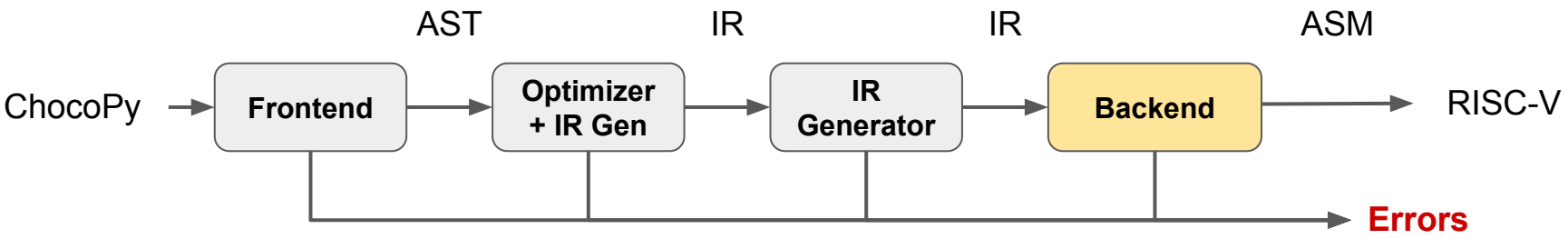
Compiling Techniques

Lecture 15: RISC-V Assembly

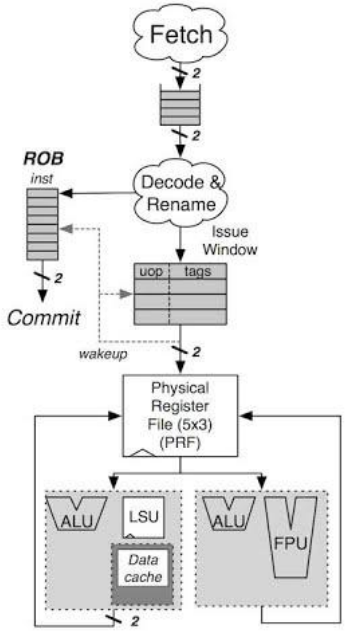
The Backend of a Compiler



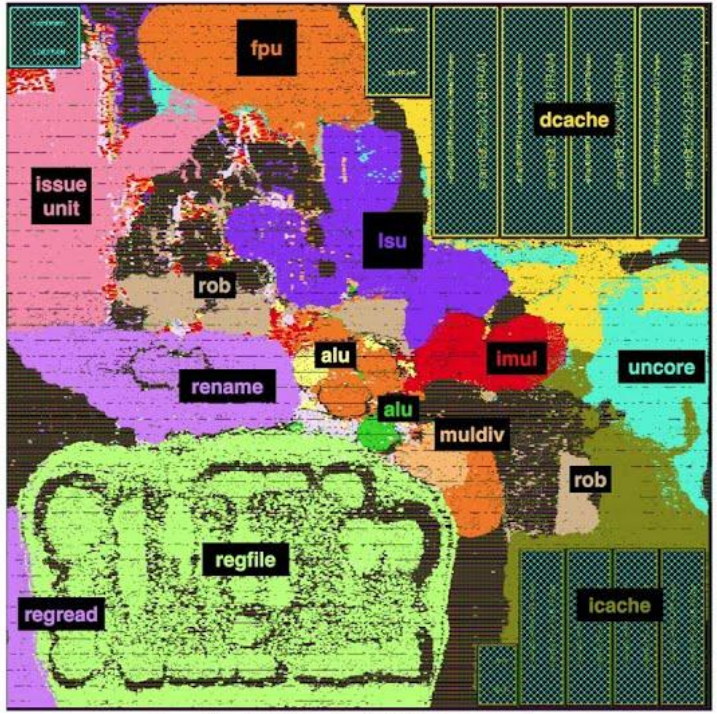
RISC-V: The Free and Open RISC Instruction Set Architecture

Synthesizable

- Targeting ASIC
- Runs on FPGA
  - Zynq zc706



TSMC 45nm floorplan



2-wide BOOM, 16kB L1 caches

1.2 mm²

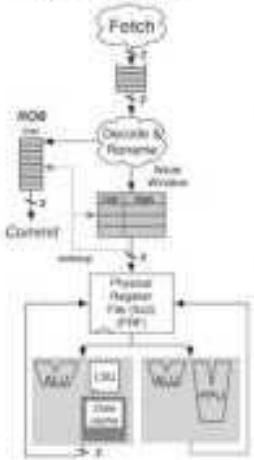
preliminary results

18

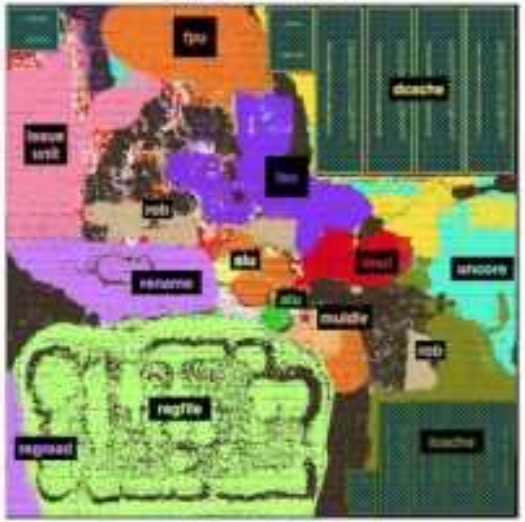
ASPIRE
UC Berkeley

Synthesizable

- Targeting ASIC
- Runs on FPGA
  - Zynq zc706



TSMC 45nm floorplan



2-wide BOOM, 16kB L1 caches
1.2 mm²

preliminary results

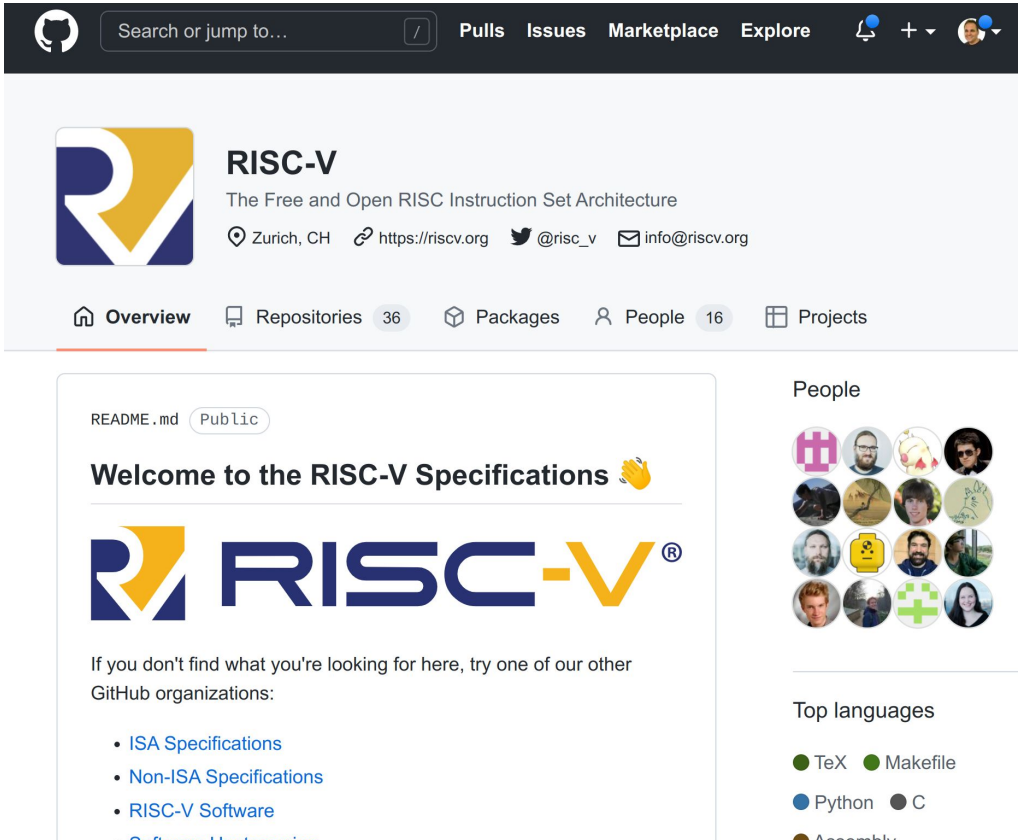
18

The ET-SoC-1 chip features over a thousand RISC-V processors on a single 7nm chip.

ET-SoC-1 features:

- 1088 energy-efficient ET-Minion 64-bit RISC-V in-order cores, each with a custom vector/tensor unit optimized for ML applications
- 4 high-performance ET-Maxion 64-bit RISC-V out-of-order cores for running an OS in self-hosted mode
- Over 160 million bytes of on-chip SRAM
- Interfaces for large external memory with low-power LPDDR4x DRAM and eMMC FLASH
- PCIe Gen4 x8 and other common I/O interfaces

RISC-V is Available on GitHub: Use and Contribute!



Assembly Program Template

.data

Data segment: constant and variable definitions go here
(including statically allocated arrays)

```
name: storage_type value
```

```
var1: .word 3 # one word of storage with initial value 3
```

```
array1: .space 40 # 40 bytes of storage for array1
```

.text

Text segment: assembly instructions go here

Components of an Assembly Program

Category	**Example**
Comment	`# I am a comment`
Assembler directives	`.data, .asciiz`
Operation mnemonic	`add, addi, lw, bne`
Register name	`zero, t3`
Address label (declaration)	`loop1:`
Address label (use)	`loop1`
Integer constant	`8, -4, 0xA9`
Character constant	`‘h’, ‘\t’`
String constant	`“Hello, world\n”`

“Hello World” Example

```
.data
hellostr: .asciiz "Hello world\n"


.text

        li a0, 1                ; Print to stdout
        la a1, hellostr         ; Load message address
        li a2, 12               ; Write 12 bytes
        li a7, 64               ; Write syscall code
        scall

        li a0, 0                ; Set exit code to 0
        li a7, 93               ; Exit syscall code
        scall
```

Executing a RISC-V Program

1) riscv-interpreter

```
tools/riscv-interpreter program.s
```

2) riscemu directly

```
python3 -m riscemu program.s
```

https://github.com/AntonLydike/riscemu

Registers

- 32 general-purpose registers
- two formats for addressing:
  - numbered (`x0` - `x31`)
  - named (`zero`, `a0-a7`)
- holds 32 bits value (= 4 bytes = 1 word)
- stack grows from bottom (high memory) to top (low memory)

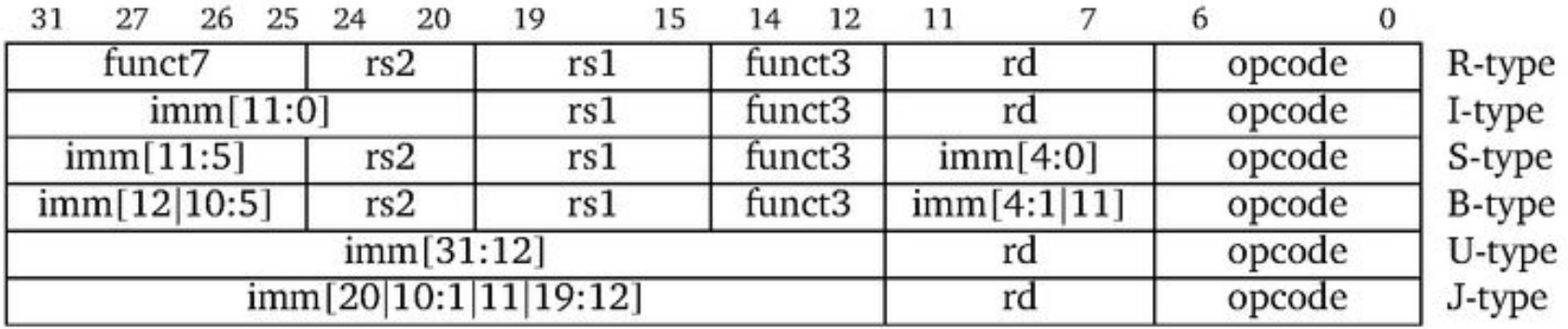
Registers

Register Number	*Name*	*Description*	*Saver*
`x0`	`zero`	hardwired zero	-
`x1`	`ra`	return address	Caller
`x2`	`sp`	stack pointer	Callee
`x3`	`gp`	global pointer	-
`x4`	`tp`	thread pointer	-
`x5 - x7`	`t0 - t2`	temporary registers	Caller
`x8`	`s0 / fp`	saved registers / frame pointer	Callee
`x9`	`s1`	saved registers	Callee
`x10 - x11`	`a0 - a1`	function arguments / return values	Caller
`x12 - x17`	`a2 - a7`	function arguments	Caller
`x18 - x27`	`s2 - s11`	saved registers	Callee
`x28 - x31`	`t3-6`	temporary registers	Caller

Arithmetic Instructions

- Most use three operands
- All operands are registered (no memory access)
- All operands are 4 bytes (a word)

RISC-V Instruction Formats



Arithmetic and Logical Operations (R-Type)

Inst	Name	FMT	Opcode	funct3	funct7	Description (C)	Note
add	ADD	R	0110011	0x0	0x00	rd = rs1 + rs2	
sub	SUB	R	0110011	0x0	0x20	rd = rs1 - rs2	
xor	XOR	R	0110011	0x4	0x00	rd = rs1 ^ rs2	
or	OR	R	0110011	0x6	0x00	rd = rs1 | rs2	
and	AND	R	0110011	0x7	0x00	rd = rs1 & rs2	
sll	Shift Left Logical	R	0110011	0x1	0x00	rd = rs1 << rs2	
srl	Shift Right Logical	R	0110011	0x5	0x00	rd = rs1 >> rs2	
sra	Shift Right Arith*	R	0110011	0x5	0x20	rd = rs1 >> rs2	msb-extends
slt	Set Less Than	R	0110011	0x2	0x00	rd = (rs1 < rs2)?1:0	
sltu	Set Less Than (U)	R	0110011	0x3	0x00	rd = (rs1 < rs2)?1:0	zero-extends

```
add   t0, t1, t2        # Compute ‘t1 + t2` and save result in t0
and   t0, t1, t2        # Compute ‘t1 & t2` and save result in t0
```

Arithmetic and Logical Operations (I-Type)

addi	ADD Immediate	I	0010011	0x0		rd = rs1 + imm	
xori	XOR Immediate	I	0010011	0x4		rd = rs1 ^ imm	
ori	OR Immediate	I	0010011	0x6		rd = rs1 | imm	
andi	AND Immediate	I	0010011	0x7		rd = rs1 & imm	
slli	Shift Left Logical Imm	I	0010011	0x1	imm[5:11]=0x00	rd = rs1 << imm[0:4]	
srli	Shift Right Logical Imm	I	0010011	0x5	imm[5:11]=0x00	rd = rs1 >> imm[0:4]	
srai	Shift Right Arith Imm	I	0010011	0x5	imm[5:11]=0x20	rd = rs1 >> imm[0:4]	msb-extends
slti	Set Less Than Imm	I	0010011	0x2		rd = (rs1 < imm)?1:0	
sltiu	Set Less Than Imm (U)	I	0010011	0x3		rd = (rs1 < imm)?1:0	zero-extends

```
addi  t0, t1, 12        # Compute ‘t1 + 12` and save result in t0
and   t0, t1, 0         # Compute ‘t1 & 0` and save result in t0
```

Load/Store Operations

lb	Load Byte	I	0000011	0x0		rd = M[rs1+imm][0:7]	
lh	Load Half	I	0000011	0x1		rd = M[rs1+imm][0:15]	
lw	Load Word	I	0000011	0x2		rd = M[rs1+imm][0:31]	
lbu	Load Byte (U)	I	0000011	0x4		rd = M[rs1+imm][0:7]	zero-extends
lhu	Load Half (U)	I	0000011	0x5		rd = M[rs1+imm][0:15]	zero-extends
sb	Store Byte	S	0100011	0x0		M[rs1+imm][0:7] = rs2[0:7]	
sh	Store Half	S	0100011	0x1		M[rs1+imm][0:15] = rs2[0:15]	
sw	Store Word	S	0100011	0x2		M[rs1+imm][0:31] = rs2[0:31]	

```
lw    t0, t1, 12        # Load word (4 bytes) from address ‘t1 + 12’ into register t0.
sw    t0, t1, 12        # Store word-sized register (4 bytes) t0 to address ‘t1 + 12`
```

Branch Operations

beq	Branch ==	B	1100011	0x0		if(rs1 == rs2) PC += imm	
bne	Branch !=	B	1100011	0x1		if(rs1 != rs2) PC += imm	
blt	Branch <	B	1100011	0x4		if(rs1 < rs2) PC += imm	
bge	Branch ≥	B	1100011	0x5		if(rs1 >= rs2) PC += imm	
bltu	Branch < (U)	B	1100011	0x6		if(rs1 < rs2) PC += imm	zero-extends
bgeu	Branch ≥ (U)	B	1100011	0x7		if(rs1 >= rs2) PC += imm	zero-extends

```
beq      t0, t1, loop_header    # If t0 and t1 are equal, branch to label
                                # ‘loop_header’

bltu     t0, t1, loop_header    # If t0 is less than t1 (interpreted as
                                # unsigned numbers) branch to label
                                # ‘loop_header’
```

Jumps, Load Upper Immediate, System Calls

jal	Jump And Link	J	1101111			rd = PC+4; PC += imm	
jalr	Jump And Link Reg	I	1100111	0x0		rd = PC+4; PC = rs1 + imm	
lui	Load Upper Imm	U	0110111			rd = imm << 12	
auipc	Add Upper Imm to PC	U	0010111			rd = PC + (imm << 12)	
ecall	Environment Call	I	1110011	0x0	imm=0x0	Transfer control to OS	
ebreak	Environment Break	I	1110011	0x0	imm=0x1	Transfer control to debugger	

An xDSL/MLIR dialect for RISC-V

Does not use operands and result values!

Instead: Attributes

```
add t0, t1, t2
```

```
  // riscv.add() ["rd" = !riscv.reg<t0>, "rs1" = !riscv.reg<t1>, "rs2" = !riscv.reg<t2>]
```

```
addi t0, t1, 0
```

```
  // riscv.addi() ["rd" = !riscv.reg<t0>, "rs1" = !riscv.reg<t1>, "immediate" = 0 : !i64]
```

```
arithmetic:
```

```
  // riscv.label() ["label" = !riscv.label<arithmetic>]
```