CS443: Compiler Construction

Lecture 21: Risc-V ISA

Stefan Muller

Based on material by Yan Garcia and Rujia Wang
You are here

Higher-order
Typed
Structured Data
Nested Expressions
Unlimited Variables

First-order
Typed
Structured Data
Nested Expressions
Unlimited Variables

First-order
Typed
Structured Data
Flat Expressions
Unlimited Variables

First-order
Untyped
No Structured Data
Flat Expressions
32 Hardware Registers

Optimization

Source Code
Lexing/Parsing
MiniCaml AST
Closure Conv./Lifting
Mini-C
IR Generation
LLVM
Register Allocation
LLVM
Instruction Selection
Risc-V
An ISA is the set of instructions a computer can execute

- The job of a CPU
  - Fetch an instruction from memory
  - Decode
  - Execute
  - Write results to memory
  - Repeat (basically) forever

```
add x3, x2, x0
```

```
01110001110110
```
There are many different ISAs with rich histories

Intel x86

Apple PowerPC

1991

2020 – M1/M2 (ARM)

We can make our own chips!

We can make our own chips!

https://www.flickr.com/people/mylerdude/

By Mike Deerkoski - https://www.flickr.com/photos/deerkoski/7178643521/in/photostream
There are many different ISAs with rich histories

Intel
- x86
- x86-64

AMD
- x86
- AMD64

1990
2000

Hey, no stealing our architecture!

...
RISC (Reduced Instruction Set Computer) idea: simpler, faster hardware

• Earlier philosophy (“CISC”):
  Want to do something new? Add an instruction!

• RISC: Cocke, Hennessy, Patterson (1980s)
RISC-V: A simple RISC Architecture, good for teaching

- Originally developed in 2010 at UC Berkeley for teaching
- Open-source
Assembly Language: Human-readable machine code

- Assembly language is tied to ISA
- (Roughly) 1-to-1 correspondence with ISA instructions
  - (Some assembly languages offer convenient mnemonics that expand to multiple instructions)
An instruction is an opcode and operands (registers)

add x3, x2, x0

• Operands can only be registers and sometimes constants ("immediates")

• Registers: Limited number of single-word storage locations in hardware
Registers in RISC-V

• (Also some floating point registers we won’t talk about)

<table>
<thead>
<tr>
<th>Register</th>
<th>ABI Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>x0</td>
<td>zero</td>
</tr>
<tr>
<td>x1</td>
<td>ra</td>
</tr>
<tr>
<td>x2</td>
<td>sp</td>
</tr>
<tr>
<td>x3</td>
<td>gp</td>
</tr>
<tr>
<td>x4</td>
<td>tp</td>
</tr>
<tr>
<td>x5-7</td>
<td>t0-2</td>
</tr>
<tr>
<td>x8</td>
<td>s0/fp</td>
</tr>
<tr>
<td>x9</td>
<td>s1</td>
</tr>
<tr>
<td>x10-11</td>
<td>a0-1</td>
</tr>
<tr>
<td>x12-17</td>
<td>a2-7</td>
</tr>
<tr>
<td>x18-27</td>
<td>s2-11</td>
</tr>
<tr>
<td>x28-31</td>
<td>t3-t6</td>
</tr>
</tbody>
</table>
Before we dive into RISC-V: A quick recap on data representation

- Bit (**binary digit**): 0 or 1
- “Nibble”: 4 bits (1 hex digit 0x0-0xF)
- Byte: 8 bits
  - 2 hex digits: 0x00-0xFF
- Word: “Natural” size of data operated on by a computer
  - 32-bit ISA: 32 bits (4 bytes)
  - Width of registers
Integers in binary/hex

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>$2^3$</td>
<td>$2^2$</td>
<td>$2^1$</td>
<td>$2^0$</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>a</td>
<td>1010</td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>10</td>
<td>10000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>20</td>
<td>100000</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

“Most significant”  “Least significant”
Review: Endianness

• Store data one byte at a time
  • Order of bits in a byte doesn’t change!

• So do we store the most significant byte at the lowest memory address (the way we’d write it left-to-right) or the highest?
  • Lowest: “Big-endian” (e.g., IBM System/360)
  • Highest: “Little-endian” (e.g., x86, RISC-V)
Little-endian

- Oxdeadbeef

| ef | be | ad | de |
Two’s complement signed integers

• A 1 in MSB (Most significant bit) subtracts $2^{31}$ (instead of adding it)
• 100000.... = $-2^{31}$
• 011111.... = $2^{31}-1$ (highest positive # representable)
• 111111.... = -1

• Can just add two’s complement #s without casing on sign!
Two’s complement means two ways to extend integers to the left

1010101

- If signed int: want to sign-extend (extend with MSB)
  - LLVM: sext
  - 101 as 3-bit int = -3 = 11101 as 5-bit int

- If unsigned: want to zero-extend (extend with 0s)
Assembly operands, registers are untyped

- Value is whatever we interpret it as – (signed/unsigned) int/char/bool, etc.

| x1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| x2 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| x3 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

add x3, x2, x1

Overflow:
char: Yes. unsigned int: No. signed int: Yes.
Still want types? Never fear

TALx86: A Realistic Typed Assembly Language*

Greg Morrisett  Karl Crary†  Neal Glew  Dan Grossman  Richard Samuels
Frederick Smith  David Walker  Stephanie Weirich  Steve Zdancewic
Cornell University

1999
Q: Why not make a bigger processor with more registers?
RISC-V Instructions are 32 bits

- **6 types of instructions:**

<table>
<thead>
<tr>
<th>R-type</th>
<th>I-type</th>
<th>S-type</th>
<th>B-type</th>
<th>U-type</th>
<th>J-type</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>funct7</td>
<td>rs2</td>
<td>rs1</td>
<td>funct3</td>
<td>rd</td>
<td>opcode</td>
</tr>
<tr>
<td>imm[11:0]</td>
<td>rs1</td>
<td>funct3</td>
<td>rd</td>
<td>opcode</td>
<td></td>
</tr>
<tr>
<td>imm[31:12]</td>
<td>rd</td>
<td>opcode</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
R-type instruction: Destination, two register operands

<table>
<thead>
<tr>
<th>Risc-V</th>
<th>LLVM</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>add x1, x2, x3</td>
<td>%x1 = add i32 %x2 %x3</td>
<td>x1 = x2 + x3</td>
</tr>
<tr>
<td>sub x3, x4, x5</td>
<td>%x3 = sub i32 %x4 %x5</td>
<td>x3 = x4 - x5</td>
</tr>
</tbody>
</table>

Also: xor, or, and, mul, div
divu (div unsigned)
sll (shift left logical)
srl (shift right logical) – fill left with 0s
sra (shift right arithmetic) – fill left with sign bit
slt (set rd to 1 iff rs1 < rs2)
x0 is always 0, writes are ignored

• Why would you want to read from x0?
  • mv rd, rs = add rd rs x0

• Why would you want to write to x0?
  • nop = add x0 x0 x0
  • (There are other ways to write a no-op instruction, but this is the conventional one)
l-type instructions: Destination, register, immediate

<table>
<thead>
<tr>
<th>Risc-V</th>
<th>LLVM</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>addi x1, x2, n</td>
<td>%x1 = add i32 %x2 n</td>
<td>x1 = x2 + n</td>
</tr>
<tr>
<td>sub x3, x4, n</td>
<td>%x3 = sub i32 %x4 n</td>
<td>x3 = x4 – n</td>
</tr>
</tbody>
</table>

Also: xori, ori, andi, (NO muli, divi)

- slti
- slli (shift left logical)
- srli (shift right logical) – fill left with 0s
- srai (shift right arithmetic) – fill left with sign bit
Example

\%x = mul i32 \%y 2

- add x1, x2, x2
- slli x1, x2, 2
- addi x1, x0, 2
  mul x1, x2, x1

x1 <- x
x2 <- y
Remember: You only get 12 bits for immediate (not very big)

- In RISC-V immediates are "sign extended"
  - So the upper bits are the same as the largest bit
  - Remember sign extended 2’s complement..
- So for a 12b immediate...
  - Bits 31:12 get the same value as Bit 11

```
<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>25 24</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>15 14</th>
<th>12 11</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>funct7</td>
<td>rs2</td>
<td>rs1</td>
<td>funct3</td>
<td>rd</td>
<td>opcode</td>
<td>R-type</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>imm[11:0]</td>
<td>rs1</td>
<td>funct3</td>
<td>rd</td>
<td>opcode</td>
<td>I-type</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```
If you need big immediates, need 2 insts

Risc-V
lui x1, n

C
x1 = n << 12 (x1 = n * 4096)

%x = add i32 %y, 5000

4096 = 1 0011 1000 1000

lui x1, 1
addi x1, x1, 904
add x1, x1, x2
Control flow in LLVM: similar to LLVM, but less structured

Assembly:

```
loopforever:
  add x0, x0, x0
  j loopforever
```

After assembling/linking:

```
add x0, x0, x0
j -4
```

**Offset:** Position independent
\textbf{j isn’t actually an instruction}

- It’s a “pseudoinstruction” that gets expanded into other instructions by the assembler (like \texttt{mv}, \texttt{nop})

- We’ll see more about this next week
B-type instructions (Conditional branches): 2 registers and a label/offset

Risc-V        LLVM                                      C
beq x1, x2, addr  %x3 = icmp eq i32 %x1 %x2         if (x1 == x2) goto addr
                 br i1 %x3, label addr, ???

Also: bne, blt, bge, (bltu, bgeu)

NO ble, bgt
Example

```assembly
%x1 = icmp lt i32 %x2, %x3
br i1 %x1, label ltrue, label lfalse

blt x2, x3, ltrue
j lfalse

slt x1, x2, x3
bne x1, x0, ltrue
j lfalse
```

Unlike LLVM, control “falls through” to next instruction
Example

%\texttt{x1} = \texttt{icmp le i32 \texttt{x2}, \texttt{x3}}
\texttt{br i1 \texttt{x1}, label ltrue, label lfalse}

\texttt{bge x3, x2, ltrue}
\texttt{j lfalse}
Announcements

• Project 5 Deadline Extended to Monday (11/14)
• OH tomorrow, 2-3, NOT today
  • May be on Zoom, I’ll let you know in the morning

• Schedule for rest of semester:
  • Mon, 11/14: Project 5 Due, Project 6 Out
  • 11/17, 11/22: Memory Management
  • Thur, 11/24: Thanksgiving, no class
  • 11/29, 12/1: TBA Lectures – suggest topics!
  • Fri, 12/2: Project 6 Due
  • **Tue 12/6, 10:30am, SB 113** – Final exam
Example

Assuming assignments below, compile if block

\[ f \rightarrow x_{10} \quad g \rightarrow x_{11} \quad h \rightarrow x_{12} \]
\[ i \rightarrow x_{13} \quad j \rightarrow x_{14} \]

\[
\text{if (i == j)} \quad \text{bne } x_{13}, x_{14}, \text{done} \\
f = g + h; \quad \text{add } x_{10}, x_{11}, x_{12} \\
\text{done:}
\]
Unconditional jump instructions: jal, jalr

• jal rd, imm
  • Jump to label (or by offset)
  • Set rd = PC + 4 (next instruction after jal)

• jalr rd, rs, imm
  • Jump to address in rs + imm
  • Set rd = PC + 4 (next instruction after jal)

• j imm = jal x0, imm
Loading from and storing to memory

Fast but limited place To hold values

Much larger place To hold values, but slower than registers!
Memory is addressed in bytes

• (But access memory a word at a time, so in practice, will only access memory at multiples of 4 bytes)

• Generally: data $\geq 1$ word must be aligned to addresses that are multiples of 4
lw loads from memory to register

lw rd, imm(rs)

Load word at rs + imm into rd
lw loads from memory to register

C code
```c
int A[100];
g = h + A[3];
```

Using Load Word (lw) in RISC-V:
- `lw x10,12(x13) # Reg x10 gets A[3]`
- `add x11,x12,x10 # g = h + A[3]`
- Assume: x13– base register (pointer to A[0]) Note: 12– offset in bytes
- Offset must be a constant known at *assembly time*
sw transfers from register to memory

C

int A[100]

RISC-V

lw x10, 12(x13)
add x10, x12, x10
sw x10, 40(x13)

Note:
• x13 – base register (pointer)
• 12, 40 – offsets in bytes
• x13 + 12 and x13 + 40 must be multiples of 4 to maintain alignment
Example

addi x11, x0, 0xfeed
addi x12, x0, 0xbeef
addi x6, x5, 4
sw x11, 0(x5)
sw x12, 4(x5)
lw x12, 0(x6)

• What’s the value in x12? Answer: 0xbeef
Example

\[
\begin{align*}
\text{addi} & \ x11, x0, 0xfeed \\
\text{addi} & \ x12, x0, 0xbeef \\
\text{addi} & \ x6, x5, 1 \\
\text{sw} & \ x11, 0(x5) \\
\text{sw} & \ x12, 4(x5) \\
\text{lw} & \ x12, 0(x6)
\end{align*}
\]

• What’s the value in \(x12\)?  
  Answer: Undefined
Memory layout in RISC-V

\[ \text{sp} = \text{bfff fff0}_{\text{hex}} \]

Stack Frame

Dynamic data

Static data

Text

Reserved

(heap)
A stack frame is where we store spilled locals, plus anything *alloca*’d

- **fp (x8)**
  - "Frame pointer"
- **sp (x2)**
  - "Stack pointer"

Q: Do we need the frame pointer if there's no dynamic stack allocation?

- Spilled var 1
- Spilled var 2
- ...
- Spilled var N

Dynamically allocated space