D	E	M	W							
`add t2, t0, t1`			F	D	E	M	W						
`sub t3, t2, t0`				F	D	E	M	W					
`stw t0, RAM(zero)`					F	D	E	M	W				
`ldw t4, RAM(zero)`						F	D	E	M	W			
`add t5, t4, t0`							F	D	E	M	W		
`br -4`								F	D	E	M	W	

1.2 Inserting NOPs

```
.equ  RAM, 0x1000
main:
    addi t0, zero, 1    ; t0 = 1
    addi t1, zero, 2    ; t1 = 2
    nop
    nop
    add  t2, t0, t1     ; t2 = t0 + t1
    nop
    nop
    sub  t3, t2, t0     ; t3 = t2 - t0
    stw  t0, RAM(zero)  ; RAM[0] = t0
    ldw  t4, RAM(zero) ; t4 = RAM[0]
    nop
    nop
    add  t5, t4, t0    ; t5 = t4 + t0
end:
    br   end
    nop
    nop
```

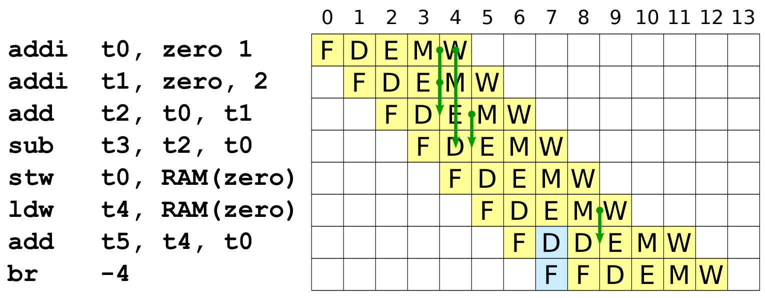


1.3 Stalling



- We can avoid the insertion of bubbles by providing forwarding paths to the pipeline or rearranging the instructions.

1.4 Forwarding



- All of the data hazards are eliminated by the forwarding paths, except for the load-use data hazard that occurs between the `ldw` and `add` instructions: this one requires the pipeline to stall.
- To avoid the stall, we can rearrange the order of the instructions as follow:

```
.equ  RAM, 0x1000
main:
    addi t0, zero, 1   ; t0 = 1
    addi t1, zero, 2   ; t1 = 2
    stw  t0, RAM(zero) ; RAM[0] = t0
    ldw  t4, RAM(zero) ; t4 = RAM[0]
    sub  t3, t2, t0    ; t3 = t2 - t0
    add  t5, t4, t0    ; t5 = t4 + t0
end:
    br   end
```

2 Exercise 2

2.1 Simulation

- The procedure counts the number of bits equal to 1 in an array of 32-bit elements
- `v0` should be equal to 22.
- About 680 cycles are necessary to execute the code.

2.2 Branch

- The `nop` instructions are used because of the 2 delay slots.
- The branch penalty is 2 instructions.
- The 2 instructions following a branch or a jump are executed.
- We can replace the `nop` instructions by other instructions that have to be executed. We have to be careful while we modify the order of execution to not break the functionality of the code.

2.3 Performance

```
proc:
    add  v0, zero, zero       ; v0 = 0
    add  t0, zero, zero       ; t0 = 0
proc_outer:
    bge  t0, a1, proc_return ; if (t0>=a1) goto fin
    ldw  t3, 0(a0)            ; t3 = mem[a0]
    addi t4, zero, 32         ; t4 = 32
proc_inner:
    beq  t4, zero, proc_next ; if (!t4) goto next
    addi t4, t4, -1           ; t4 = t4 - 1
    andi t1, t3, 1            ; t1 = t3 & 1
    br   proc_inner           ; goto inner
    srli t3, t3, 1            ; t3 = t3 >> 1
    add  v0, v0, t1           ; v0 = v0 + t1

```

```
proc_next:
    br   proc_outer           ; goto outer
    addi t0, t0, 1            ; t0 = t0 + 1
    addi a0, a0, 4            ; a0 = a0 + 4
proc_return:
    ret                       ; return to caller
    nop
    nop
```

- This version of the code is executed in about 420 cycles.