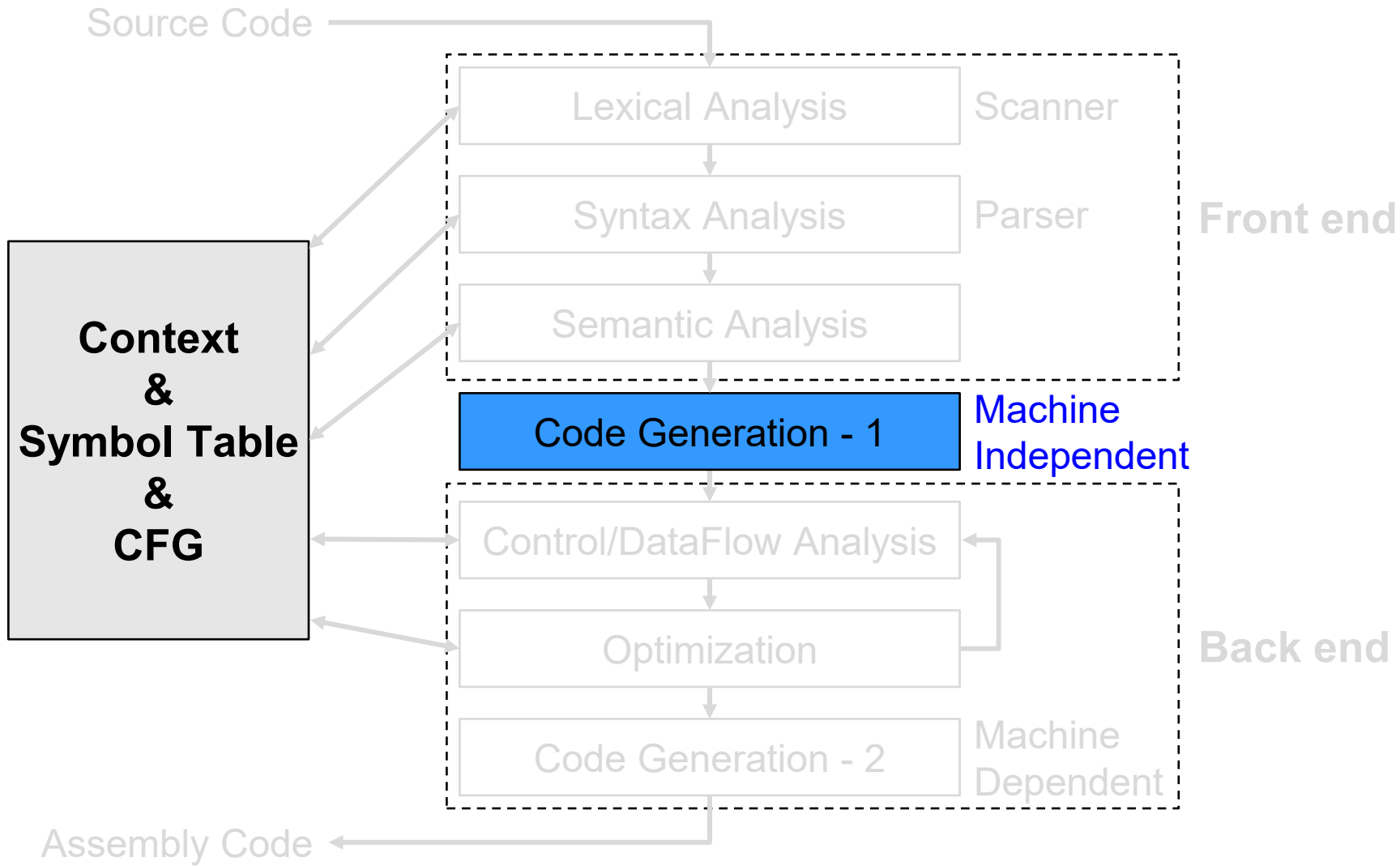
7. Code Generation - 2

2025 Fall

Hunjun Lee

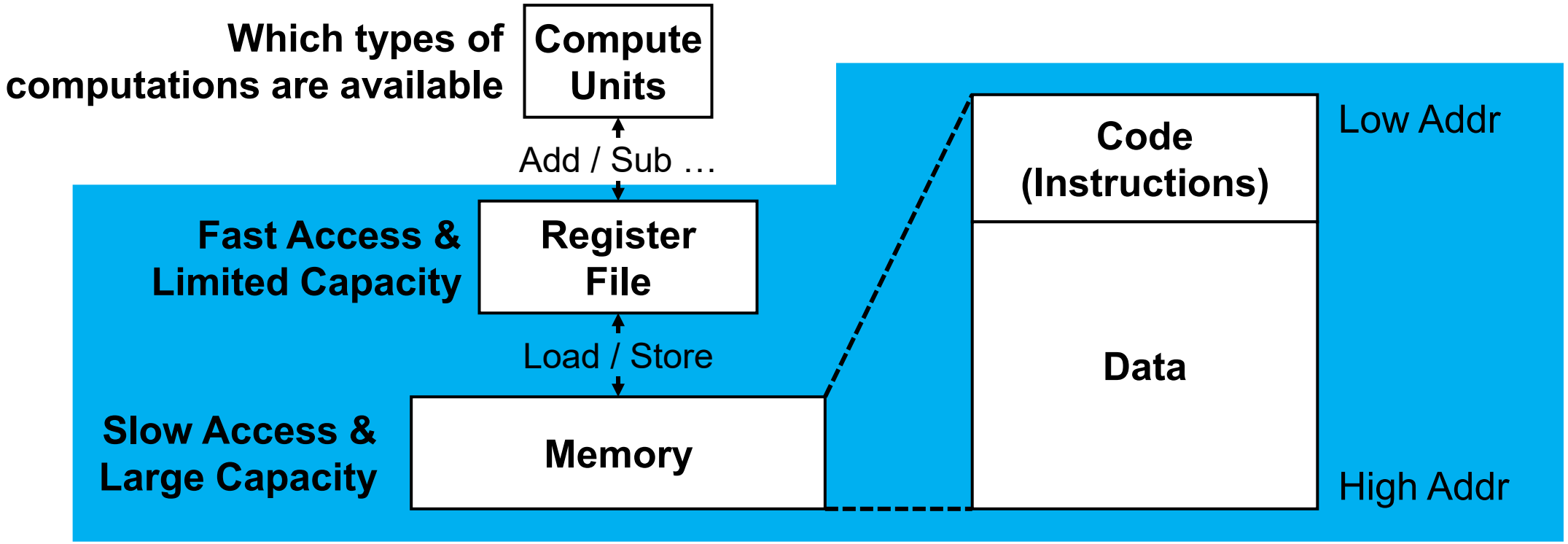
Hanyang University

What You Will Learn



ISA: What the Compiler Knows

- In a “software view”, the CPU consists of a compute units, register file, and memory



Storage Class Selection Problem

- **Determines where to place the data (register file vs. memory)**

- **Standard approach:**
  - Globals / Statics ➔ memory
  - Locals:
    - Composite types (structs, arrays, etc) ➔ memory
    - Rest ➔ Virtual register (this will be mapped in later lectures)

- **All memory approach:**
  - Put all variables into memory

Code Generation Challenge

- **How to generate the code for …**

```
int x;
int y;
x = 3; y = 2;
x + y;
```

- **What about more complex examples**

```
(x + y) + z;
x = func(1+2, 3);
if ((x + z) == 3) { return 3; } else { return func(3); }
```

Simple Model for Code Gen: Stack Machine - 1

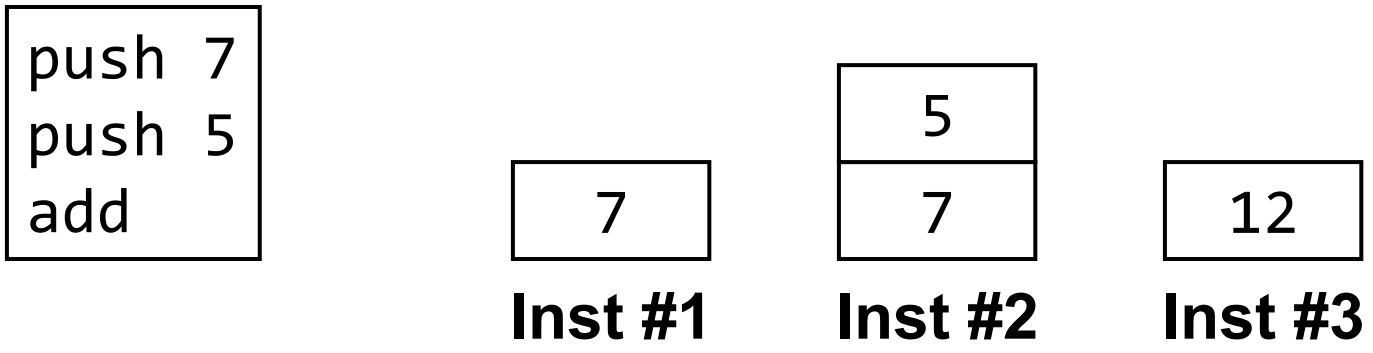
- **Q: How to generate the code for the expressions (after the data placement for the stack is done)**
  - We assume that all the data is stored in the stack @ runtime

- **An instruction has the form of "r = F(a1, …, an)"**
  - Pops n operands from the stack
  - Computes the operation F using the operands
  - Pushes the result r on the stack

Stack Machine Example

- **Consider two instructions for a stack machine**
  - `push i :` pushes an integer i on the stack
  - `add :` add two integers

- **We are trying to execute 7 + 5**



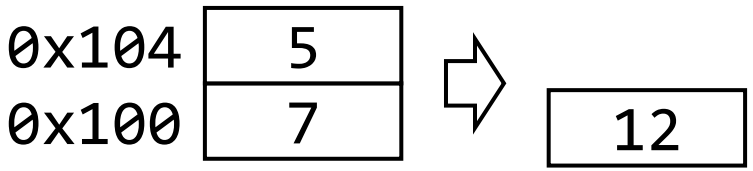
Stack Machine - 2

- **Location of the operands and result is not explicitly stated**
  - They are always on the top of the stack

- **In contrast to a register machine, it enables more compact programs**
  - add instead of add `r1, r2, r3`
  - Java bytecode is a representative usage

Stack Machine - 3

- **In a pure stack machine, an add requires three memory operations**
  - Two memory reads and one write operation (this is slow)




```
load $1, 0x100
load $2, 0x104
add $3, $1, $2
store $3, 0x100
```

Utilizing Register Files - 1

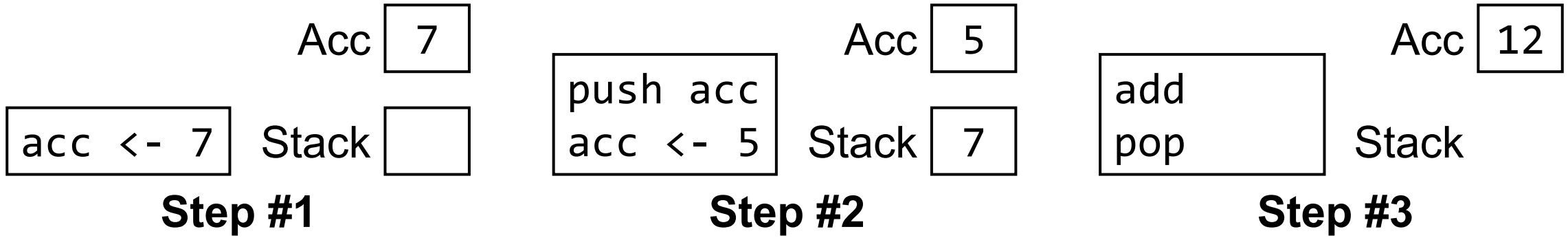
- **We can utilize the register to optimize the stack machine**
  - Keep the top of the stack in a register (register is faster)
  - We can force the add instruction to accumulate the data
  - The add instruction now requires a single memory access

```
acc <- acc + top_of_stack
```

Utilizing Register Files Example

- **Consider four instructions for a stack machine**
  - `acc <- i :` assigns an integer i to the acc register file
  - `push acc :` pushes the integer stored in the register file on the stack
  - `pop :` pops the top of the stack
  - `add :` accumulates the data at the top of the stack to the acc register file

- **We are trying to execute 7 + 5**



Utilizing Register Files - 2

- **In general form, op(e_1, e_2, … e_n) and $e_{1\sim n}$ are expressions**
  - The result of computing an expression is always in the accumulator
  - For each e_i (0 <= i < n) compute e_i and push the result to the stack
  - Pop n-1 values from the stack to compute “op”
  - The stack should be same before and after executing op(e_1, e_2, … e_n)

Complex Example

- **Consider executing 3 + (7 + 5)**
  - e_1: 3 and e_2: (7+ 5)

	Code	Acc	Stack
e1	acc <- 3	3	<init>
	push acc	3	3,<init>
	Acc <- 7	7	3,<init>
	Push acc	7	7,3,<init>
e2	Acc <- 5	5	7,3,<init>
	Add	12	7,3,<init>
	Pop	12	3,<init>
	Add	15	3,<init>
	Pop	15	<init>

Code Generation Using Stack Machine

- **We provide an example code generation using the MIPS processor**
- **The stack is kept in memory and grows towards lower addresses**
- **MIPS assumes two specialized register**
  - The accumulator is kept in `$a0`
  - The address of the next location on the stack is kept in register `$sp`
    - `$sp + 4` is used to address the data at the top of the stack

MIPS Architecture

- **Highly simple reduced instruction set computer (RISC)**
  - Most operations use registers for operands & results
  - Use load & store instructions to use values in memory
  - 32 registers (32 bits each)
    - In this example, we use `$sp` (stack), `$a0` (accumulator), and `$t1` (a temporary register used in operations)

- **There are various documentations for MIPS instruction set architecture (ISA)**
  - ISA indicates the machine-level instructions a processor can execute
  - Microarchitecture indicates how the processor supports the ISA

Example MIPS ISA

- **`lw $1 offset($2)`**
  - Load 32-bit word from address `($2 + offset)` into `$1`
- **`add $1 $2 $3`**
  - `$1 = $2 + $3`
- **`sw $1 offset($2)`**
  - Store 32-bit word in `$1` at address `($2 + offset)`
- **`addi $1 $2 imm`**
  - `$1 = $2 + imm`
- **`li $1 imm`**
  - `$1 = imm`
- **`mv $1 $2`**
  - `$1 = $2`

Converting to MIPS ISA

- The following provides an example of converting stack-machine code to MIPS ISA

- We consider 7 + 5

```
acc <- 7
push acc

acc <- 5
add

pop
```

```
li    $a0   7
sw    $a0   0($sp)
addi  $sp   $sp      -4
li    $a0   5
lw    $t1   4($sp)
add   $a0   $a0      $t1
addi  $sp   $sp      4
```

Generalizing Code Generation - 1

- **We will cover a language with integer and integer operations**

- **You can think of the following CFG**

```
decl_list  ➔ decl; decl_list | decl
type       ➔ int | void
decl       ➔ type id(args) {e}
args       ➔ id, args | id
e          ➔ int | id
                 | if (e1 == e2) {e3} else {e4}
                 | e1 + e2 | e1 – e2
                 | id(e1, e2, …, en)
```

Generalizing Code Generation - 2

- **For each expression “e”, generate MIPS code that:**
  - Computes the value of e in `$a0`
  - Preserves `$sp` and the contents of the stack

- **We define a code generation function cgen(e) which generates the code for the expression e**

- **The code to evaluate a constant is to copy the data**

```
cgen (i) = li $a0 i
```

Code Generation for Add/Sub - 1

- **Example using an addition of two expressions**
  - `cgen(e1 + e2)`

```
cgen(e1 + e2) =
      cgen(e1)          // e1 code gen
      sw $a0 0($sp)     // Store Acc to the Stack
      addi $sp $sp -4
      cgen(e2)          // e2 code gen
      lw $t1 4($sp)     // load the e1 result to temp
      add $a0 $t1 $a0   // add
      addi $sp $sp 4    // pop the stack
```

Class Exercise

- **Generate a code generation template for the following:**
  - `cgen(e1 – e2)`
  - you can use a new ISA: `sub $1 $2 $3`

```
cgen(e1 - e2) =
      cgen(e1)           // e1 code gen
      sw $a0 0($sp)      // Store Acc to the Stack
      addi $sp $sp -4
      cgen(e2)           // e2 code gen
      lw $t1 4($sp)      // load the e1 result to temp
      sub $a0 $t1 $a0    // sub
      addi $sp $sp 4     // pop the stack
```

Code Generation for Add/Sub - 2

- **The code generation for an operation is a template with "pre-defined" instructions before/after generating code for sub-expressions**

```
cgen(e1 + e2) =
      cgen(e1)
      // Do Sth After cgen(e1)
      cgen(e2)
      // Do Sth After cgen(e2)
```

- **Stack machine code generation is recursive**
  - The code generation can be written as a recursive descent of an AST (at least for expressions)

Optimizing Code Generation

- **Can we utilize the register file instead of using the stack**
  - We can rely on $t1 to temporally store the operand

```
cgen(e1 + e2) =
      cgen(e1)
      mv $t1 $a0
      cgen(e2)
      add $a0 $t1 $a0
```

Is this correct?
Consider 1 + (2 + 3)

Code Generation for Branch

- **There are additional MIPS instructions**
  - `beq $1 $2 label`
    - Branch to label if `$1 == $2`
  - `b label`
    - Unconditional jump to `label`

```
cgen(if (e1 == e2) {e3} else {e4}) =
      cgen(e1)
      sw $a0 0($sp)
      addi $sp $sp -4
      cgen(e2)
      lw $t1 4($sp)     // e1 to $t1
      addi $sp $sp 4
      beq $a0 $t1 true_branch
```

```
false_branch:
      cgen(e4)
      b end_if
      …
true_branch:
      cgen(e3)
end_if:
      …
```

Code Generation for Function - 1

- **Code generation for function calls and definitions depends on the layout of the AR**

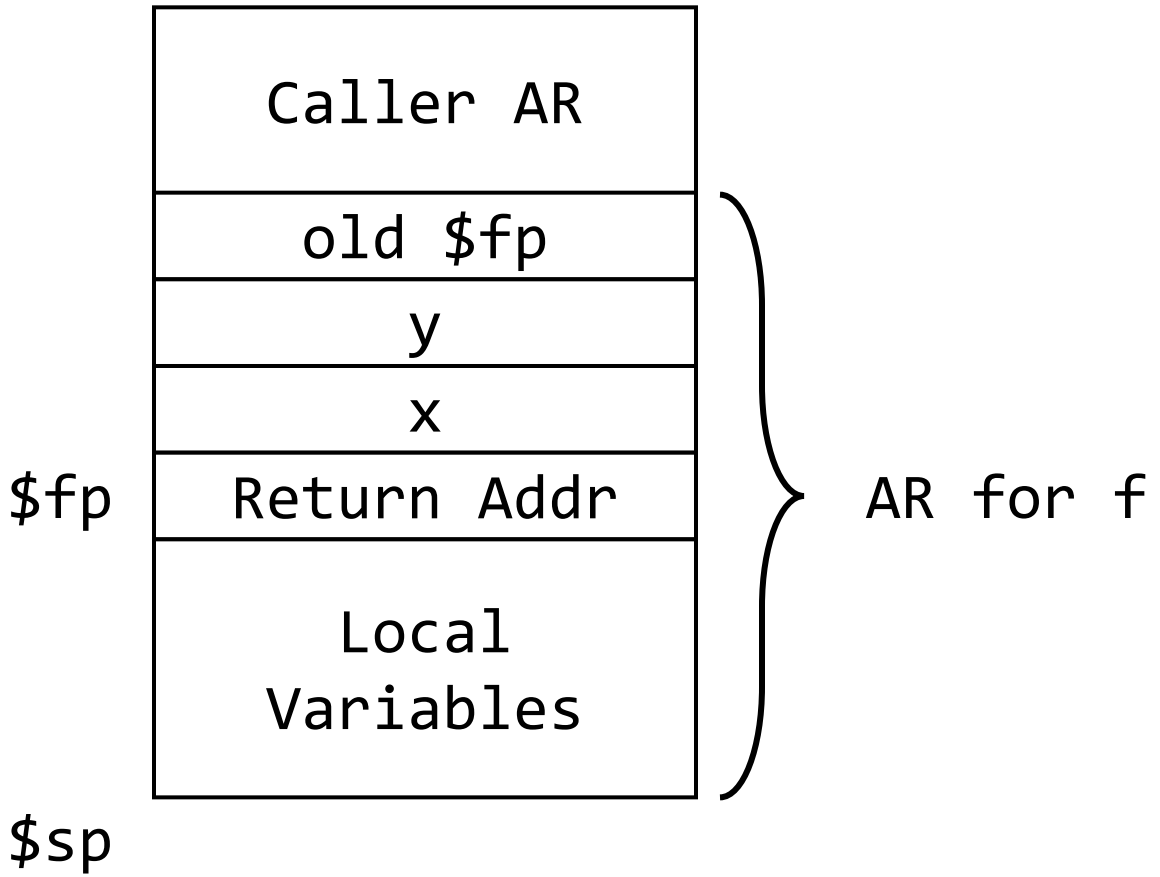
- **A very simple AR layout format:**
  - The result is always in the accumulator
    - No need to store the result in the AR (use register file instead)
  - The AR holds parameters
    - For $f(x_1, \ldots, x_n)$ push parameters $x_1 \sim x_n$ on the stack

Code Generation for Function - 2

- **The stack machine guarantees that `$sp` is the same before and after we enter the function call**
  - Therefore, we do not need an explicit control link at the AR
- **We need the return address at the AR**
- **A pointer to the current activation is useful!**
  - This pointer is set in register `$fp` (frame pointer)

AR for Function Example

- **Consider a call to f(x, y), the AR is as follows:**



Additional ISA for Function Calls

- **The calling sequence is the instructions (of both caller and callee) to set up a function invocation**

- **New instruction: `jal label`**
  - Jump to the `label` and save the next address in `$ra`
  - On some architectures, the return address is stored on the stack by the "call" instruction

- **New instruction: `jr $1`**
  - Jump to address in register `$1`

Function Code Generation: Caller

- **The following is the stack setup at the caller side**
  - Saves the frame pointer to the stack
  - Saves the actual parameters in reverse order
  - Saves the return address to `$ra`
  - The AR is (currently) 4*n + 4 bytes long

```
// caller side
cgen(f(e1, …, en) {e}) =
       sw $fp 0($sp)
       addi $sp $sp -4
       cgen(en)
       sw $a0 0($sp)
       addi $sp $sp -4
       cgen(en-1)
       …
       cgen(e1)
       sw $a0 0($sp)
       addi $sp $sp -4
       jal f_entry
```

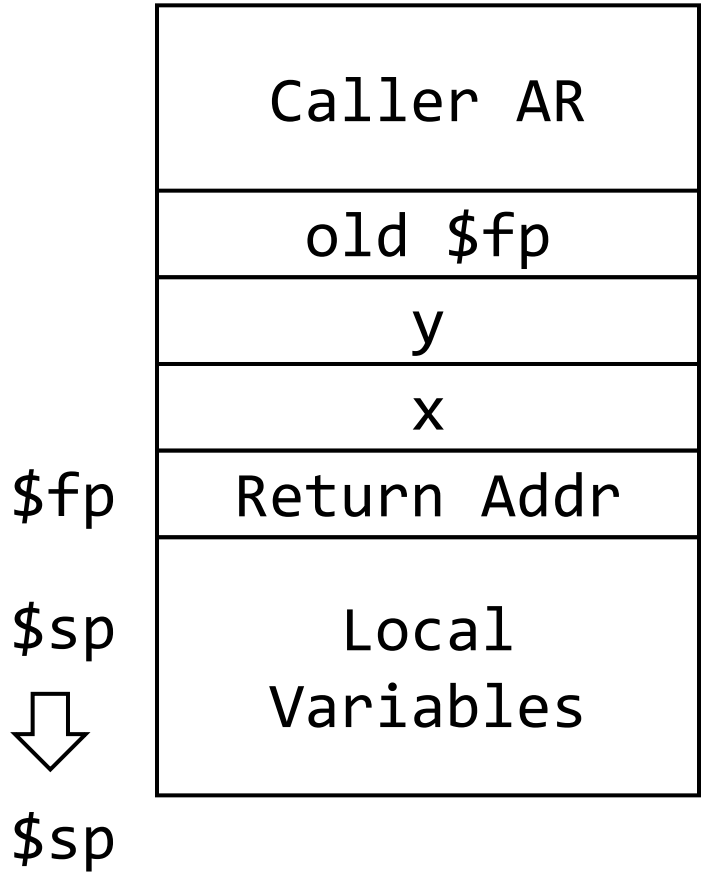
Function Code Generation: Callee

- **The following is the stack setup at the callee side**
  - The frame pointer points to the top of the frame
  - The callee saves the return address
  - Loads the return address and frame pointer after `cgen(e)`
  - Increment the stack pointer by z (the total size of the AR)
    - z = 4*n + 4 bytes (caller) + 4 bytes (callee)

```
// callee side
cgen(f(e1, …, en) {e}) =
f_entry:
        mv $fp $sp
        sw $ra 0($sp)
        addi $sp $sp -4
        cgen(e)
        lw $ra 4($sp)
        addi $sp $sp z
        lw $fp 0($sp)
        jr $ra
```

Referencing Variables

- **The stack pointer (`$sp`) grows dynamically with instructions**
  - It is hard to reference parameters using `$sp`
  - Therefore, it is more plausible to use `$fp` for indexing
    - `lw $x 4($fp)`
    - `lw $y 8($fp)`



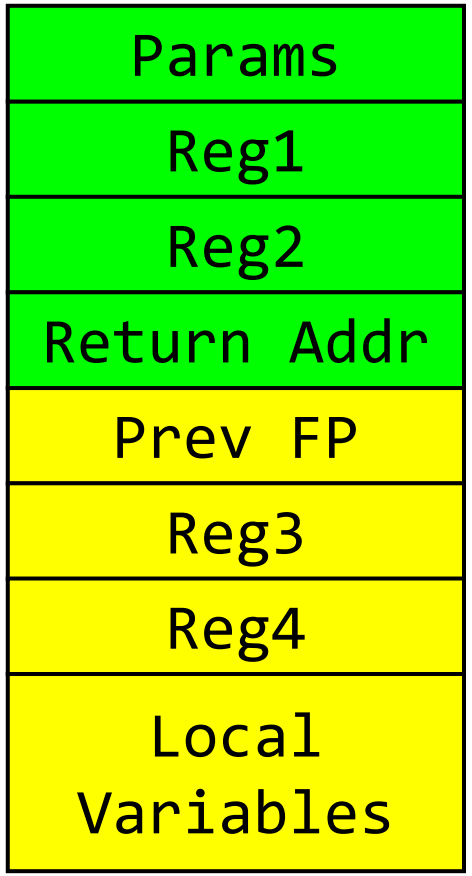
Class Exercise

- **Generate the code for the following function declaration**
  - You can consider only the callee side

```
cgen(int sumto(x) {
      if (x==0) return 0;
      else return x + sumto(x-1);})
```

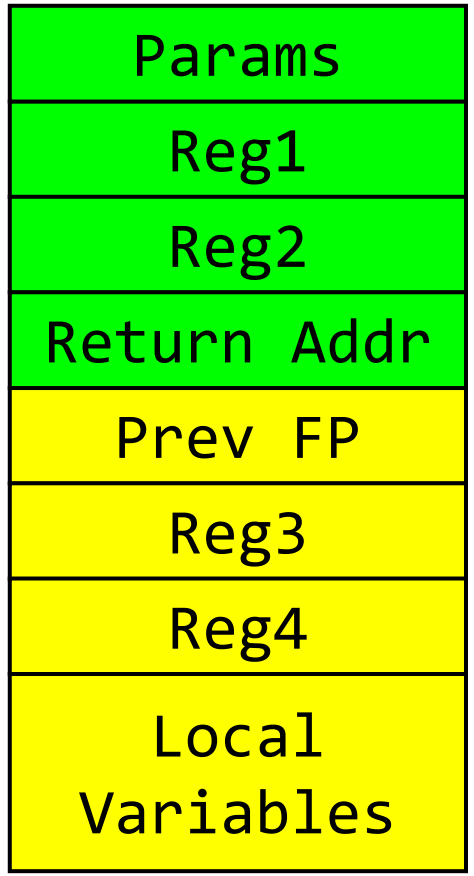
Pushing Values on the Stack

- **Code before call instruction**
  - Push each actual parameter
  - Push caller-saved registers
  - Push return address (current PC) and jump to callee code
- **Prologue = code at function entry**
  - Push dynamic link (i.e., FP)
  - Old stack pointer becomes new frame pointer
  - Push callee-saved registers
  - Push local variables



Popping Values on the Stack

- **Epilogue = code at return instruction**
  - Pop (restore) callee-saved registers
  - Store return value at appropriate place
  - Restore old stack pointer (pop callee frame)
  - Pop old frame pointer
  - Pop return address and jump to that address
- **Code after call**
  - Pop (restore) caller-saved registers
  - Use return value



```
cgen(int sumto(x) {
      if (x==0) return 0;
      else return x + sumto(x-1);})
```

```
sumto_entry:
      mv $fp $sp
      sw $ra 0($sp)
      addi $sp $sp -4
      // subexp if(x == 0)
      lw $a0 4($fp)
      sw $a0 0($sp)
      addi $sp $sp -4 // recall 7+5
      li $a0 0
      lw $t1 4($sp)
      addi $sp $sp 4
      beq $a0 $t1 true1
false1: // x + sum(x-1)
      // subexp x
      lw $a0 4($fp)
      sw $a0 0($sp)
      addi $sp $sp -4
      // cgen(sum(x-1))
      sw $fp 0($sp)
      addi $sp $sp -4
      // cgen(x-1)
      lw $a0 4($fp)
      sw $a0 0($sp)
      addi $sp $sp -4
      li $a0 1
      lw $t1 4($sp)
      sub $a0 $t1 $a0
      addi $sp $sp 4
      sw $a0 0($sp)
      addi $sp $sp -4
      jal sumto_entry
      lw $t1 4($sp)
      add $a0 $t1 $a0
      addi $sp $sp 4
      b end_if1
true1:
      li $a0 0
end_if1: …
```

Temporary Variables - 1

- **There are various intermediate variables that should be stored in the AR**
  - The compiler should statically allocate fixed locations of these variables in the stack

```
int fib (x) {
        if (x == 1) {return 0;}
        else if (x == 2) {return 1;}
        else {return fib(x-1) + fib(x-2);}
}
```

```
int fib (x) {
        if (x == 1) {return 0;}
        else if (x == 2) {return 1;}
        else {
                int temp1 = x - 1;
                int temp2 = fib(temp1);
                int temp1 = x - 2;
                int temp1 = fib(temp1);
                int temp1 = temp1 + temp2;
                return temp1;}
}
```

Temporary Variables - 2

- **We use the function `NT(e)` to determine the number of temporary variables when evaluating `e`**
  - We calculate the maximum number of temporary variables to statically allocate the templates

- **`NT(e1 + e2) = max(NT(e1), NT(e2) + 1)`**
  - The space used for temporaries in `e1` is reused for temporaries in `e2`

Temporary Variables - 3

- **There are a few representative rules**
  - `NT(e1 + e2) = max(NT(e1), NT(e2) + 1)`
  - `NT(e1 - e2) = max(NT(e1), NT(e2) + 1)`
  - `NT(if (e1 == e2) {e3} else {e4} =`
    `max(NT(e1), NT(e2) + 1, NT(e3), NT(e4))`
  - `NT(func(e1, …, en)) = max(NT(e1), …, NT(en))`
    - The results of `e1 ~ en` are not stored in the current activation record, they are stored in the next activation record
  - `NT(NUM) = 0`
  - `NT(id) = 0`

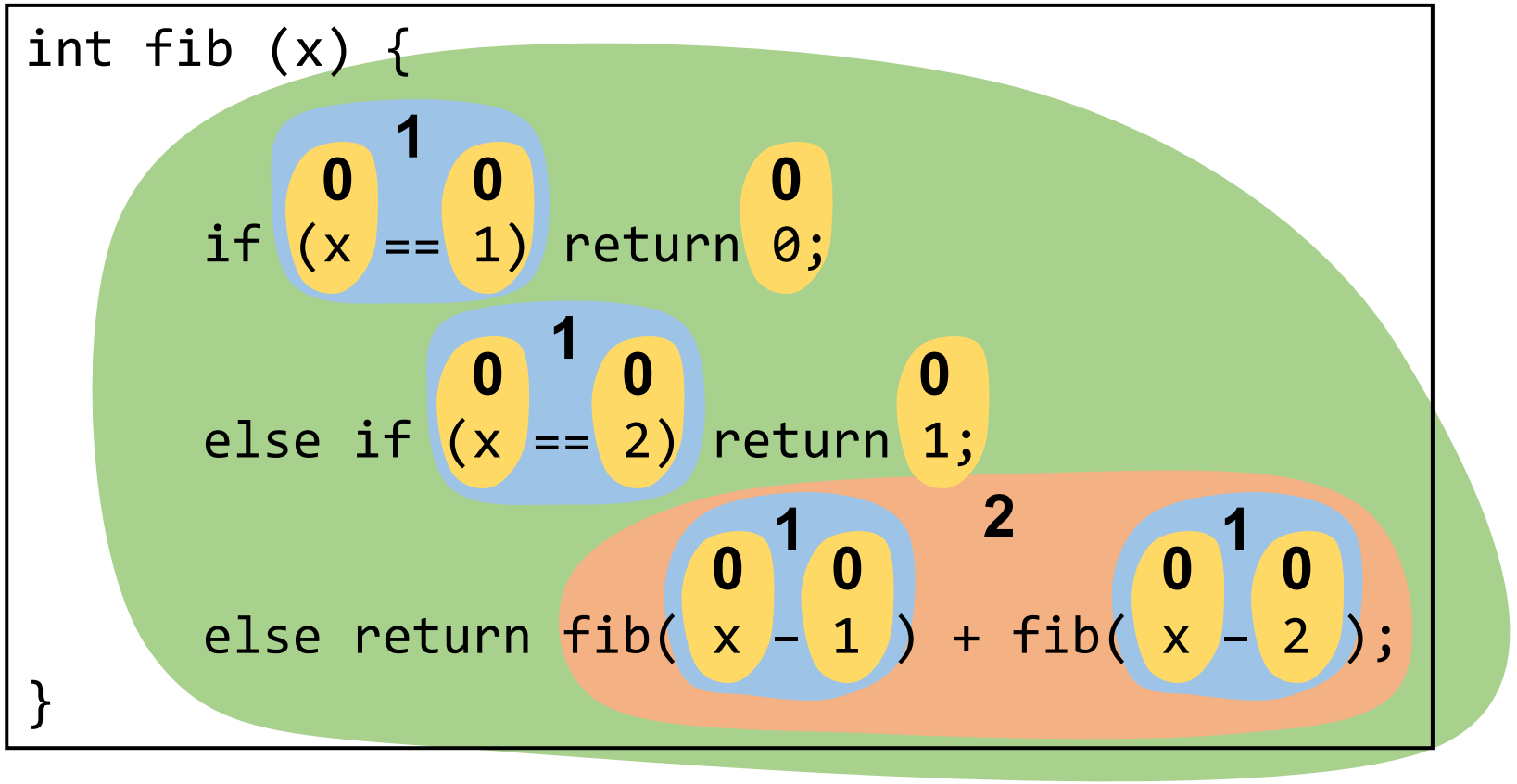
Class Exercise

- **Calculate the NT for the given function**

```
int fib (x) {


      if (x == 1) return 0;


      else if (x == 2) return 1;


      else return fib( x – 1 ) + fib( x – 2 );
}
```

Class Exercise

- **Calculate the NT for the given function**



Code Generation Using NT

- **Code generation utilizes the number of temporaries in use at each point**

- **Add a new argument to the cgen**
  - `cgen(e, nt);` where e is the target expression and `nt` is the position of the next available temporary area

- **The temporary area is used like a small, fixed-size stack**
  - All the computations to increase and decrease the stack pointer for temporaries are now done at compile-time (not run-time)

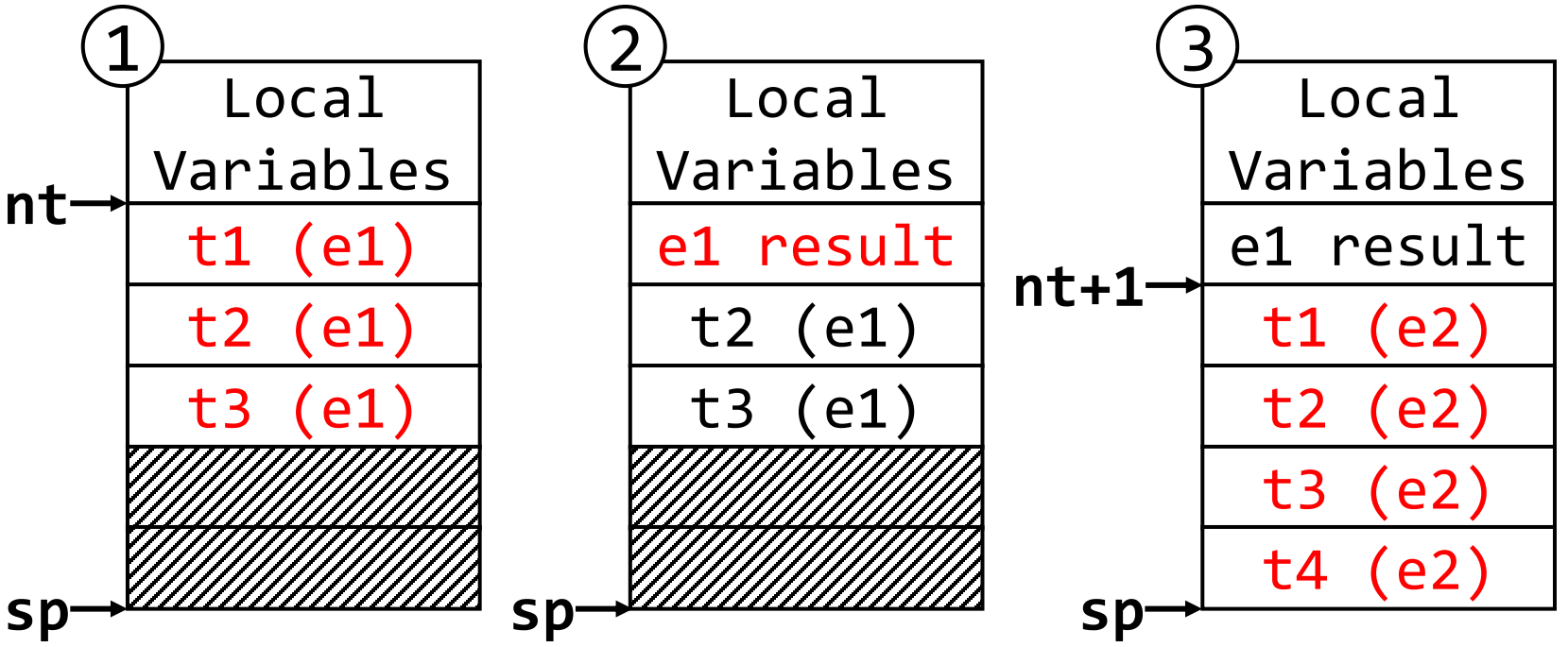
Using NT For Simpler Code Generation

```
cgen(e1 + e2) =
      cgen(e1)
      sw $a0 0($sp)
      addi $sp $sp -4
      cgen(e2)
      lw $t1 4($sp)
      add $a0 $t1 $a0
      addi $sp $sp 4
```

```
cgen(e1 + e2, nt) =
      cgen(e1, nt)
      sw $a0 nt($fp)
      cgen(e2, nt + 4)
      lw $t1 nt($fp)
      add $a0 $t1 $a0
```

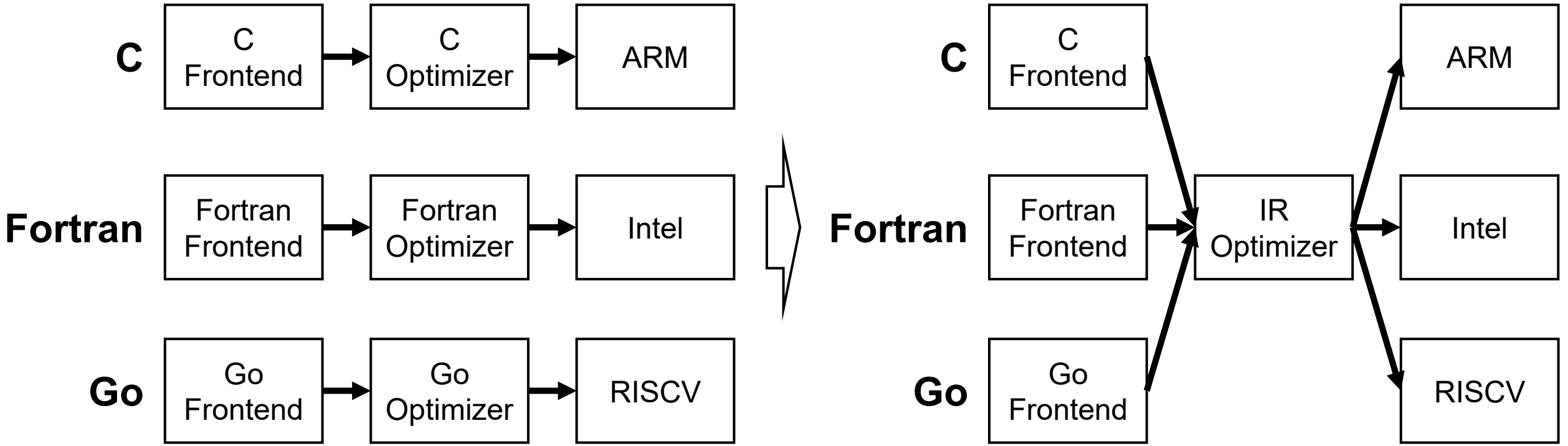
Using NT For Simpler Code Generation

```
cgen(e1 + e2, nt) =
    ①cgen(e1, nt)
    ②sw $a0 nt($fp)
    ③cgen(e2, nt + 4)
      lw $t1 nt($fp)
      add $a0 $t1 $a0

// Assume NT(e1 + e2)
// == 5
```

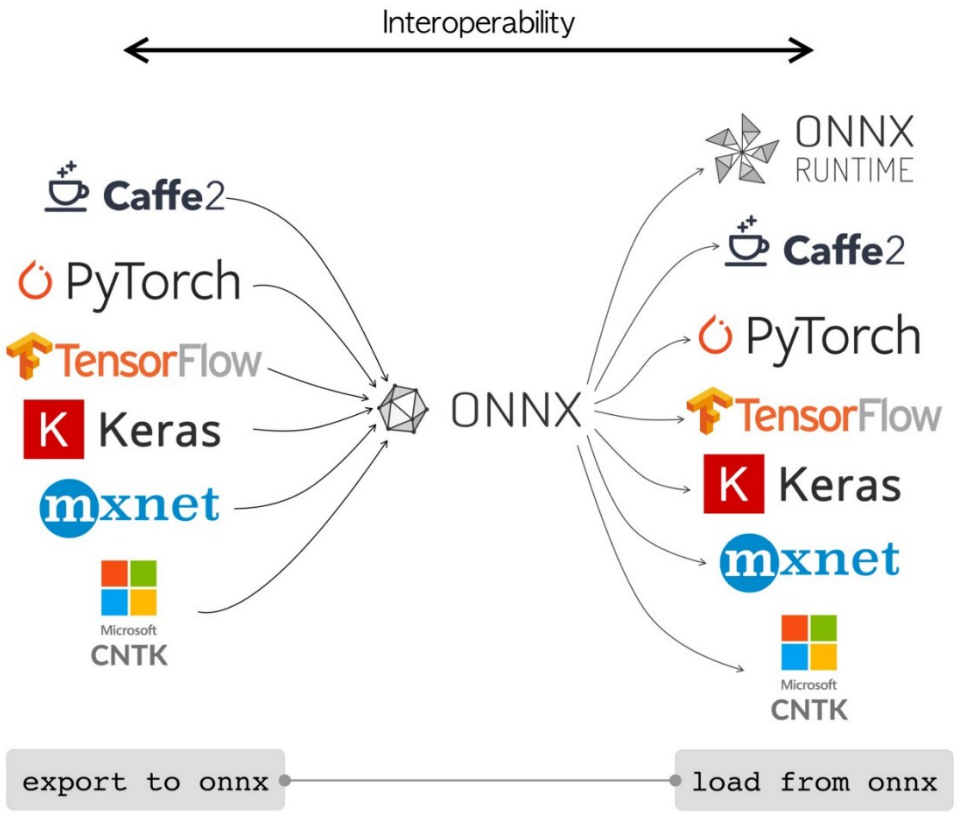


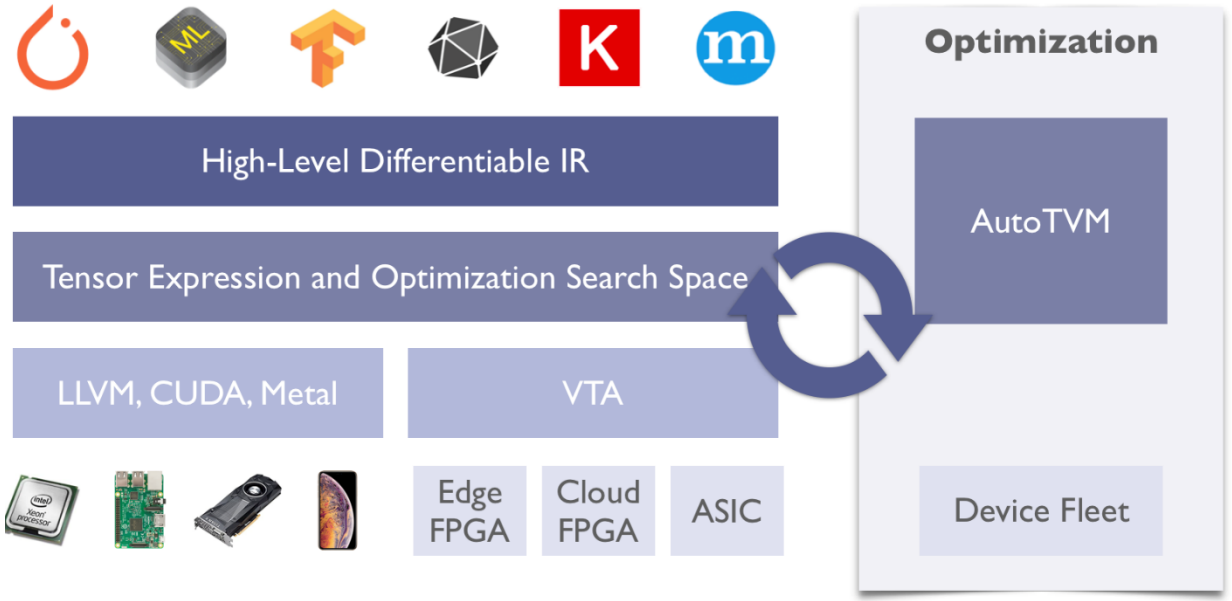
Intermediate Representation (IR) - 1

- **IR is an internal representation of the compilers**
  - Enables language-independent and machine-independent optimization



Intermediate Representation (IR) - 2





High-Level Assembly - 1

- **In this class, we will use high-level assembly as an example**
  - Uses register names, but has an unlimited number
  - Uses control structures like assembly language
    - Explicit jumps and labels
  - Uses opcodes but some are higher level
    - Ex) Push is translated into several assembly instructions

High-Level Assembly - 2

- **Each instruction is of the three-address code**
  - x := y op z (binary op)
  - x := op y (unary op)
  - y and z are registers or constants

- **A complex operation is split into multiple simple operations**
  - k = x + (y * z) is
    - t := y * z
    - k := x + t

Code Generation of IR

- **`igen (e, t)` (you've seen this in the stack machine)**
  - Code to compute the value of e in register t

- **We briefly discuss only a simple example**

```
igen(e1 + e2, t) =
      // t1 and t2 are fresh reg
      igen(e1, t1)
      igen(e2, t2)
      t := t1 + t2
```

- **It is simple (there are unlimited # of registers)**