Compiler Construction

Loop unrolling

Can we do better?

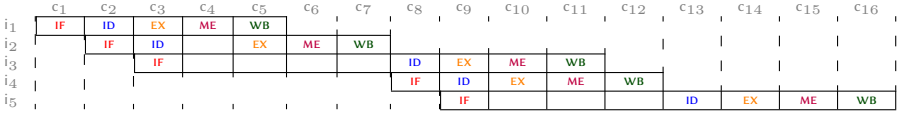
Consider the following code (representing a basic block):

```
i1: Loop: lw    $t0, 0($s1)     # t0=array element
i2:       addu  $t0, $t0, $s2   # add scalar in s2
i3:       sw    $t0, 0($s1)     # store result
i4:       addi  $s1, $s1,-4     # decrement pointer
i5:       bne   $s1, $0, Loop   # branch s1!=0
```

Can we do better?

Consider the following code
(representing a basic block):

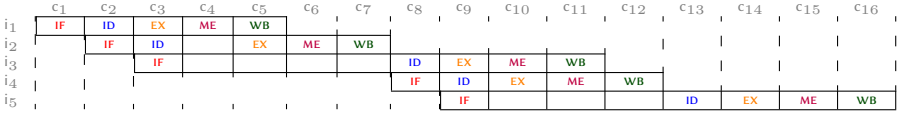
```
i1: Loop: lw   $t0, 0($s1)    # t0=array element
i2:       addu $t0, $t0, $s2  # add scalar in s2
i3:       sw   $t0, 0($s1)    # store result
i4:       addi $s1, $s1,-4    # decrement pointer
i5:       bne  $s1, $0, Loop  # branch s1!=0
```



Can we do better?

Consider the following code (representing a basic block):

```
i1: Loop: lw   $t0, 0($s1)    # t0=array element
i2:       addu $t0, $t0, $s2  # add scalar in s2
i3:       sw   $t0, 0($s1)    # store result
i4:       addi $s1, $s1,-4    # decrement pointer
i5:       bne  $s1, $0, Loop  # branch s1!=0
```



16 cycles for 5 instructions that are all dependant (IPC $= 0.31$)!

Loop Unrolling

- Replicate loop body to expose more parallelism
- Reduces loop-control overhead

At high level, it can be seen as following:

Without Loop Unrolling	With Loop Unrolling

int i;
for (i = 0; x < 100; ++i)
  tab[i] = tab[i] +42;
	int i;
	for (i = 0; x < 100; i+=5)
	  tab[i] = tab[i] +42;
	  tab[i+1] = tab[i+1] +42;
	  tab[i+2] = tab[i+2] +42;
	  tab[i+3] = tab[i+3] +42;
	  tab[i+4] = tab[i+4] +42;

Loop Unrolling – back to the example

```
i1:   Loop:  lw    $t0, 0($s1)
i2:          addu  $t0, $t0, $s2
i3:          sw    $t0, 0($s1)
i4:          addi  $s1, $s1,-4
i5:          bne   $s1, $0, Loop
i6:   Loop:  lw    $t0, 0($s1)
i7:          addu  $t0, $t0, $s2
i8:          sw    $t0, 0($s1)
i9:          addi  $s1, $s1,-4
i10:         bne   $s1, $0, Loop
i11:  Loop:  lw    $t0, 0($s1)
i12:         addu  $t0, $t0, $s2
i13:         sw    $t0, 0($s1)
i14:         addi  $s1, $s1,-4
i15:         bne   $s1, $0, Loop
```

First duplicate N times the the body of the loop!

Loop Unrolling – back to the example

```
i1:  Loop:  lw    $t0, 0($s1)
i2:         addu  $t0, $t0, $s2
i3:         sw    $t0, 0($s1)
i4:         addi  $s1, $s1,-4
i6:         lw    $t0, 0($s1)
i7:         addu  $t0, $t0, $s2
i8:         sw    $t0, 0($s1)
i9:         addi  $s1, $s1,-4
i11:        lw    $t0, 0($s1)
i12:        addu  $t0, $t0, $s2
i13:        sw    $t0, 0($s1)
i14:        addi  $s1, $s1,-4
i15:        bne   $s1, $0, Loop
```

Remove redundant labels and jump
(by supposing that we are able to do it!)

Loop Unrolling – back to the example

```
i1:   Loop:  lw    $t0, 0($s1)
i2:          addu  $t0, $t0, $s2
i3:          sw    $t0, 0($s1)
i4:          addi  $s1, $s1,-4
i6:          lw    $t1, 0($s1)
i7:          addu  $t1, $t1, $s2
i8:          sw    $t1, 0($s1)
i9:          addi  $s1, $s1,-4
i11:         lw    $t2, 0($s1)
i12:         addu  $t2, $t2, $s2
i13:         sw    $t2, 0($s1)
i14:         addi  $s1, $s1,-4
i15:         bne   $s1, $0, Loop
```

Use other temporaries name when possible!

Loop Unrolling – back to the example

```
i4:   Loop:  addi   $s1, $s1,-12
i1:          lw     $t0, 0($s1)
i2:          addu   $t0, $t0, $s2
i3:          sw     $t0, 0($s1)
i6:          lw     $t1, 4($s1)
i7:          addu   $t1, $t1, $s2
i8:          sw     $t1, 4($s1)
i11:         lw     $t2, 8($s1)
i12:         addu   $t2, $t2, $s2
i13:         sw     $t2, 8($s1)
i15:         bne    $s1, $0, Loop
```

Grab redundant operation and merge them **carefully**!

Loop Unrolling – back to the example

```
i1:  Loop:  addi   $s1, $s1,-12
i2:         lw     $t0, 0($s1)
i3:         lw     $t1, 4($s1)
i4:         lw     $t2, 8($s1)
i5:         addu   $t0, $t0, $s2
i6:         addu   $t1, $t1, $s2
i7:         addu   $t2, $t2, $s2
i8:         sw     $t0, 0($s1)
i9:         sw     $t1, 4($s1)
i10:        sw     $t2, 8($s1)
i11:        bne    $s1, $0, Loop
```

Schedule the instructions and renumber them

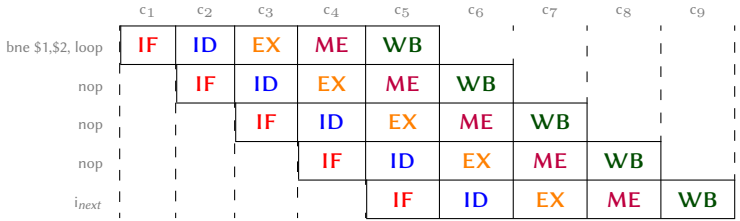
Pros & Cons

- We avoid a lot of conditional jumps (and many stall hence)
- We require 19 cycles for 11 instructions: IPC=0.57 (a lot better than the previous 0.31)
- This trick allows to have more independent instructions to insert, and thus, less stalls!
- But we have now a prologue and an epilogue: i.e., two more basic blocks
- Require more temporaries: register allocation will be harder!
- Try it by yourself in gcc `-funroll-loops`

A very last word on Branch Hazards 1/2

- Conditional jumps often introduce delays since we cannot pre-fetch instrcutions
- Can we avoid them?

We only know i_{next} at cycle 5!



A very last word on Branch Hazards 2/2

- X delayed slot: the X instructions after a branch are systematically executed
- The original SPARC and MIPS processors each used a single branch delay slot to eliminate single-cycle stalls after branches
- We need branch prediction... but nowadays, most of processors do it for us (and use `slt`...)!
- Some architectures have bypass between stages to avoid stalls

Avoid as possible floating points and jumps!

Summary

”Do you program in `mips`?” she asked.
”`nop`”, he said.