Topic V09

Conditional Instructions
and Loops
Reading: (Section 2.7)

Conditional Operations (recap)

Branch to a labeled instruction if a condition is true

Otherwise, continue sequentially

beq rs1, rs2, L1

if (rs1 == rs2) branch to instruction labeled L1

bne rs1, rs2, L2

if (rs1 != rs2) branch to instruction labeled L2

jal zero, L3

unconditional jump to instruction labeled L3

While Loops

How can we improve this code?

C code:

```
while (save[i] == k)
      i = i + j;
…
```

Assumption

i ↔ s3
j ↔ s4
k ↔ s5
base of save[] ↔ s6
save is an array of 4-byte integers

RISC-V code:

```
Loop: slli  t1, s3, 2          # t1 ← i * 4
      add   t1, t1, s6         # t1 ← Addr(save[i])
      lw    t0, 0(t1)          # t0 ← save[i]
      bne   t0, s5, DoneLoop   # if save[i] ≠ k goto DoneLoop
      add   s3, s3, s4         # i ← i + j
      jal   zero, Loop         # goto Loop
DoneLoop:
```

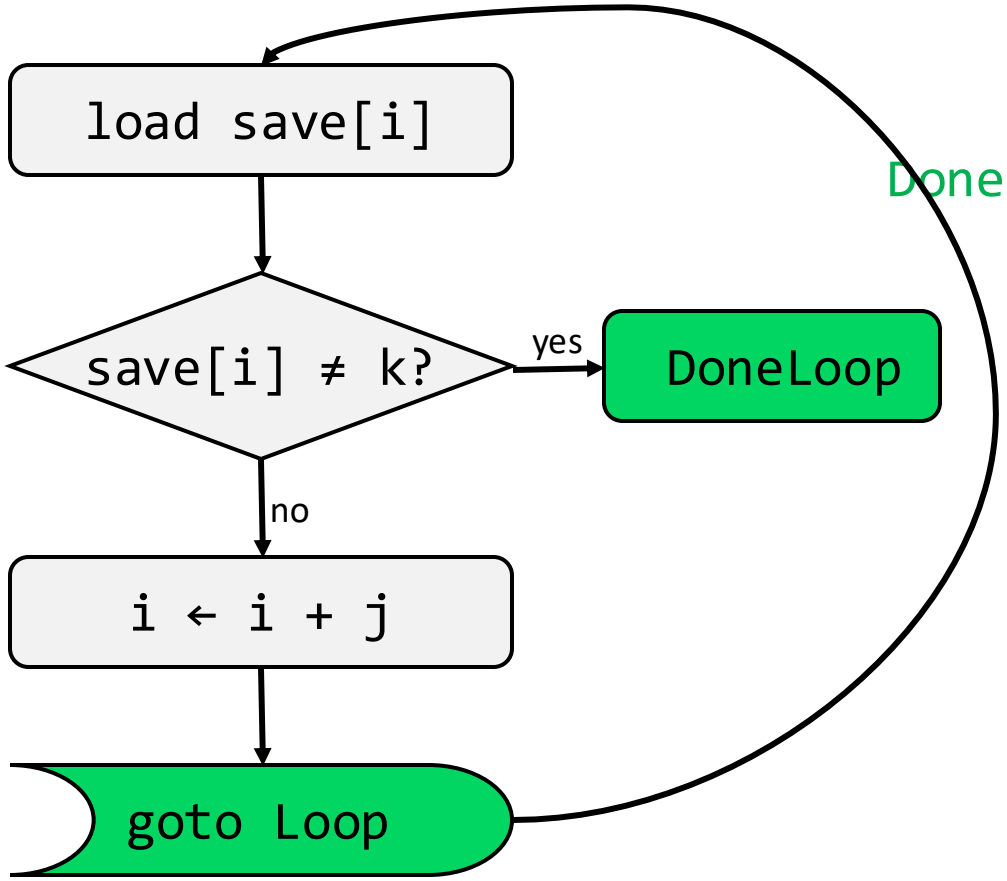
C code:

```
while (save[i] == k)
        i = i + j;
…
```

RISC-V code:

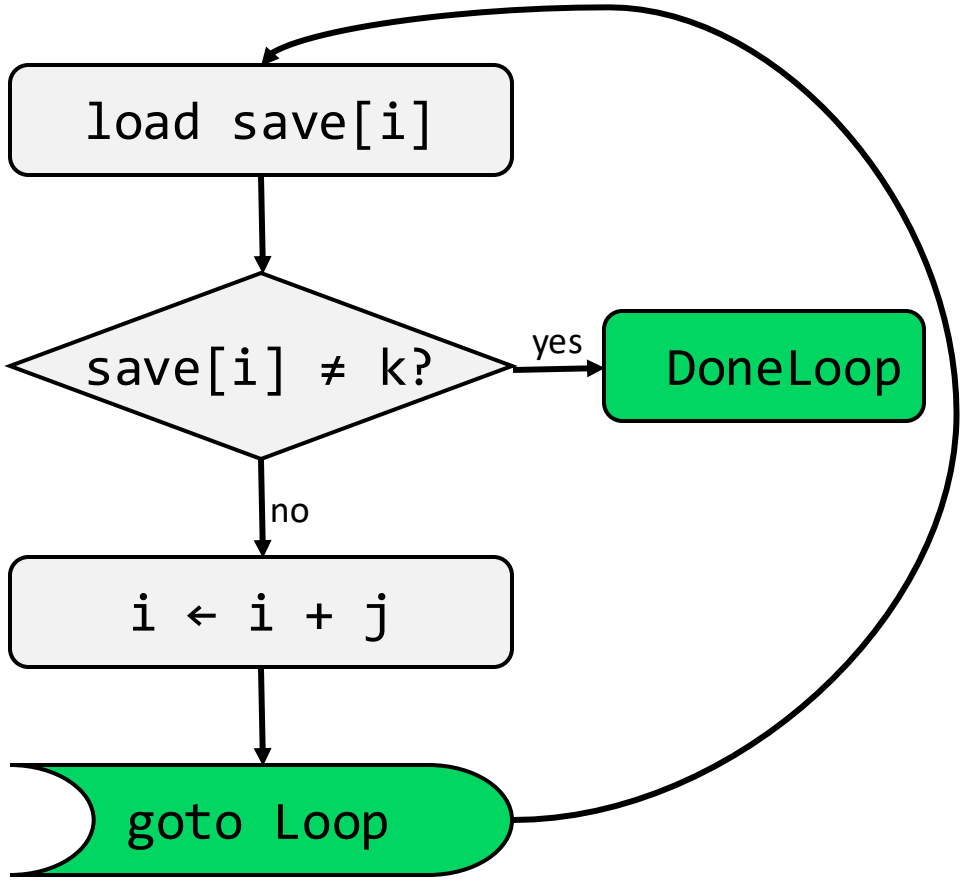
```
Loop:  slli   t1, s3, 2     # t1 ← i * 4
       add    t1, t1, s6    # t1 ← Addr(save[i])
       lw     t0, 0(t1)     # t0 ← save[i]
       bne    t0, s5, DoneLoop    # if save[i] ≠ k got
       add    s3, s3, s4    # i ← i + j
       jal    zero, Loop    # goto Loop

DoneLoop:
```

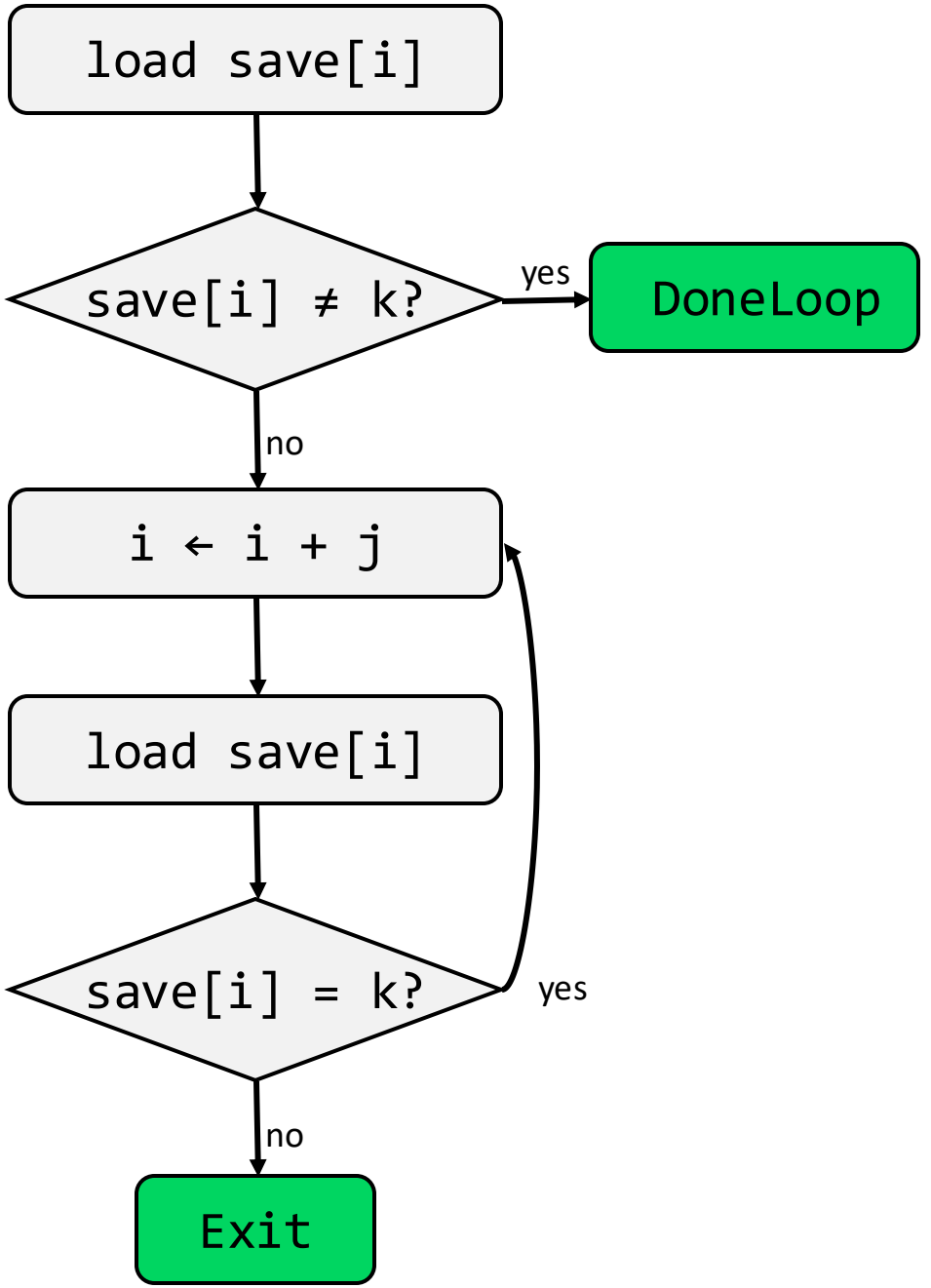


C code:

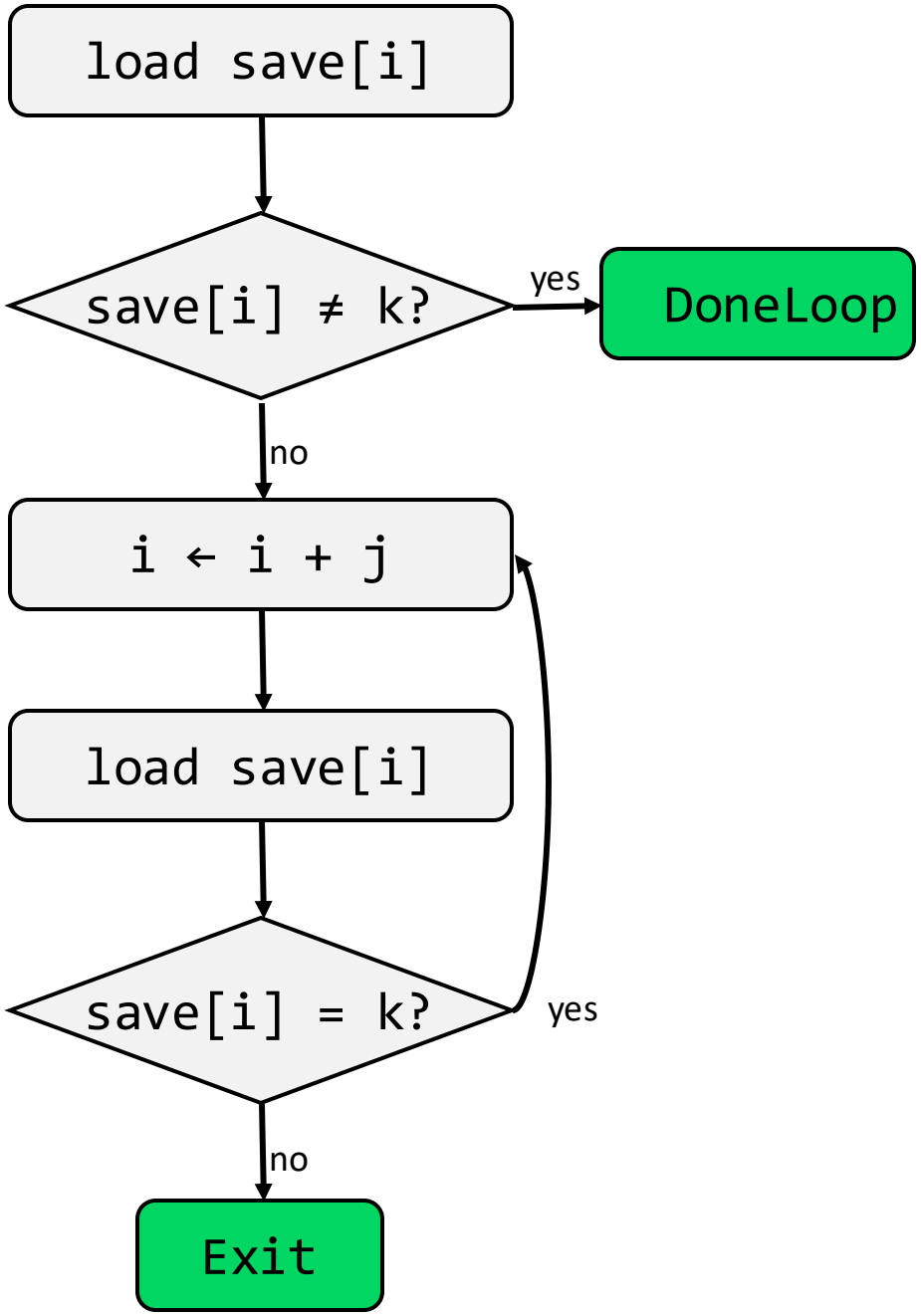
```
while (save[i] == k)
        i = i + j;
…
```



Before Transformation



After Transformation



After Transformation

Before Transformation:

```
Loop:   slli   t1, s3, 2          # t1 ← i * 4
        add    t1, t1, s6         # t1 ← Addr(save[i])
        lw     t0, 0(t1)          # t0 ← save[i]
        bne    t0, s5, DoneLoop   # if save[i] ≠ k goto
        add    s3, s3, s4         # i ← i + j
        jal    zero, Loop         # goto Loop
DoneLoop:
```

After Transformation:

```
        slli   t1, s3, 2          # t1 ← i * 4
        add    t1, t1, s6         # t1 ← Addr(save[i])
        lw     t0, 0(t1)          # t0 ← save[i]
        bne    t0, s5, DoneLoop   # if save[i] ≠ k goto
Loop:   add    s3, s3, s4         # i ← i + j
        slli   t1, s3, 2          # t1 ← i * 4
        add    t1, t1, s6         # t1 ← Addr(save[i])
        lw     t0, 0(t1)          # t0 ← save[i]
        beq    t0, s5, Loop       # if save[i] ≠ k goto
DoneLoop:
```

Non-Branch Comparison

Set result to 1 if a condition is true

Otherwise, set to 0

Set less than

```
slt  rd, rs1, rs2
```

Set less than immediate

```
slti rd, rs, immediate
```

```
if (rs1 < rs2)
      rd ← 1;
else
      rd ← 0;
```

```
if (rs1 < imm)
      rd ← 1;
else
      rd ← 0;
```

Example: slt avoids branches

a0

```
int signum(int x)
{
  if(x > 0) return 1;
  if(x < 0) return -1;
  return 0;
}
```

RISC-V Assembly:

```
signum:
        slt  t1, a0, zero  # t1 ← 1 if x < 0
        slt  a0, zero, a0  # a0 ← 1 is x > 0
        sub  a0, a0, t1    # 1 is x>0; 0 if x==0, -1 if x<0
        ret
```

return value

Other Branches

Branch on less than

```
blt  rs1, rs2, L1
```

```
if (rs1 < rs2)
      PC ← PC + {imm,1b'0}
else
      PC ← PC + 4;
```

Branch on greater than or equal

```
bge  rs1, rs2, L1
```

```
if (rs1 ≥ rs2)
      PC ← PC + {imm,1b'0}
else
      PC ← PC + 4;
```

What about the other types of comparisons; why are there only instructions for `blt` and `bge`?

Other Branches

Suppose you have two registers, s1 and s2, that you wish to compare:

Branch on greater than or equal

```
bge  s2, s1, L1
```

How do you branch on s1 less than or equal s2?

Branch on less than

```
blt  s2, s1, L1
```

How do you branch on s1 greater than s2?

Other Branches (Pseudo Instructions)

Branch on equal zero

```
beqz s1, L1
```

Equivalent to:

```
beq  s1, zero, L1
```

```
if (rs1 = 0)
      PC ← PC + {imm,1b'0}
else
      PC ← PC + 4;
```

Branch on not equal zero

```
bnez s1, L1
```

Equivalent to:

```
bne  s1, zero, L1
```

```
if (rs1 ≠ 0)
      PC ← PC + {imm,1b'0}
else
      PC ← PC + 4;
```

Signed vs. Unsigned

Signed comparison: `slt`, `slti`

Unsigned comparison: `sltu`, `sltiu`

```
s0 = 1111 1111 1111 1111 1111 1111 1111 1111
s1 = 0000 0000 0000 0000 0000 0000 0000 0001
```

```
slt   t0, s0, s1 #signed
```

`t0←1` because `-1 < +1`

```
sltu  t0, s0, s1 #unsigned
```

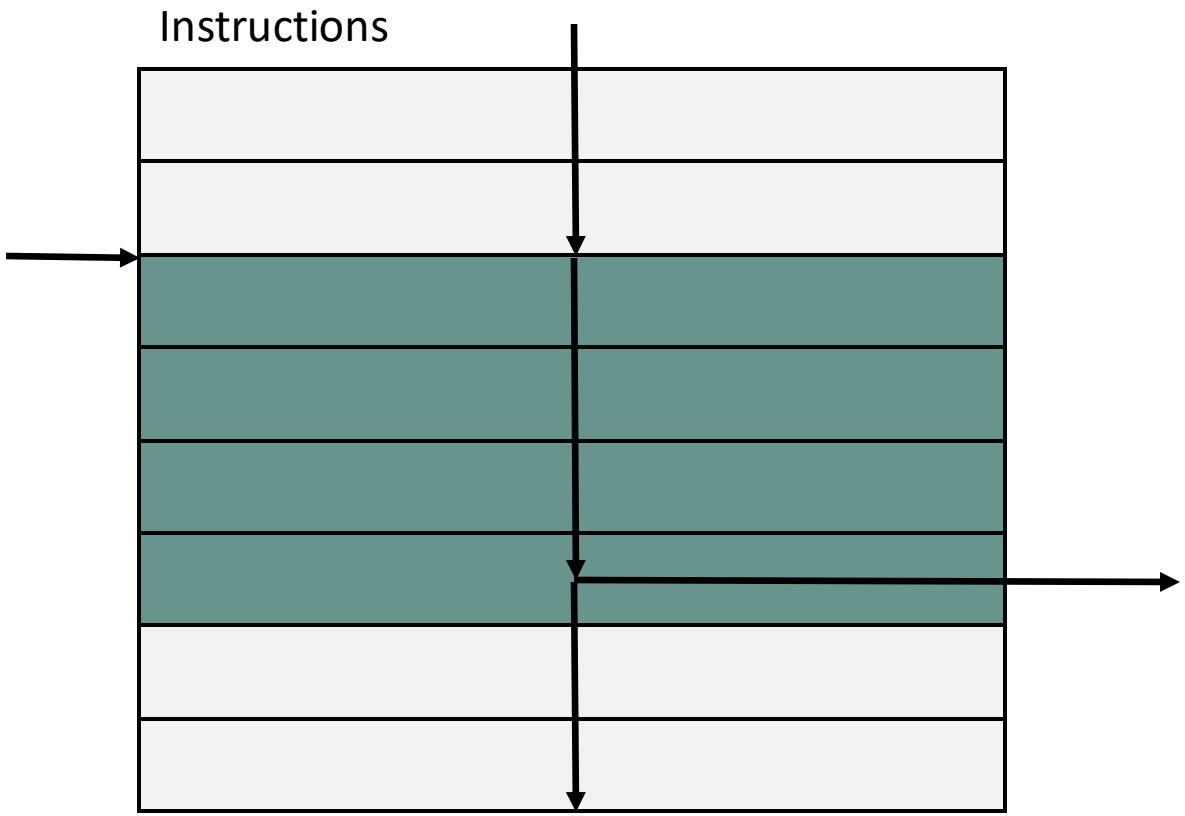
`t0←0` because `+4,294,967,295 > +1`

Basic Blocks

A basic block is a sequence of instructions with:

No branch targets (except at beginning)

No embedded branches (except at end)



A compiler identifies basic blocks for optimization

An advanced processor can accelerate execution of basic blocks

slt

blt

sltu

bge

beq

bnez

beqz

```
Loop:   slli    t1, s3, 2
        add     t1, t1, s6
        lw      t0, 0(t1)
        bne     t0, s5, Exit
        add     s3, s3, s4
        jal     zero, Loop
Exit:
```

```
        slli    t1, s3, 2
        add     t1, t1, s6
        lw      t0, 0(t1)
        bne     t0, s5, Exit
Loop:   add     s3, s3, s4
        slli    t1, s3, 2
        add     t1, t1, s6
        lw      t0, 0(t1)
        beq     t0, s5, Loop
Exit:
```

Recap