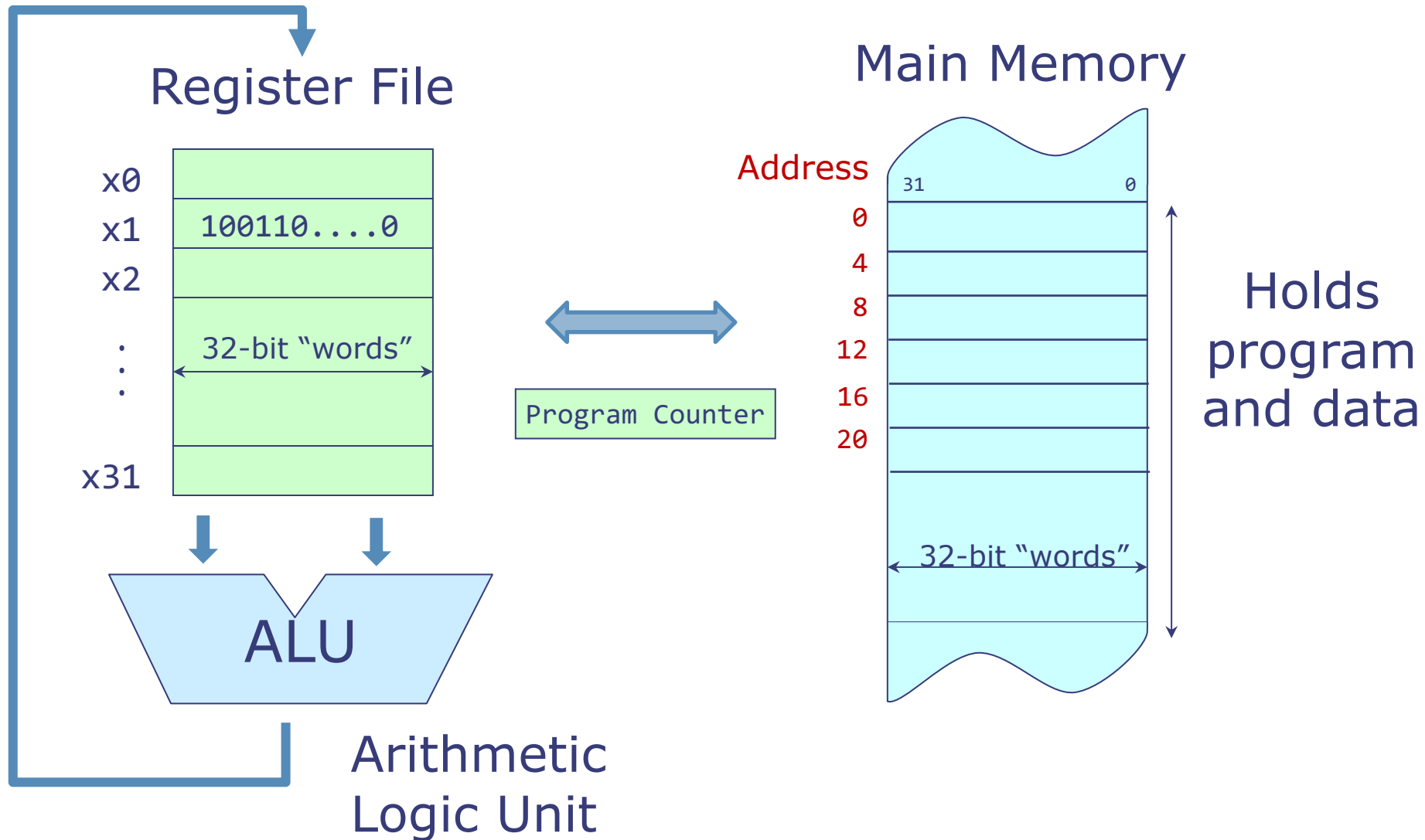
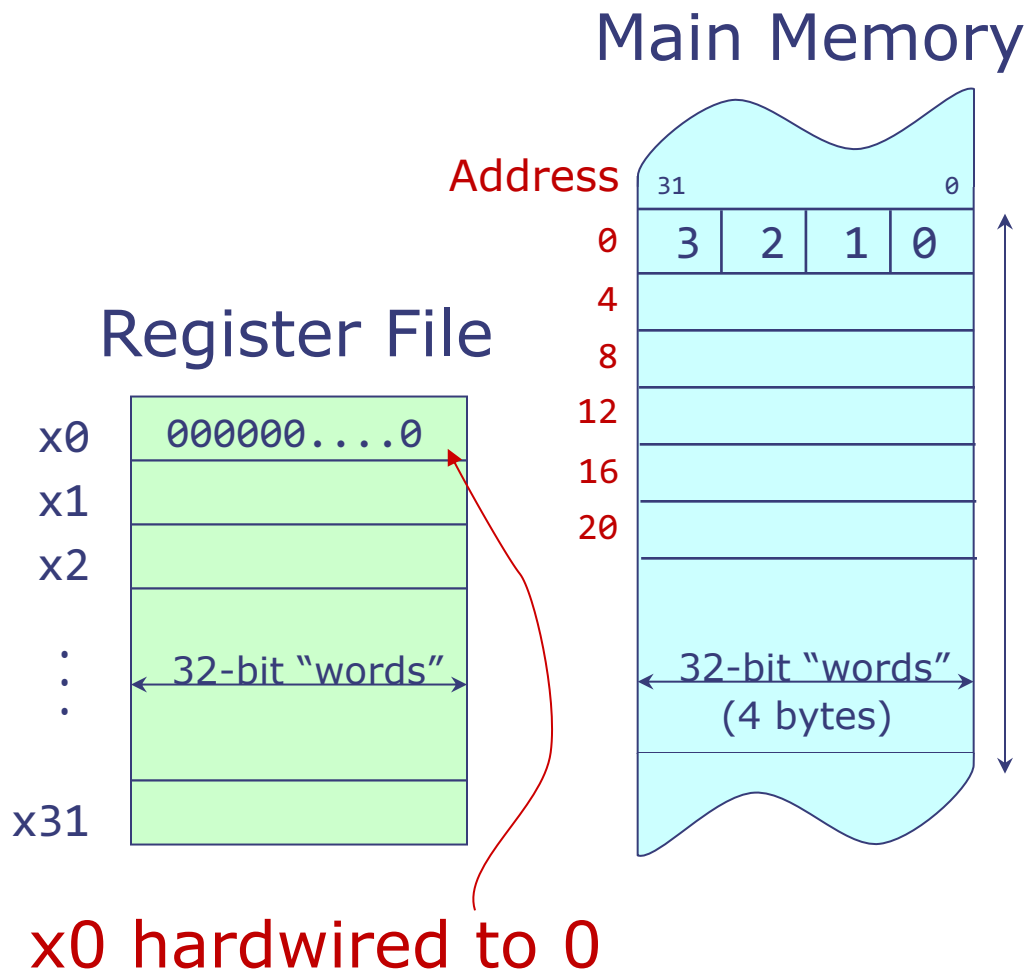


Compilers and RISC-V Assembly

Components of a MicroProcessor



RISC-V Processor Storage



Registers:

- 32 General Purpose Registers
- Each register is 32 bits wide
- $x0 = 0$

Memory:

- Each memory location is 32 bits wide (1 word)
 - Instructions and data
- Memory is byte (8 bits) addressable
- Address of adjacent words are 4 apart.
- Address is 32 bits
- Can address 2^{32} bytes or 2^{30} words.

RISC-V Instruction Types

- Computational Instructions executed by ALU
 - Register-Register: `op dest, src1, src2`
 - Register-Immediate: `op dest, src1, const`
- Control flow instructions
 - Conditional: `br_comp src1, src2, label`
 - Unconditional: `jal dest, label` and `jalr dest, offset(register)`
- Loads and Stores
 - `lw dest, offset(register)`
 - `sw src, offset(register)`
- Pseudoinstructions
 - Shorthand for other instructions

Compiling Simple Expressions

- Assign variables to registers
- Translate operators into computational instructions
- Use register-immediate instructions to handle operations with small constants
- Use the `li` pseudoinstruction for large constants

Example C code

```
int x, y, z;
```

```
...
```

```
y = (x + 3) | (y + 123456);
```

```
z = (x * 4) ^ y;
```

RISC-V Assembly

```
// x: x10, y: x11, z: x12
```

```
// x13, x14 used for temporaries
```

```
addi x13, x10, 3
```

```
li x14, 123456
```

```
add x14, x11, x14
```

```
or x11, x13, x14
```

```
slli x13, x10, 2
```

```
xor x12, x13, x11
```

Compiling Conditionals

- *if-else* statements are similar:

C code

```
if (expr) {  
    if-body  
} else {  
    else-body  
}
```

RISC-V Assembly

```
(compile expr into xN)  
beqz xN, else  
(compile if-body)  
j endif  
else:  
    (compile else-body)  
endif:
```



Compiling Loops

- Loops can be compiled using *backward* branches:

C code

```
while (expr) {  
    while-body  
}
```

RISC-V Assembly

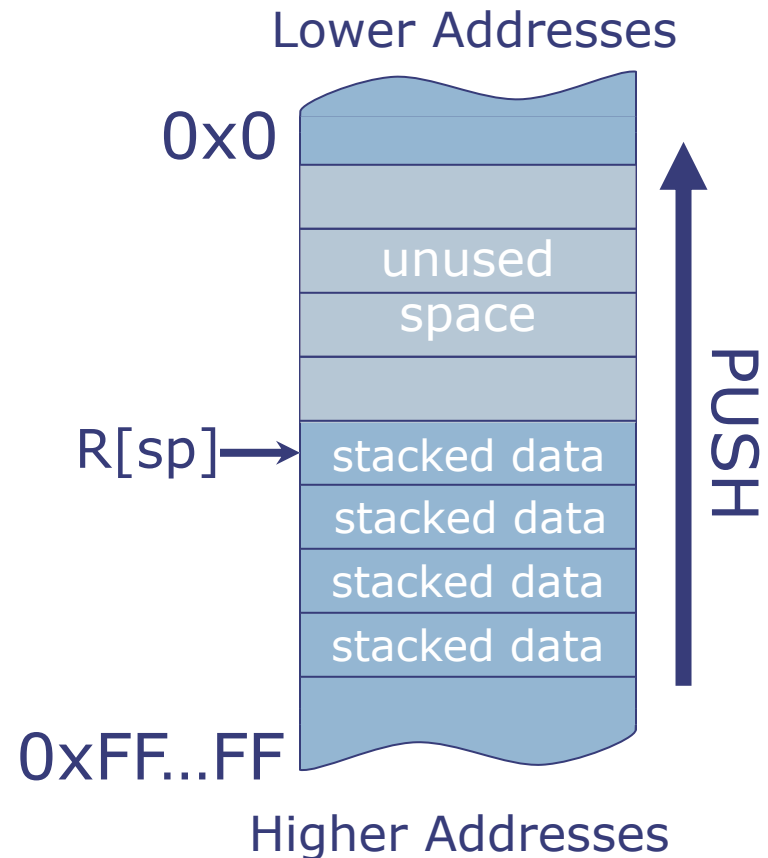
```
while:   
    (compile expr into xN)  
    beqz xN, endwhile  
    (compile while-body)  
    j while   
endwhile:
```

RISC-V Stack

- Stack is in memory
- Stack grows down from higher to lower addresses
- `sp` points to top of stack (last pushed element)
- Push sequence:

```
addi sp, sp, -4 //allocate
sw a1, 0(sp)
```
- Pop sequence:

```
lw a1, 0(sp)
addi sp, sp, 4 //deallocate
```
- Discipline: Can use stack *at any time*, but leave it as you found it!



Compiling Procedures

- A **calling convention** specifies rules for register usage across procedures
- RISC-V calling convention gives symbolic names to registers x0-x31 to denote their role:

Symbolic name	Registers	Description	Saver
a0 to a7	x10 to x17	Function arguments	Caller
a0 and a1	x10 and x11	Function return values	Caller
ra	x1	Return address	Caller
t0 to t6	x5-7, x28-31	Temporaries	Caller
s0 to s11	x8-9, x18-27	Saved registers	Callee
sp	x2	Stack pointer	Callee
gp	x3	Global pointer	---
tp	x4	Thread pointer	---
zero	x0	Hardwired zero	---

Using the Stack to Satisfy Calling Conventions

Caller: Saves any **caller-saved** register (aN, tN, or ra registers), whose values need to be maintained past procedure call, on the stack prior to proc call and restores it upon return.

```
addi sp, sp, -8
sw ra, 0(sp)
sw a1, 4(sp)
call func
lw ra, 0(sp)
lw a1 4(sp)
addi sp, sp, 8
```

func:

Callee: Saves **callee-saved** registers (sN) on stack before using them in a procedure. Must restore sN registers and stack before exiting procedure.

```
func:
    addi sp, sp, -4
    sw s0, 0(sp)
    ...
    lw s0, 0(sp)
    addi sp, sp, 4
    ret
```

Compile Factorial to RISC-V

```
int n = 20;
int r = 0;

r = 1;

while (n > 0) {
    r = r*n;
    n = n-1;
}
```

Diagram showing the mapping of C code to RISC-V assembly instructions using curly braces for grouping:

- `int n = 20;` and `int r = 0;` are grouped together and mapped to:
 - `. = 0x100`
 - `n: .word 0x00000014`
 - `r: .word 0x00000000`
- `r = 1;` is mapped to:
 - `start:`
 - `addi t0, zero, 1`
 - `sw t0, 0x104(zero)`
- The `while` loop body is mapped to:
 - `loop:`
 - `lw t1, 0x100(zero)`
 - `slt t2, zero, t1`
 - `beqz t2, done`
 - `lw t3, 0x104(zero)`
 - `lw t1, 0x100(zero)`
 - `mul t3, t1, t3`
 - `sw t3, 0x104(zero)`
 - `lw t1, 0x100(zero)`
 - `addi t1, t1, -1`
 - `sw t1, 0x100(zero)`
 - `j loop`
- The `done:` label is shown at the end of the assembly block.

Easy translation

Slow code
(11 instructions
in the loop)

done:

Anatomy of a Modern Compiler

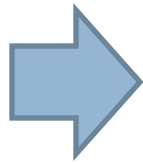


- Read source program
 - Break it up into basic elements
 - Check correctness, produce errors
 - Translate to generic **intermediate representation (IR)**
- Optimize IR
 - Translate IR to ASM
 - Optimize ASM

Frontend Stages

- Lexical analysis (scanning): Source → List of tokens

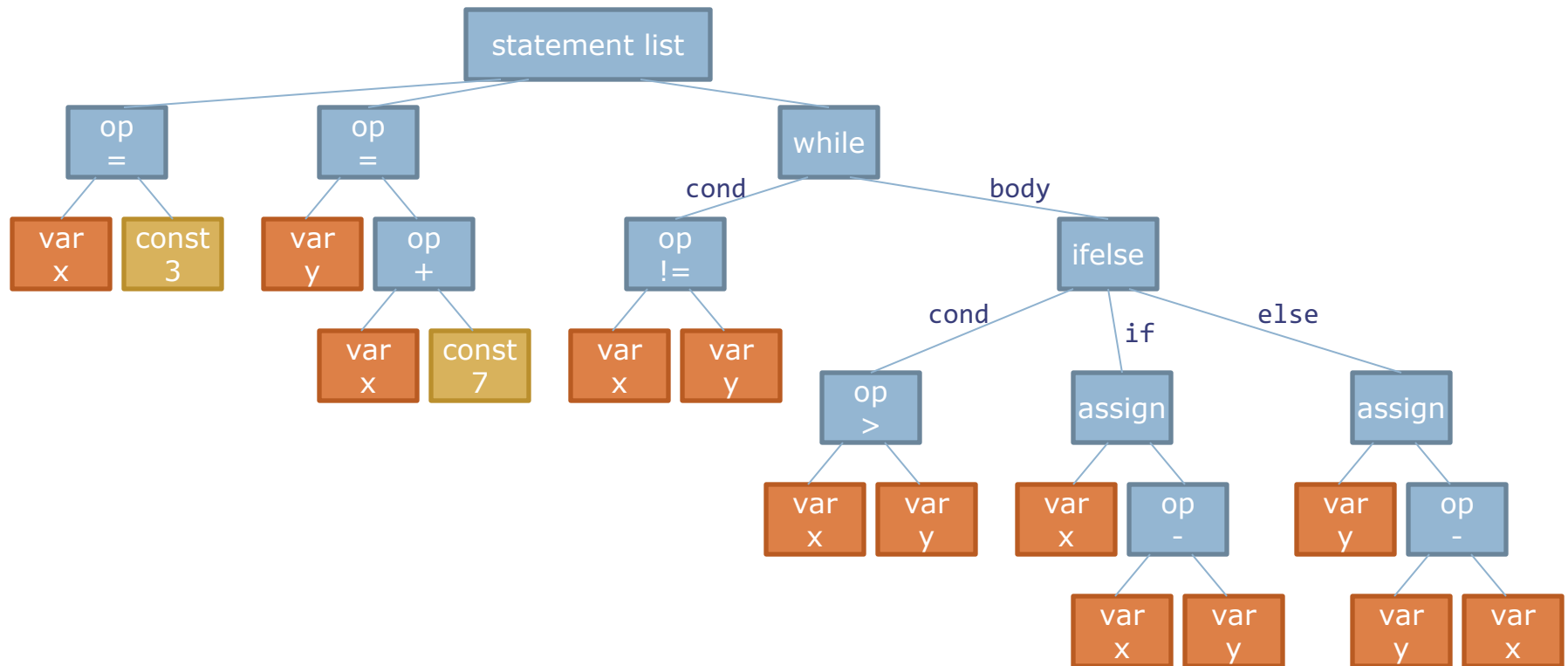
```
int x = 3;
int y = x + 7;
while (x != y) {
    if (x > y) {
        x = x - y;
    } else {
        y = y - x;
    }
}
```



```
("int", KEYWORD),
("x", IDENTIFIER),
("=", OPERATOR),
("3", INT_CONSTANT),
(";", SPECIAL_SYMBOL),
("int", KEYWORD),
("y", IDENTIFIER),
("=", OPERATOR),
("x", IDENTIFIER),
("+", OPERATOR),
("7", INT_CONSTANT),
(";", SPECIAL_SYMBOL),
("while", KEYWORD),
("(", SPECIAL_SYMBOL),
...
```

Frontend Stages

- Lexical analysis (scanning): Source → Tokens
- Syntactic analysis (parsing): Tokens → Syntax tree



Frontend Stages

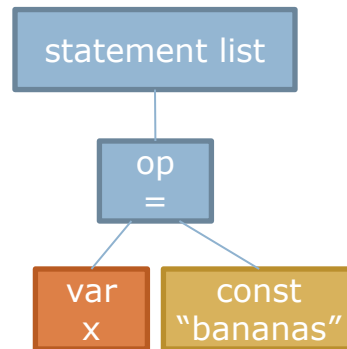
- Lexical analysis (scanning): Source → Tokens
- Syntactic analysis (parsing): Tokens → Syntax tree
- Semantic analysis (mainly, type checking)

Consider:

```
int x = "bananas";
```

Syntax OK

Semantically (meaning)
WRONG



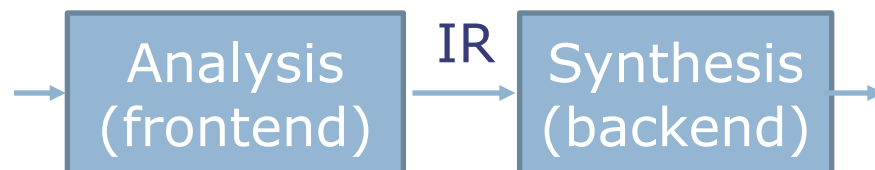
Var	Type
x	int

Line 1: error, invalid conversion from string
constant to int

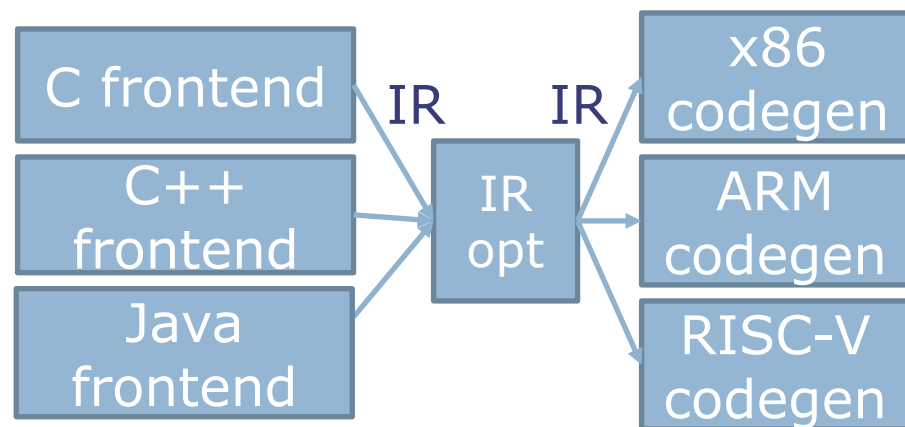
Intermediate Representation (IR)

- Internal compiler language that is:

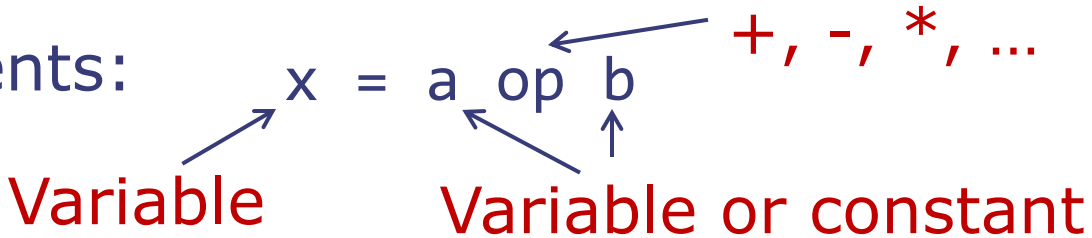
- Language-independent
- Machine-independent
- Easy to optimize



- Why yet another language?
 - Assembly does not have enough info to optimize it well
 - Enables modularity and reuse



Common IR: Control Flow Graph

- Assignments:


Variable Variable or constant

- Basic block: Sequence of assignments with an optional branch at the end

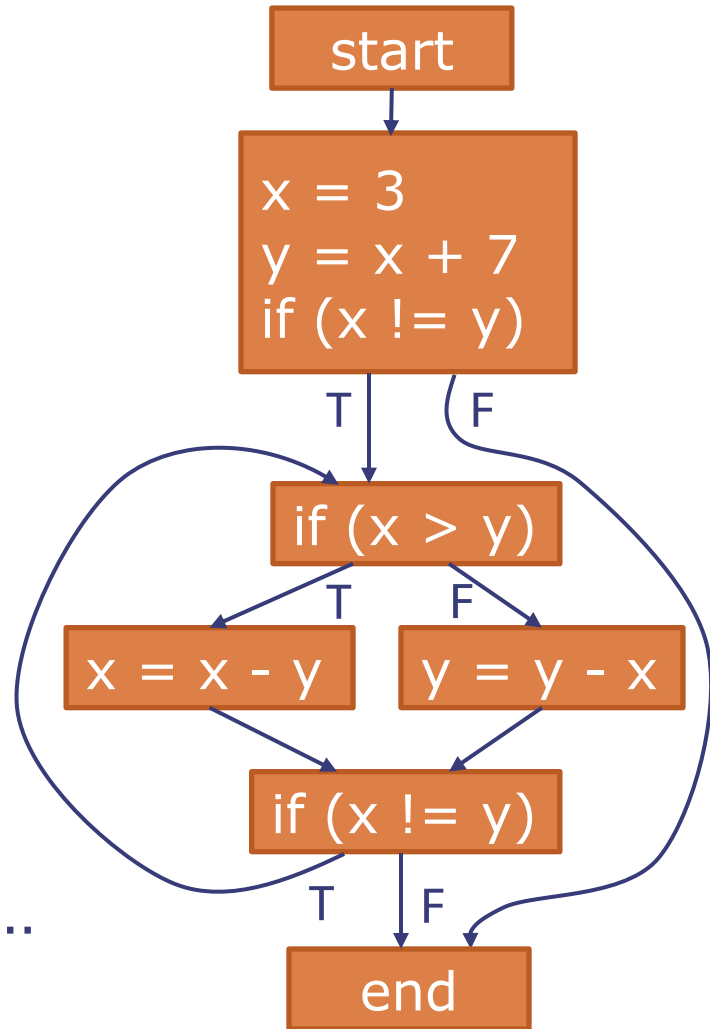
```
x = 3
y = x + 7
if (x != y)
```

- Control flow graph:
 - Nodes: Basic blocks
 - Edges: Jumps or branches between basic blocks

Control Flow Graph for GCD

```
int x = 3;
int y = x + 7;
while (x != y) {
    if (x > y) {
        x = x - y;
    } else {
        y = y - x;
    }
}
```

Looks like a high-level FSM...

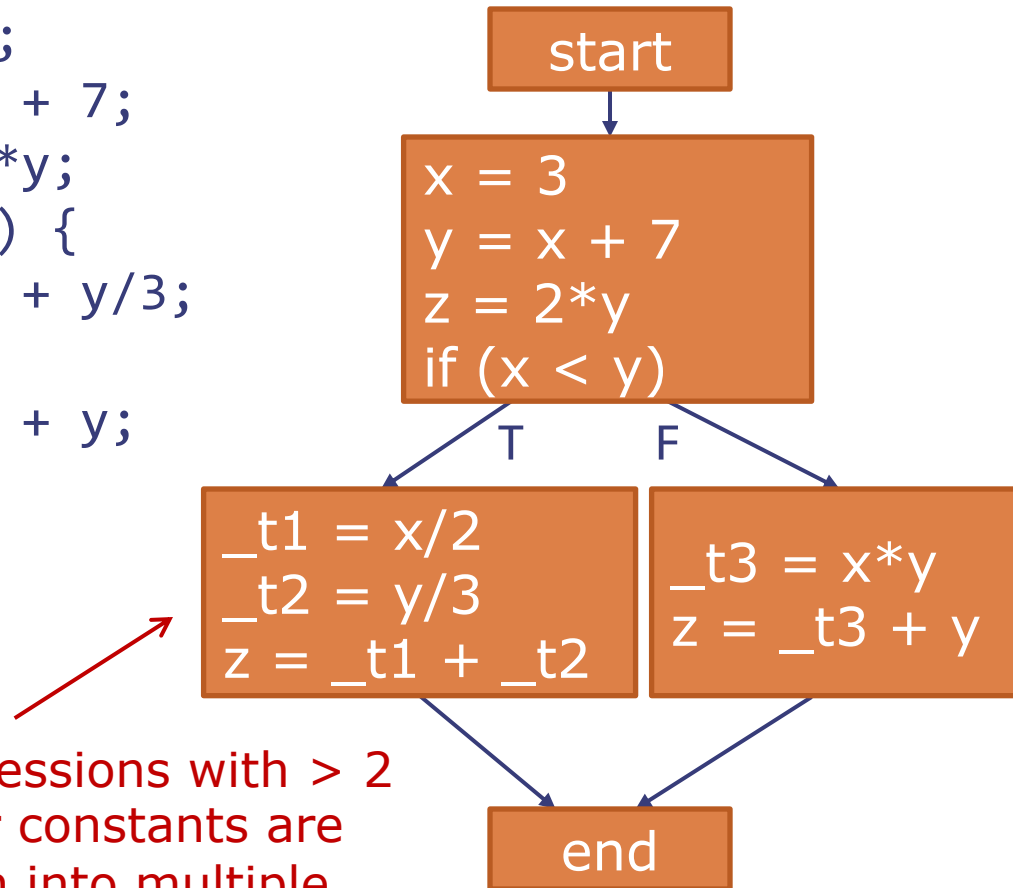


IR Optimization

- Perform a set of passes over the CFG
 - Each pass does a specific, simple task over the CFG
 - By repeating multiple simple passes on the CFG over and over, compilers achieve very complex optimizations
- Example optimizations:
 - Dead code elimination: Eliminate assignments to variables that are never used, or basic blocks that are never reached
 - Constant propagation: Identify variables that are constant, substitute the constant elsewhere
 - Constant folding: Compute and substitute constant expressions

Example IR Optimizations

```
int x = 3;
int y = x + 7;
int z = 2*y;
if (x < y) {
    z = x/2 + y/3;
} else {
    z = x*y + y;
}
```

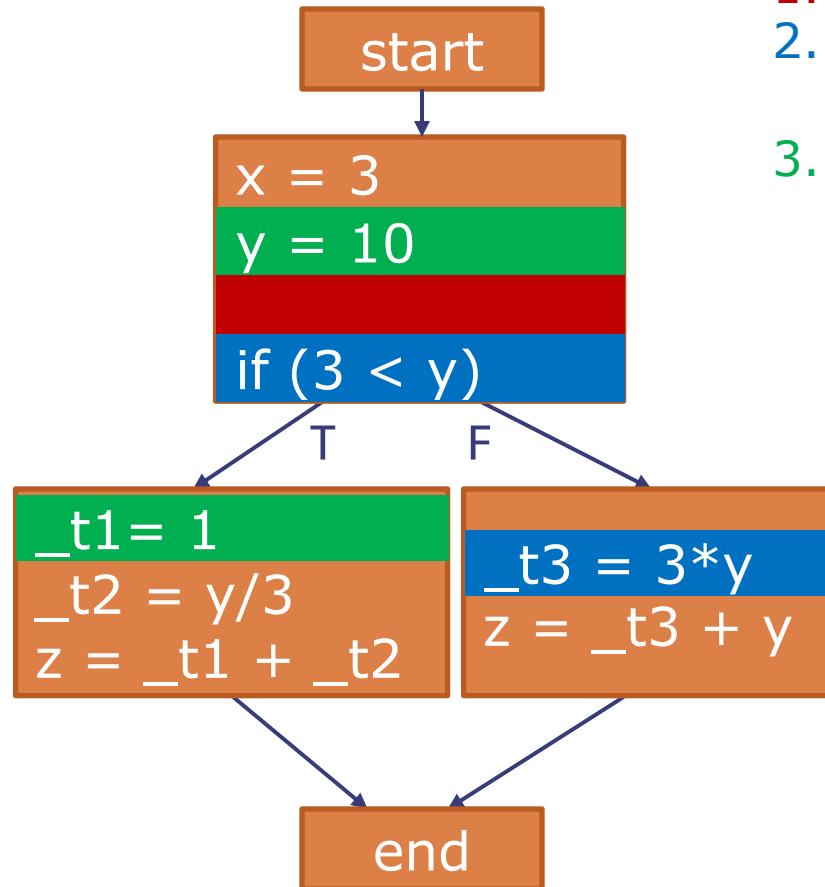


NOTE: Expressions with > 2 ops, vars, or constants are broken down into multiple assignments, using temporary variables

Example IR Optimizations

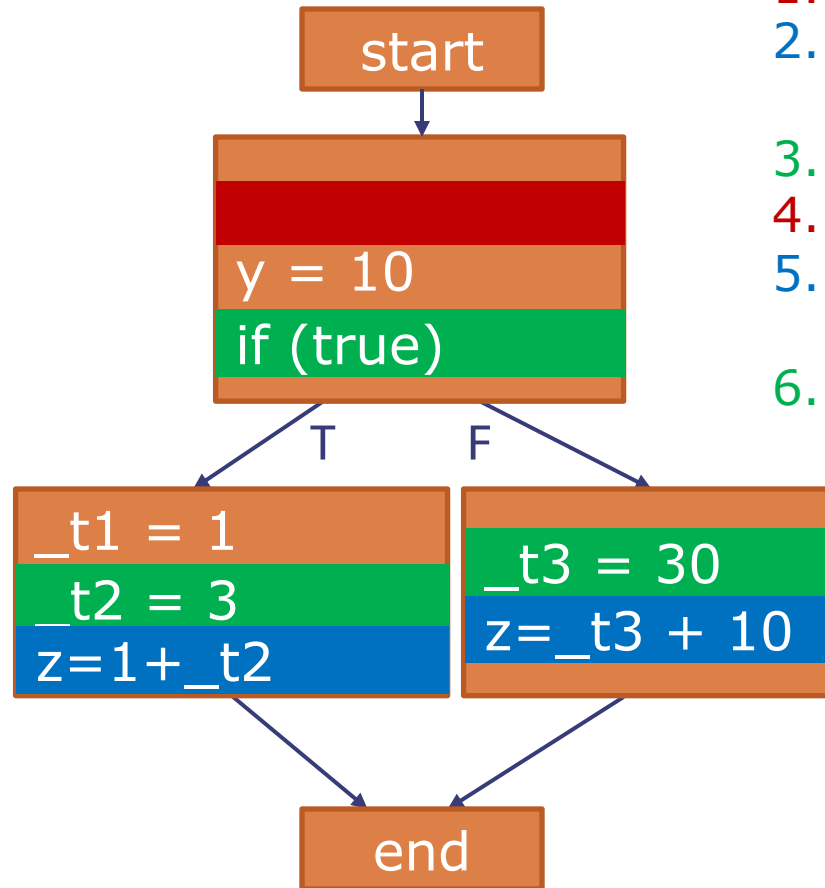
```
int x = 3;
int y = x + 7;
int z = 2*y;
if (x < y) {
    z = x/2 + y/3;
} else {
    z = x*y + y;
}
```

1. Dead code elim
2. Constant propagation
3. Constant folding



Example IR Optimizations

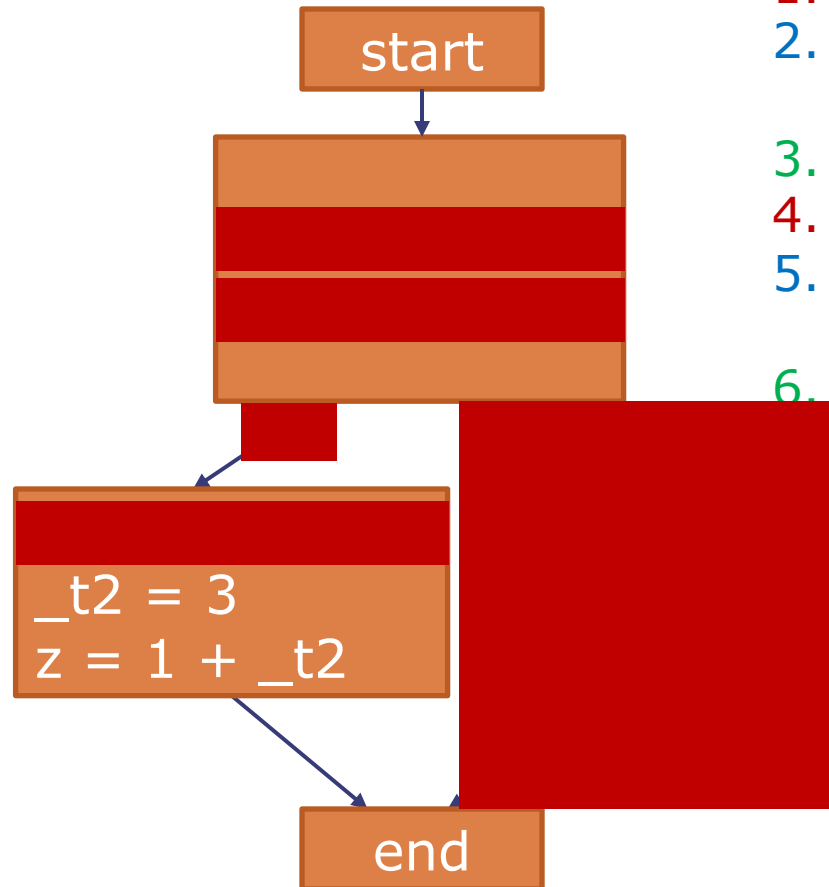
```
int x = 3;
int y = x + 7;
int z = 2*y;
if (x < y) {
    z = x/2 + y/3;
} else {
    z = x*y + y;
}
```



1. Dead code elim
2. Constant propagation
3. Constant folding
4. Dead code elim
5. Constant propagation
6. Constant folding

Example IR Optimizations

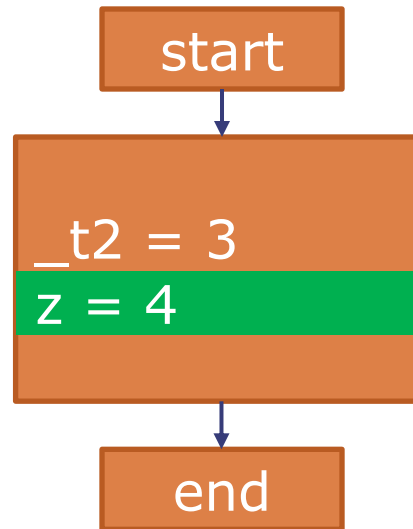
```
int x = 3;
int y = x + 7;
int z = 2*y;
if (x < y) {
    z = x/2 + y/3;
} else {
    z = x*y + y;
}
```



1. Dead code elim
 2. Constant propagation
 3. Constant folding
 4. Dead code elim
 5. Constant propagation
 6. Constant folding
- Dead code elim

Example IR Optimizations

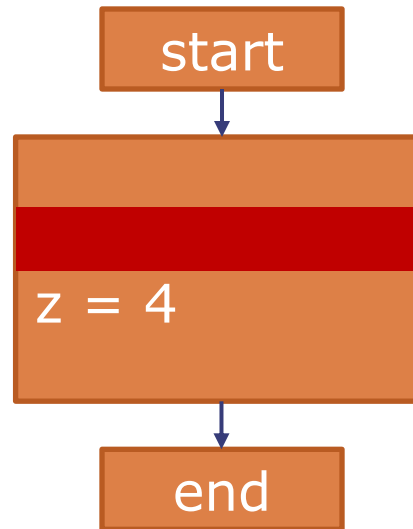
```
int x = 3;
int y = x + 7;
int z = 2*y;
if (x < y) {
    z = x/2 + y/3;
} else {
    z = x*y + y;
}
```



1. Dead code elim
2. Constant propagation
3. Constant folding
4. Dead code elim
5. Constant propagation
6. Constant folding
7. Dead code elim
8. Constant propagation
9. Constant folding

Example IR Optimizations

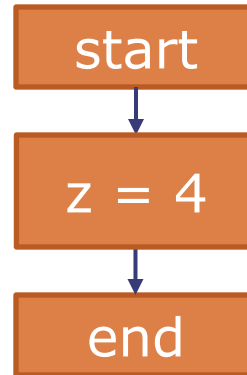
```
int x = 3;
int y = x + 7;
int z = 2*y;
if (x < y) {
    z = x/2 + y/3;
} else {
    z = x*y + y;
}
```



1. Dead code elim
2. Constant propagation
3. Constant folding
4. Dead code elim
5. Constant propagation
6. Constant folding
7. Dead code elim
8. Constant propagation
9. Constant folding
10. Dead code elim
11. Constant propagation
12. Constant folding

Example IR Optimizations

```
int x = 3;
int y = x + 7;
int z = 2*y;
if (x < y) {
    z = x/2 + y/3;
} else {
    z = x*y + y;
}
```



Dumb repetition of
simple transformations on CFGs



Extremely powerful
optimizations

More optimizations by adding passes: Common
subexpression elimination, loop-invariant code
motion, loop unrolling...

1. Dead code elim
2. Constant propagation
3. Constant folding
4. Dead code elim
5. Constant propagation
6. Constant folding
7. Dead code elim
8. Constant propagation
9. Constant folding
10. Dead code elim
11. Constant propagation
12. Constant folding
13. Dead code elim
14. Constant propagation
15. Constant folding

No changes in 13,14,15 → DONE

Code Generation

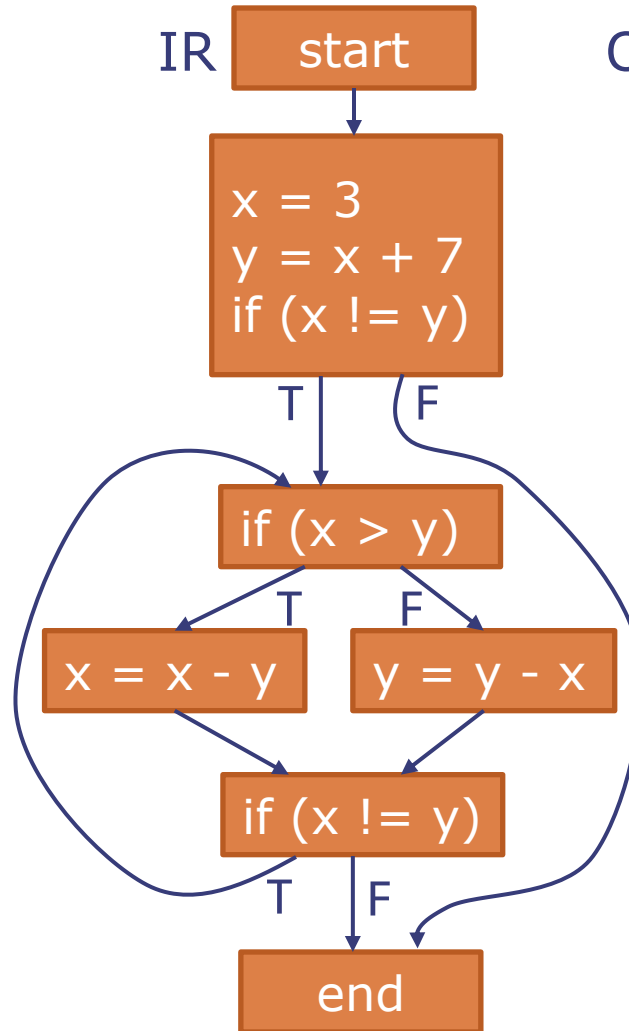
- Translate generated IR to assembly
- Register allocation: Map variables to registers
 - If variables > registers, map some to memory, and load/store them when needed
- Translate each assignment to instructions
 - Some assignments may require > 1 instr if our ISA doesn't have op
- Emit each basic block: label, assignments, and branch or jump
- Lay out basic blocks, removing superfluous jumps
- ISA and CPU-specific optimizations
 - e.g., if possible, reorder instructions to improve performance

Putting It All Together: GCD

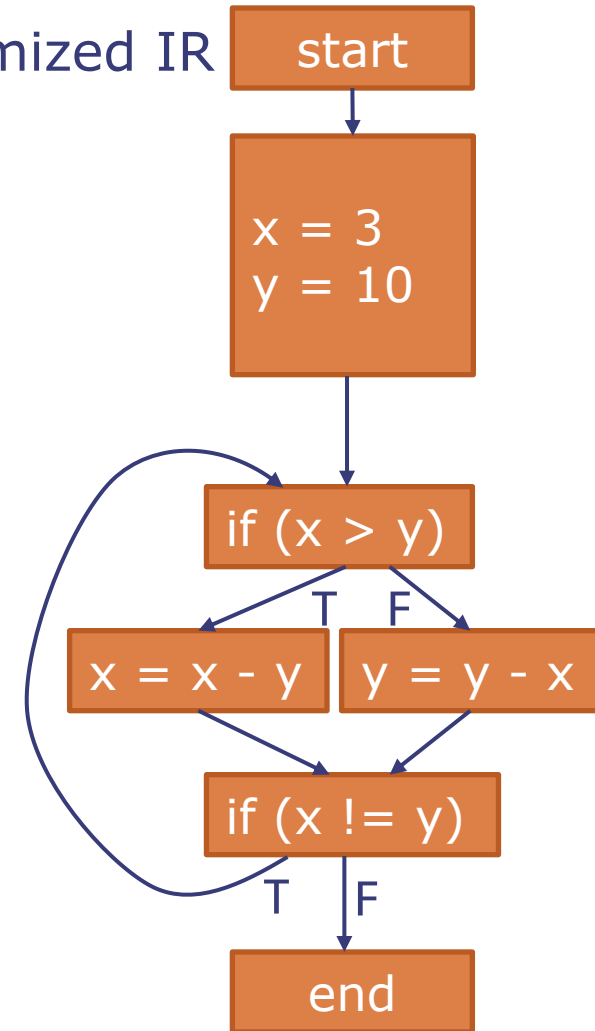
Source code

```
int x = 3;
int y = x + 7;
while (x != y) {
    if (x > y) {
        x = x - y;
    } else {
        y = y - x;
    }
}
```

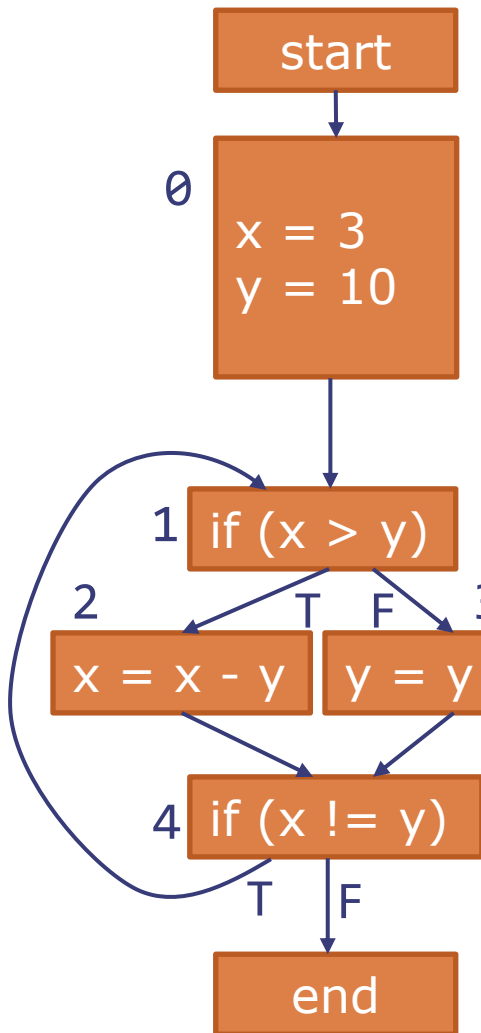
IR



Optimized IR



Putting It All Together: GCD



1. Allocate registers:

`x: t0, y: t1`

2. Produce each basic block for target processor:

BBL0: `li t0, 3`
 `li t1, 10`
 `j BBL1`

BBL1: `slt t2, t1, t0`
 `bnez t2, BBL2`
 `j BBL3`

BBL2: `sub t0, t0, t1`
 `j BBL4`

BBL3: `sub t1, t1, t0`
 `j BBL4`

BBL4: `beq t0, t1, end`
 `j BBL1`

end:

3. Lay out BBLs, removing superfluous branches:

BBL0: `li t0, 3`
 `li t1, 10`

BBL1: `slt t2, t1, t0`
 `bnez t2, BBL2`

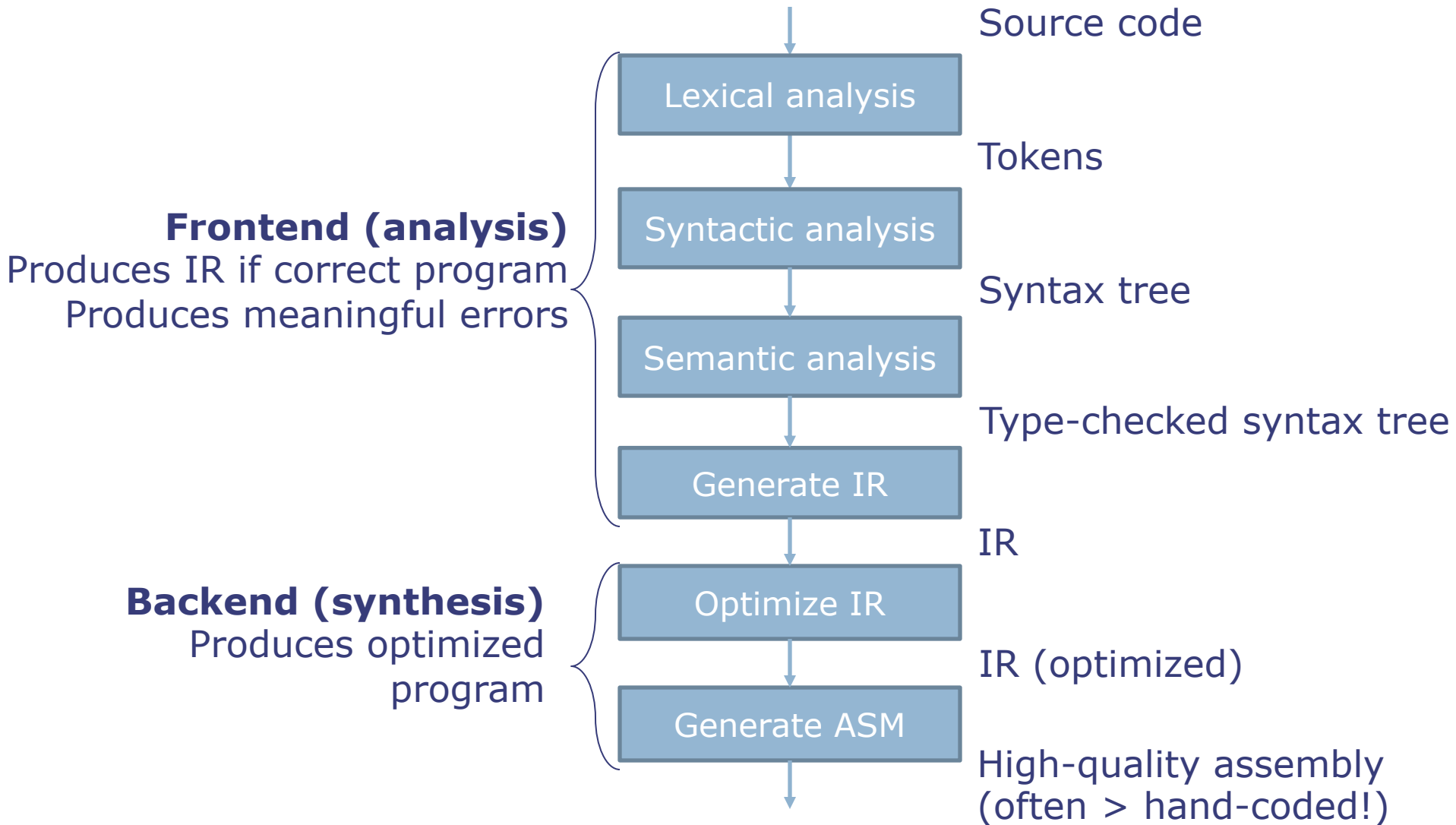
BBL3: `sub t0, t0, t1`
 `j BBL4`

BBL2: `sub t0, t0, t1`

BBL4: `beq t0, t1, end`
 `j BBL1`

end:

Summary: Modern Compilers



Thank you!

Next lecture:
Building a RISC-V Processor