

Instruction Sets and Code Generation

Eelco Visser



CS4200 | Compiler Construction | November 25, 2021

Operational Semantics

- of ChocoPy

Machine Architecture

- components of a (virtual) machine

RISC-V Instruction Set

- instructions, registers, conventions

Code Generation by Term Transformation

- from source AST to target AST

Compilation Schemas

- how do source language constructs map to machine code

Operational Semantics

Operational Semantics

6 Operational semantics

This section contains the formal operational semantics for the ChocoPy language.

The operational semantics define how every definition, statement, or expression in a ChocoPy program should be evaluated in a given context.

Literals

$$\frac{}{G, E, S \vdash \text{None} : \text{None}, S, -} \quad [\text{NONE}]$$

$$\frac{}{G, E, S \vdash \text{False} : \text{bool}(\text{false}), S, -} \quad [\text{BOOL-FALSE}]$$

$$\frac{}{G, E, S \vdash \text{True} : \text{bool}(\text{true}), S, -} \quad [\text{BOOL-TRUE}]$$

$$\frac{i \text{ is an integer literal}}{G, E, S \vdash i : \text{int}(i), S, -} \quad [\text{INT}]$$

$$\frac{\begin{array}{l} s \text{ is a string literal} \\ n \text{ is the length of the string } s \end{array}}{G, E, S \vdash s : \text{str}(n, s), S, -} \quad [\text{STR}]$$

Expression Statement

$$\frac{G, E, S \vdash e : v, S', -}{G, E, S \vdash e : -, S', -} \quad [\text{EXPR-STMT}]$$

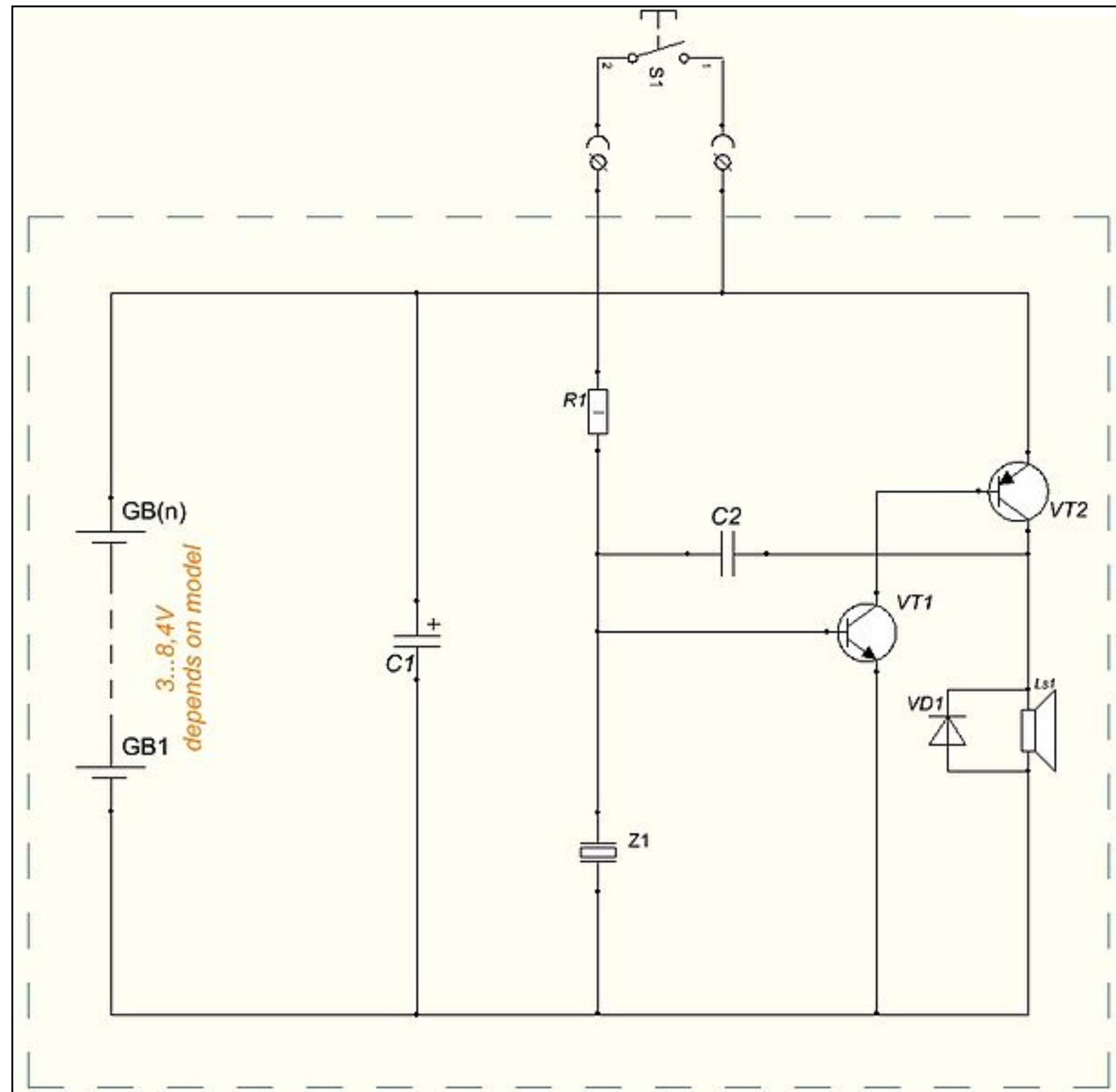
Arithmetic Expressions

$$\frac{G, E, S \vdash e : \text{int}(i_1), S_1, - \\ v = \text{int}(-i_1)}{G, E, S \vdash - e : v, S_1, -} \quad [\text{NEGATE}]$$

$$\frac{G, E, S \vdash e_1 : \text{int}(i_1), S_1, - \\ G, E, S_1 \vdash e_2 : \text{int}(i_2), S_2, - \\ op \in \{+, -, *, //, \%\} \\ op \in \{//, \%\} \Rightarrow i_2 \neq 0 \\ v = \text{int}(i_1 \text{ } op \text{ } i_2)}{G, E, S \vdash e_1 \text{ } op \text{ } e_2 : v, S_2, -} \quad [\text{ARITH}]$$

Machine Architecture

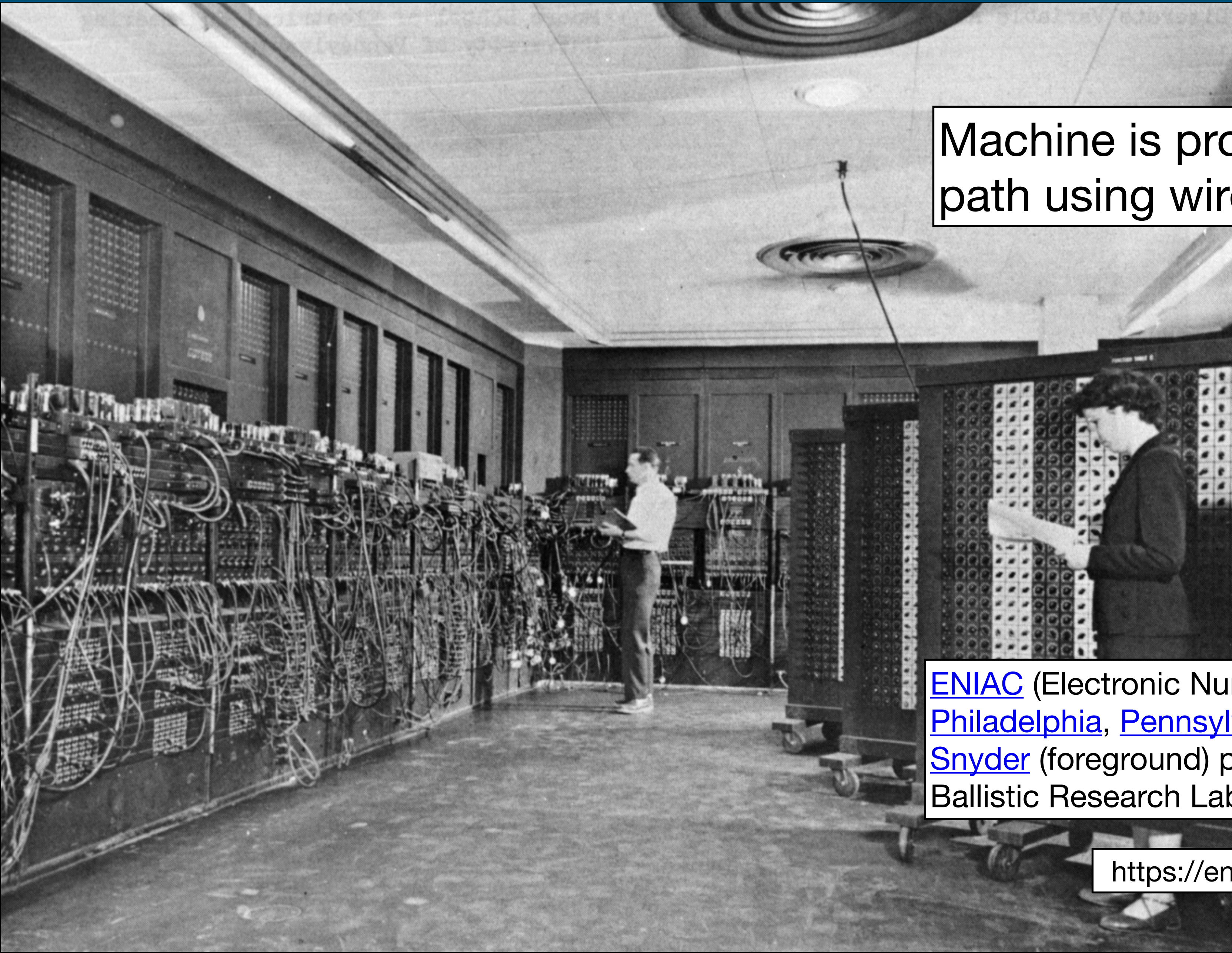
Hard-Wired Programs



Fixed to perform one computation
from input to output

Programmable Machines

Machine is programmed by creating data path using wires



[ENIAC](#) (Electronic Numerical Integrator And Computer) in [Philadelphia, Pennsylvania](#). Glen Beck (background) and [Betty Snyder](#) (foreground) program the ENIAC in building 328 at the Ballistic Research Laboratory (BRL).

<https://en.wikipedia.org/wiki/ENIAC#/media/File:Eniac.jpg>

Stored-Program Computer (Von Neumann Architecture)

Central Processing Unit

- Processor registers
- Arithmetic logic unit

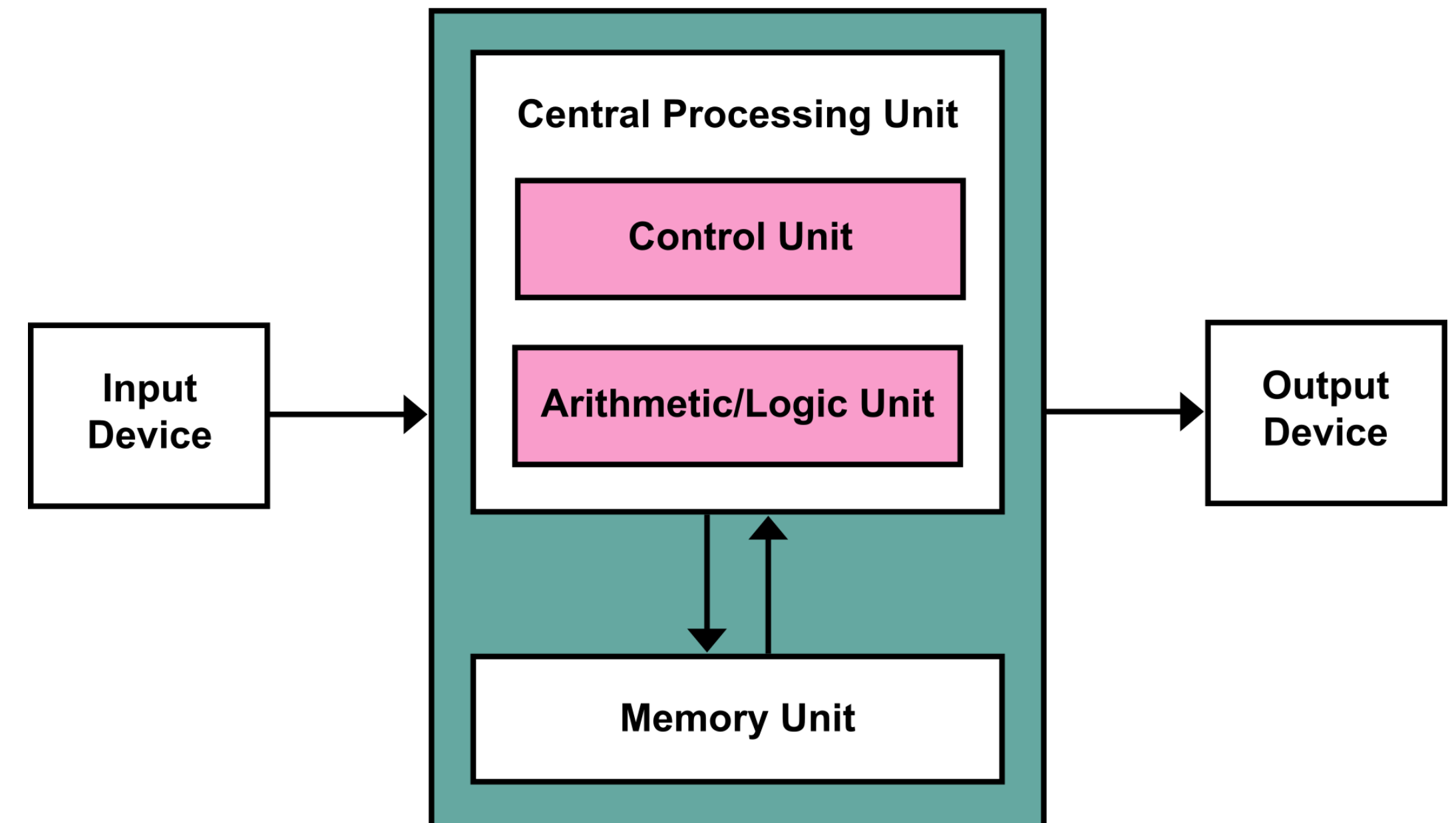
Main Memory

- Stores data and instructions

External Storage

- Persistent storage of data

Input/Output



State

Machine state

- data stored in memory
- memory hierarchy: registers, RAM, disk, network, ...

Imperative program

- computation is series of changes to memory
- basic operations on memory (increment register)
- controlling such operations (jump, return address, ...)
- control represented by state (program counter, stack, ...)

Example: x86 Assembler

```
mov AX [1]
```

```
mov CX AX
```

```
L: dec CX
```

```
mul CX
```

```
cmp CX 1
```

```
ja L
```

```
mov [2] AX
```

Example: x86 Assembler

mov AX [1]

read memory

mov CX AX

L: dec CX

mul CX

cmp CX 1

ja L

mov [2] AX

Example: x86 Assembler

mov AX [1]

read memory

mov CX AX

L: dec CX

mul CX

cmp CX 1

ja L

mov [2] AX

write memory

Example: x86 Assembler

mov AX [1]

read memory

mov CX AX

L: dec CX

mul CX

calculation

cmp CX 1

ja L

mov [2] AX

write memory

Example: x86 Assembler

mov AX [1]

read memory

mov CX AX

L: dec CX

mul CX

calculation

cmp CX 1

ja L

jump

mov [2] AX

write memory

Example: Java Bytecode

```
.method static public m(I)I
```

```
    iload 1  
    ifne else  
    iconst_1  
    ireturn
```


```
else: iload 1  
      dup  
      iconst_1  
      isub  
      invokestatic Math/m(I)I  
      imul  
      ireturn
```

Example: Java Bytecode

```
.method static public m(I)I
```

```
    iload 1  
    ifne else  
    iconst_1  
    ireturn
```

```
else: iload 1  
      dup  
      iconst_1  
      isub  
      invokestatic Math/m(I)I  
      imul  
      ireturn
```



Example: Java Bytecode

```
.method static public m(I)I
```

```
    iload 1  
    ifne else  
    iconst_1  
    ireturn
```

```
else: iload 1
```

	read memory
--	-------------

```
    dup  
    iconst_1
```

isub	calculation
------	-------------

```
    invokestatic Math/m(I)I  
    imul  
    ireturn
```

Example: Java Bytecode

.method static public m(I)I

iload 1

ifne else

jump

iconst_1

ireturn

else:

iload 1

read memory

dup

iconst_1

isub

calculation

invokestatic Math/m(I)I

imul

ireturn

Memory & Control Abstractions

Memory abstractions

- variables: abstract over data storage
- expressions: combine data into new data
- assignment: abstract over storage operations

Control-flow abstractions

- structured control-flow: abstract over unstructured jumps
- ‘go to statement considered harmful’ Edgser Dijkstra, 1968

Example: C

```
int f = 1
int x = 5
int s = f + x

while (x > 1) {
    f = x * f ;
    x = x - 1
}
```

Example: C

```
int f = 1  
int x = 5  
int s = f + x
```

variable

```
while (x > 1) {  
    f = x * f ;  
    x = x - 1  
}
```


Example: C

```
int f = 1  
int x = 5  
int s = f + x  
  
while (x > 1) {  
    f = x * f ;  
    x = x - 1  
}
```

variable

expression

Example: C

```
int f = 1  
int x = 5  
int s = f + x  
  
while (x > 1) {  
    f = x * f ;  
    x = x - 1  
}
```

variable

expression

assignment

Example: C

<code>int f = 1</code>	variable
<code>int x = 5</code>	
<code>int s = f + x</code>	expression
<code>while (x > 1) {</code> <code>f = x * f ;</code> <code>x = x - 1</code> <code>}</code>	control flow
	assignment

Procedural Abstraction

Control-flow abstraction

- Procedure: named unit of computation
- Procedure call: jump to unit of computation and return

Procedural Abstraction

Control-flow abstraction

- Procedure: named unit of computation
- Procedure call: jump to unit of computation and return

Memory abstraction

- Formal parameter: the name of the parameter
- Actual parameter: value that is passed to procedure
- Local variable: temporary memory

Procedural Abstraction

Control-flow abstraction

- Procedure: named unit of computation
- Procedure call: jump to unit of computation and return

Memory abstraction

- Formal parameter: the name of the parameter
- Actual parameter: value that is passed to procedure
- Local variable: temporary memory

Recursion

- Procedure may (indirectly) call itself
- Consequence?

RISC-V Instruction Set

Concrete Syntax

```
.globl main
main:
    lui a0, 8192
    add s11, s11, a0
    jal heap.init
    mv gp, a0
    mv s10, gp
    add s11, s10, s11
    mv ra, zero
    mv fp, zero
    mv fp, zero
    mv ra, zero
    addi sp, sp, -@..main.size
    sw ra, @..main.size-4(sp)
    sw fp, @..main.size-8(sp)
    addi fp, sp, @..main.size
    jal initchars
    li a0, 1
    beqz a0, label_1
    li a0, 0
    seqz a0, a0
label_1:
    .equiv @..main.size, 16
label_0:
    li a0, 10
    ecall
```

Initialize heap size (in multiples of 4KB)
Save heap size
Call heap.init routine
Initialize heap pointer
Set beginning of heap
Set end of heap (= start of heap + heap size)
No normal return from main program.
No preceding frame.
Top saved FP is 0.
No function return from top level.
Reserve space for stack frame.
return address
control link
New fp is at old SP.
Initialize one-character strings.
Load boolean literal: true
Operator and: short-circuit left operand
Load boolean literal: false
Logical not
Done evaluating operator: and

End of program
Code for ecalls: exit

Syntax Definition (*)

```
// RV32I - Base
// Math
Instruction.Add = <add <ID>, <ID>, <ID>> {case-insensitive}
Instruction.Addi = <addi <ID>, <ID>, <IntOrID>> {case-insensitive}
Instruction.And = <and <ID>, <ID>, <ID>> {case-insensitive}
Instruction.Andi = <andi <ID>, <ID>, <IntOrID>> {case-insensitive}
Instruction.Or = <or <ID>, <ID>, <ID>> {case-insensitive}
Instruction.Ori = <ori <ID>, <ID>, <IntOrID>> {case-insensitive}
Instruction.Xor = <xor <ID>, <ID>, <ID>> {case-insensitive}
Instruction.Xori = <xori <ID>, <ID>, <IntOrID>> {case-insensitive}
Instruction.Sub = <sub <ID>, <ID>, <ID>> {case-insensitive}

// Branches
Instruction.Beq = <beq <ID>, <ID>, <IntOrID>> {case-insensitive}
Instruction.Bne = <bne <ID>, <ID>, <IntOrID>> {case-insensitive}
Instruction.Blt = <blt <ID>, <ID>, <IntOrID>> {case-insensitive}
Instruction.Bge = <bge <ID>, <ID>, <IntOrID>> {case-insensitive}
Instruction.Bltu = <bltu <ID>, <ID>, <IntOrID>> {case-insensitive}
Instruction.Bgeu = <bgeu <ID>, <ID>, <IntOrID>> {case-insensitive}

// Misc.
Instruction.Ecall = <ecall>
Instruction.Lui = <lui <ID>, <IntOrID>> {case-insensitive}
Instruction.Auipc = <auipc <ID>, <IntOrID>> {case-insensitive}

// Jumps
Instruction.Jal = <jal <ID>, <IntOrID>> {case-insensitive}
Instruction.Jalr = <jalr <ID>, <ID>, <IntOrID>> {case-insensitive}
```

Abstract Syntax Signature (*)

```
module signatures/RV32IM-sig

imports signatures/Common-sig

signature
  sorts Start Line Label Statement Pseudodirective Instruction IntOrID
  constructors
    Program
      : List(Line) → Start
      : Statement → Line
      : Label → Line
    Label
      : ID → Label
      : INT → IntOrID
      : ID → IntOrID
      : Pseudodirective → Statement
      : Instruction → Statement
    PSData
      : Pseudodirective
    PSText
      : Pseudodirective
    PSSString
      : STRING → Pseudodirective
    PSAsciiiz
      : STRING → Pseudodirective
    PSWord
      : List(IntOrID) → Pseudodirective
    PSSpace
      : INT → Pseudodirective
    ...
    Add
      : ID * ID * ID → Instruction
    Addi
      : ID * ID * IntOrID → Instruction
    And
      : ID * ID * ID → Instruction
    Andi
      : ID * ID * IntOrID → Instruction
    Or
      : ID * ID * ID → Instruction
    Ori
      : ID * ID * IntOrID → Instruction
    ...
```

RISC-V Assembly Programmer's Manual

Load Immediate

The following example shows the `li` pseudo instruction which is used to load immediate values:

```
.equ    CONSTANT, 0xdeadbeef  
  
li      a0, CONSTANT
```

Which, for RV32I, generates the following assembler output, as seen by `objdump` :

```
00000000 <.text>:  
0:    deadc537          lui     a0,0xdeadc  
4:    eef50513          addi    a0,a0,-273 # deadbeef <CONSTANT+0x0>
```

RISC-V Assembly Programmer's Manual

Load Address

The following example shows the `la` pseudo instruction which is used to load symbol addresses:

```
la    a0, msg + 1
```

Which generates the following assembler output and relocations for non-PIC as seen by `objdump` :

```
0000000000000000 <.text>:
0: 00000517          auipc   a0,0x0
           0: R_RISCV_PCREL_HI20    msg+0x1
4: 00050513          mv      a0,a0
           4: R_RISCV_PCREL_L012_I  .L0
```

And generates the following assembler output and relocations for PIC as seen by `objdump` :

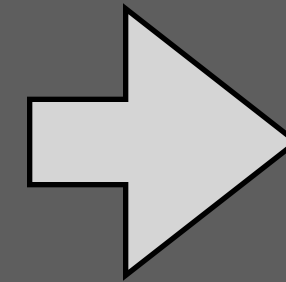
```
0000000000000000 <.text>:
0: 00000517          auipc   a0,0x0
           0: R_RISCV_GOT_HI20       msg+0x1
4: 00053503          ld      a0,0(a0) # 0 <.text>
           4: R_RISCV_PCREL_L012_I  .L0
```

From Concrete Syntax to Abstract Syntax (*)

```
.text
```

```
li a0, 1  
li a1, 15  
ecall
```

```
li a0, 10  
ecall
```

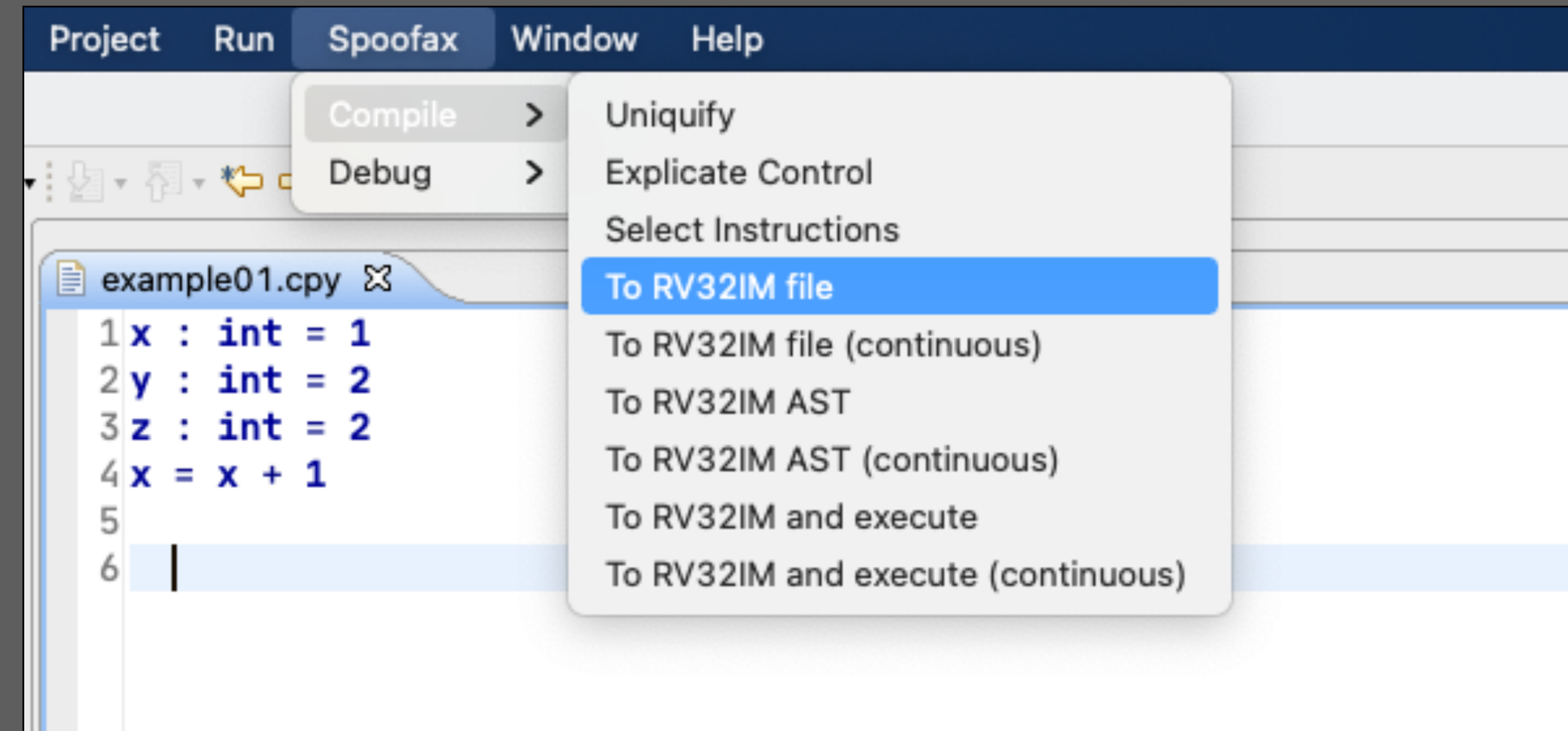


```
Program(  
  [ PSText()  
    , Li("a0", "1")  
    , Li("a1", "15")  
    , Ecall()  
    , Li("a0", "10")  
    , Ecall()  
  ]  
)
```


Code Generation by Term Transformation

Compilation Menu: chocopy.cfg

```
editor-context-menu [  
  menu "Compile" [  
    command-action {  
      command-def = unifyToAstCommand  
      execution-type = Once  
      required-enclosing-resource-types = [Project]  
    }  
    command-action {  
      command-def = explicateControlToAstCommand  
      execution-type = Once  
      required-enclosing-resource-types = [Project]  
    }  
    command-action {  
      command-def = selectInstructionsToAstCommand  
      execution-type = Once  
      required-enclosing-resource-types = [Project]  
    }  
    command-action {  
      command-def = toFileCommand  
      execution-type = Once  
      required-enclosing-resource-types = [Project]  
    }  
    command-action {  
      command-def = toFileCommand  
      execution-type = Continuous  
      required-enclosing-resource-types = [Project]  
    }  
  ]  
]
```



Invoking the Compiler: chocopy.cfg

```
let toFile = task-def mb.chocopy.show.ShowCompileToRv32ImFile
let toFileCommand = command-def {
  task-def = toFile
  display-name = "To RV32IM file"
  parameters = [
    rootDirectory = parameter {
      type = java mb.resource.hierarchical.ResourcePath
      argument-providers = [EnclosingContext(Project)]
    }
    file = parameter {
      type = java mb.resource.hierarchical.ResourcePath
      argument-providers = [Context(File)]
    }
  ]
}
```

Invoking the Compiler: Java

```
package mb.chocopy.show;

import ...

@ChocopyScope
public class ShowCompileToRv32ImFile implements TaskDef<ShowCompileToRv32ImFile.Args, CommandFeedback> {
    @SuppressWarnings("serial") public static final class Args implements Serializable {
        public final ResourcePath rootDirectory;
        public final ResourcePath file;
        public Args(ResourcePath rootDirectory, ResourcePath file) {
            this.rootDirectory = rootDirectory;
            this.file = file;
        }
        @Override public boolean equals(@Nullable Object o) {
            if(this == o) return true;
            if(o == null || getClass() != o.getClass()) return false;
            Args input = (Args)o;
            if(!rootDirectory.equals(input.rootDirectory)) return false;
            return file.equals(input.file);
        }
        @Override public int hashCode() {
            int result = rootDirectory.hashCode();
            result = 31 * result + file.hashCode();
            return result;
        }
        @Override public String toString() {
            return "ShowCompileToRv32ImFile$Args{" +
                "rootDirectory=" + rootDirectory +
                ", file=" + file +
                '}';
        }
    }

    private final ChocopyAnalyzeFile analyzeFile;
    private final CompileToRv32ImText compileToRv32ImText;

    @Inject public ShowCompileToRv32ImFile(
        ChocopyAnalyzeFile analyzeFile,
        CompileToRv32ImText compileToRv32ImText
    ) {
        this.analyzeFile = analyzeFile;
        this.compileToRv32ImText = compileToRv32ImText;
    }

    @Override public CommandFeedback exec(ExecContext context, Args args) throws IOException {
        final ResourcePath file = args.file;
        return context.require(compileToRv32ImText.createSupplier(analyzeFile.createSupplier(new ChocopyAnalyzeFile.Input(args.rootDirectory,
                                                                                                                                            args.file))))).mapThro

        text → {
            final ResourcePath outputFilePath = file.appendToLeafExtension("rv32im");
            final WritableResource outputFile = context.getWritableResource(outputFilePath);
            outputFile.writeString(text);
            context.provide(outputFile);
            return CommandFeedback.of(ShowFeedback.showFile(outputFilePath));
        },
        e → CommandFeedback.ofTryExtractMessagesFrom(e, file)
    );
}

    @Override public String getId() {
        return getClass().getName();
    }
}
```

The Compiler Pipeline

rules

compile-to-rv32im-ast :: Program → RProgram

compile-to-rv32im-ast =
 compile-cpy-to-cir
 ; compile-cir-to-rv32im
 ; compile-rv32im

rules

compile-cpy-to-cir :: Program → CProgram

compile-cpy-to-cir =
 explicate-types
 ; desugar
 ; uniquify
 ; remove-complex-operands
 ; explicate-control

compile-cir-to-rv32im :: CProgram → RProgram

compile-cir-to-rv32im =
 select-instructions-cprogram

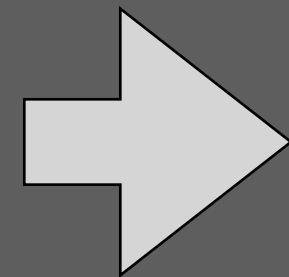
compile-rv32im :: RProgram → RProgram

compile-rv32im =
 assign-homes
 ; patch-instructions

Uniquify

Uniquify

```
x : int = 1
y : int = 2
z : int = 2
x = x + 1
z = x + y
```



```
Program(
  [ VarDef(TypedVar("x3", Type("int")), Int("1"))
    , VarDef(TypedVar("y3", Type("int")), Int("2"))
    , VarDef(TypedVar("z1", Type("int")), Int("2"))
  ]
, [ Assign(
    [Target(Var("x3"))]
    , Add(Var("x3"), Int("1"))
  )
  , Assign(
    [Target(Var("z1"))]
    , Add(Var("x3"), Var("y3"))
  )
]
)
```

Uniquify

rules

```
declare-new-name :: ID → ID
```

```
declare-new-name :
```

```
  x1 → x2
```

```
  with <newname> x1 ⇒ x2
```

```
  with rules( Rename : x1 → x2 )
```

```
rename :: string → string
```

```
rename :
```

```
  x1 → x2
```

```
  with <stx-get-ast-analysis> x1 ⇒ analysis
```

```
  with <stx-get-ast-ref(|analysis)  
      ; if is-list then Hd end> x1 ⇒ d
```

```
  where <Rename>d ⇒ x2
```

rules

```
uniquify      :: Program → Program
```

```
uniquify-gen  :: ? → ?
```

```
uniquify-def  :: Definition → Definition
```

```
uniquify-ref  :: Var → Var
```

```
uniquify =  
  uniquify-gen
```

```
uniquify-gen =  
  topdown(try(is(Definition); uniquify-def))  
  ; topdown(try(is(Var); uniquify-ref))
```

```
uniquify-def :
```

```
  VarDef(TypedVar(x1, t), e1) → VarDef(TypedVar(x2, t), e1)
```

```
  with <declare-new-name> x1 ⇒ x2
```

```
uniquify-ref :
```

```
  Var(x1) → Var(x2)
```

```
  where <rename> x1 ⇒ x2
```

C-IR

'C' Intermediate Representation

signature

sorts CID CINT CProgram CBlock CLabel CTail CStmt
CType CExp CAtom CVar

constructors

	: string → CID
	: string → CINT
CProgram	: List(CBlock) → CProgram
CBlock	: CLabel * CTail → CBlock
CLabel	: CID → CLabel
CReturn	: CExp → CTail
CReturnNone	: CTail
CSeq	: CStmt * CTail → CTail
CVarDec	: CVar * CType * CExp → CStmt
CAssign	: CVar * CExp → CStmt
CIntT	: CType
	: CAtom → CExp
CRead	: CExp
CMin	: CAtom → CExp
CAdd	: CAtom * CAtom → CExp
CMul	: CAtom * CAtom → CExp
CDiv	: CAtom * CAtom → CExp
CInt	: CINT → CAtom
	: CVar → CAtom
CVar	: CID → CVar

Explicate Control: Generating a Control-Flow Graph

```
rules // control-flow graph
```

```
add-cfg-node  :: CBlock → CBlock
```

```
all-cfg-nodes :: List(CBlock) → List(CBlock)
```

```
add-cfg-node =
```

```
  ?block
```

```
  ; rules( CFGNode :+ _ → block )
```

```
all-cfg-nodes =
```

```
  <bagof-CFGNode <+ ![]>()
```

```
rules
```

```
explicate-control :: Program → CProgram
```

```
explicate-control :
```

```
  Program(defs, stms) → CProgram([CBlock(CLabel("Main"), tail2) | blocks])
```

```
  with <explicate-tail-seq> stms ⇒ tail1
```

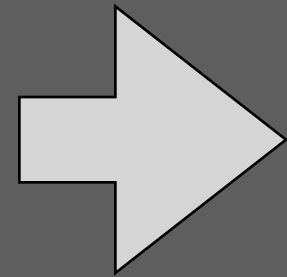
```
  with <explicate-defs(|tail1)> defs ⇒ tail2
```

```
  with <all-cfg-nodes>[] ⇒ blocks
```

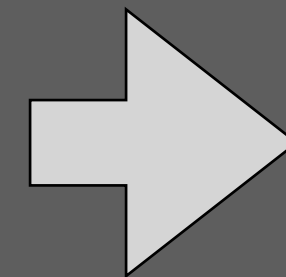
Instruction Selection

Select Instructions: Example

```
z : int = 3  
z + 1
```



```
CProgram(  
  [ CBlock(  
    CLabel("Main")  
    , CSeq(  
      CVarDec(CVar("z4"), CIntT(), CInt("3"))  
      , CReturn(CAdd(CVar("z4"), CInt("1")))  
    )  
  )  
]  
)
```



```
RProgram(  
  [ RPSTData()  
    , RPSText()  
    , RLabel("Main")  
    , RLocal(RVar("z3"), RIntT())  
    , RLi(RVar("z3"), RInt("3"))  
    , RAddi(RReg("a0"), RVar("z3"), RInt("1"))  
  ]  
)
```

Select Instructions: Programs

```
rules select-instructions-cprogram :: CProgram → RProgram
```

```
select-instructions-cprogram :
```

```
  CProgram(blocks) → RProgram(<concat>[
```

```
    [RPSData()],
```

```
    [RPSText()],
```

```
    instrs
```

```
  ])
```

```
  with <mapconcat(select-instrs-block(|"a0"))> blocks ⇒ instrs
```

```
rules select-instrs-block(|string) :: CBlock → List(RLine)
```

```
select-instrs-block(|r) :
```

```
  CBlock(CLabel(lbl), tail) → [RLabel(<cid-to-string>lbl) | instrs]
```

```
  with <select-instrs-tail(|RReg("a0"))> tail ⇒ instrs
```

```
rules select-instrs-tail(|RArg) :: CTail → List(RLine)
```

```
select-instrs-tail(|r) :
```

```
  CReturn(exp) → instrs
```

```
  with <select-instrs-exp(|r)> exp ⇒ instrs
```

```
select-instrs-tail(|r) :
```

```
  CReturnNone() → []
```

Select Instructions: Programs

```
rules select-instructions-cprogram :: CProgram → RProgram
```

```
select-instructions-cprogram :
```

```
  CProgram(blocks) → RProgram(<concat>[
```

```
    [RPSData()],
```

```
    [RPSText()],
```

```
    instrs
```

```
  ])
```

```
  with <mapconcat(select-instrs-block(|"a0"))> blocks ⇒ instrs
```

```
rules select-instrs-block(|string) :: CBlock → List(RLine)
```

```
select-instrs-block(|r) :
```

```
  CBlock(CLabel(lbl), tail) → [RLabel(<cid-to-string>lbl) | instrs]
```

```
  with <select-instrs-tail(|RReg("a0"))> tail ⇒ instrs
```

```
rules select-instrs-tail(|RArg) :: CTail → List(RLine)
```

```
select-instrs-tail(|r) :
```

```
  CReturn(exp) → instrs
```

```
  with <select-instrs-exp(|r)> exp ⇒ instrs
```

```
select-instrs-tail(|r) :
```

```
  CReturnNone() → []
```

Select Instructions: Expressions

```
rules select-instrs-exp(|RArg) :: CExp → List(RLine)

select-instrs-exp(|x) :
  CInt(i) → [RLi(x, <cint-to-rint>i)]

select-instrs-exp(|x) :
  CVar(y) → [RMv(x, RVar(<cid-to-string>y))]

select-instrs-exp(|x) :
  CAdd(y@CVar(_), z@CVar(_)) → [RAdd(x, <cvar-to-rvar>y, <cvar-to-rvar>z)]
```

Compilation Schemas

Abstract From Implementation Details

$$|[\textcolor{green}{i}]|_{r, \text{regs}} \Rightarrow \textcolor{violet}{li} \textcolor{blue}{r}, \textcolor{green}{i}$$

$$|[\textcolor{black}{e} + \textcolor{green}{i}]|_{r, \text{regs}} \Rightarrow |[\textcolor{black}{e}]|_{r, \text{regs}} \\ \textcolor{violet}{addi} \textcolor{blue}{r}, \textcolor{blue}{r}, \textcolor{green}{i}$$

$$|[\textcolor{black}{e1} + \textcolor{blue}{e2}]|_{r1, r2, \text{regs}} \Rightarrow |[\textcolor{black}{e1}]|_{r1, r2, \text{regs}} \\ |[\textcolor{black}{e2}]|_{r2, \text{regs}} \\ \textcolor{violet}{add} \textcolor{blue}{r1}, \textcolor{blue}{r1}, \textcolor{blue}{r2}$$

Except where otherwise noted, this work is licensed under

