Software Pipelining

- Goal is to decrease latency stalls due to RAW hazards.
- Reorganizes loops such that each iteration in the software-pipelined loop is comprised of instructions chosen from different iterations of the original loop.
- Need startup code before the loop and cleanup code after the loop.

Original Loop and Final Software Pipelined Loop

```
Original Loop
loop: fld    f0,0(x1)
      fadd.d f4,f0,f2
      fsd    f4,0(x1)
      addi   x1,x1,-8
      bne    x1,x2,loop

Iteration i:
      fld    f0,0(x1)
      fadd.d f4,f0,f2
      fsd    f4,0(x1)
Iteration i+1:
      fld    f0,0(x1)
      fadd.d f4,f0,f2
      fsd    f4,0(x1)
Iteration i+2:
      fld    f0,0(x1)
      fadd.d f4,f0,f2
      fsd    f4,0(x1)
```

```
Software Pipelined Loop
      fld    f0,0(x1)
      fadd.d f4,f0,f2
      fld    f0,-8(x1)
      addi   x1,x1,-16
loop: fsd    f4,16(x1)
      fadd.d f4,f0,f2
      fld    f0,0(x1)
      addi   x1,x1,-8
      bne    x1,x2,loop
      fsd    f4,0(x1)
      fadd.d f4,f0,f2
      fsd    f4,-8(x1)
```


Original Loop with Stalls

- Assume there is a 2 cycle latency between the **fld** and the **fadd.d**. Also assume there is a 3 cycle latency between the **fadd.d** and the **fsd**.

```
Loop: fld    f0,0(x1)
      stall
      stall
      fadd.d f4,f0,f2
      stall
      stall
      stall
      fsd    f4,0(x1)
      addi   x1,x1,-8
      bne    x1,x2,loop
```


Original Loop with Stalls (cont.)

- Instruction scheduling can only remove one stall.

```
  before scheduling
loop: fld    f0,0(x1)
      stall
      stall
      fadd.d f4,f0,f2
      stall
      stall
      stall
      fsd    f4,0(x1)
      addi   x1,x1,-8
      bne    x1,x2,loop
```

```
    after scheduling
loop: fld    f0,0(x1)
      stall
      stall
      fadd.d f4,f0,f2
      addi   x1,x1,-8
      stall
      stall
      fsd    f4,8(x1)
      bne    x1,x2,loop
```

After Software Pipelining Once

- We will first software pipeline once by moving the **fld** and **fadd.d** before the loop and the **fsd** after the loop to decrease the stalls between the **fadd.d** and the **fsd**.

after SW pipelining

```
      fld    f0,0(x1)
      fadd.d f4,f0,f2
      addi   x1,x1,-8
loop: fsd    f4,8(x1)
      fld    f0,0(x1)
      fadd.d f4,f0,f2
      addi   x1,x1,-8
      bne    x1,x2,loop
      fsd    f4,8(x1)
```

after scheduling

```
      fld    f0,0(x1)
      fadd.d f4,f0,f2
      addi   x1,x1,-8
loop: fld    f0,0(x1)
      fsd    f4,8(x1)
      fadd.d f4,f0,f2
      addi   x1,x1,-8
      bne    x1,x2,loop
      fsd    f4,8(x1)
```


After Software Pipelining Once (cont.)

- We still have a stall in the loop between the **fld** and the **fadd.d**.

without stalls shown

```
      fld    f0,0(x1)
      fadd.d f4,f0,f2
      addi   x1,x1,-8
loop: fld    f0,0(x1)
      fsd    f4,8(x1)
      fadd.d f4,f0,f2
      addi   x1,x1,-8
      bne    x1,x2,loop
      fsd    f4,8(x1)
```

with stalls shown

```
      fld    f0,0(x1)
      stall
      stall
      addi   x1,x1,-8
      fadd.d f4,f0,f2
      stall
loop: fld    f0,0(x1)
      fsd    f4,8(x1)
      stall
      fadd.d f4,f0,f2
      addi   x1,x1,-8
      bne    x1,x2,loop
      stall
      fsd    f4,8(x1)
```


After Software Pipelining Again

- We can software pipeline again, where the **fld** is performed once before the loop and the **fsd** and **fadd.d** are performed once afterwards.

after software pipelining again

```
      fld    f0,0(x1)
      fadd.d f4,f0,f2
      addi   x1,x1,-8
      fld    f0,0(x1)
      addi   x1,x1,-8
loop: fsd    f4,16(x1)
      fadd.d f4,f0,f2
      fld    f0,0(x1)
      addi   x1,x1,-8
      bne    x1,x2,loop
      fsd    f4,16(x1)
      fadd.d f4,f0,f2
      fsd    f4,8(x1)
```

after combining the addi instructions

```
      fld    f0,0(x1)
      fadd.d f4,f0,f2
      fld    f0,-8(x1)
      addi   x1,x1,-16
loop: fsd    f4,16(x1)
      fadd.d f4,f0,f2
      fld    f0,0(x1)
      addi   x1,x1,-8
      bne    x1,x2,loop
      fsd    f4,16(x1)
      fadd.d f4,f0,f2
      fsd    f4,8(x1)
```


Final Software Pipelined Code with Stalls

- Below is the final code with no stalls in the loop.

```
      fld    f0,0(x1)
      stall
      stall
      fadd.d f4,f0,f2
      fld    f0,-8(x1)
      addi   x1,x1,-16
      stall
loop: fsd    f4,16(x1)
      fadd.d f4,f0,f2
      fld    f0,0(x1)
      addi   x1,x1,-8
      bne    x1,x2,loop
      fsd    f4,16(x1)
      fadd.d f4,f0,f2
      stall
      stall
      stall
      fsd    f4,8(x1)
```

Using Conditional or Predicated Instructions

- *Conditional* or *predicated* instructions can be used in some cases to exploit more ILP.
- Predicate registers containing a single bit that are set by evaluating a condition formally associated with a branch are attached to control dependent instructions.
- A conditional or predicated instruction is only committed (store to memory or write back to the register file) if the condition in the predicate register is true.
- The process is called *if conversion* and is used to eliminate branches in simple code sequences.
- It converts control dependences into data dependences.
- Another view is that if conversion merges multiple paths together.

When If Conversion Is Beneficial

- If conversion is often applied on a branch when:
  - the branch has a high misprediction rate
  - there are not many instructions that are control dependent on the branch
  - the different paths of control dependent instructions after the branch are fairly balanced

Using Conditional Instructions Example

```
   C code
if (B < 0)
   A = -B;
else
   A = B;
=>
RISC-V conventional code
    slt   x2,x1,x0      # x2 = x1 < 0 ? 1 : 0;
    beq   x2,x0,L1
    sub   x3,x0,x1      # x3 = -x1;
    j    L2
L1:
    mov   x3,x1
L2:
=>
RISC-V with conditional instructions
    slt   x2,x1,x0      # x2 = x1 < 0 ? 1 : 0;
    subn  x3,x0,x1,x2   # if (x2 != 0) x3 = -x1;
    movz  x3,x1,x2      # if (x2 == 0) x3 = x1;
```


Costs of Using Conditional or Predicated Instructions

- There are still dependences between a condition (predicate) setting instruction and the conditional (predicated) instructions.
- Conditional or predicated instructions that are annulled still take execution time.
- Conditional or predicated instructions are less useful when the control flow is imbalanced as there would be an execution penalty when the shorter execution path would have been taken.
- When branches are highly predictable, execution may take longer due to a longer dependency chain.
- Conditional or predicated instructions require more encoding space.
- Conditional or predicated instructions may lengthen the clock cycle time.