Recitation 2: RISC-V

CENG 444 - Language Processors
Fall 2022

Advice for Project II

- **Start Early!**
    - **Implement assignment statements, binary expressions, if-else-while *<u>this weekend!</u>*** You should already know how to do it.
- Don’t go into read creep.
    - Put down the dragon book when the due date is near. You will always have something you will have to come up with an ad-hoc solution.
- Done is better than perfect.
    - Your language should compile to <u>correct</u> assembly *first.* Then you can focus on compiling to efficient assembly.
    - Don’t prepare for optional features before you implement the required ones!

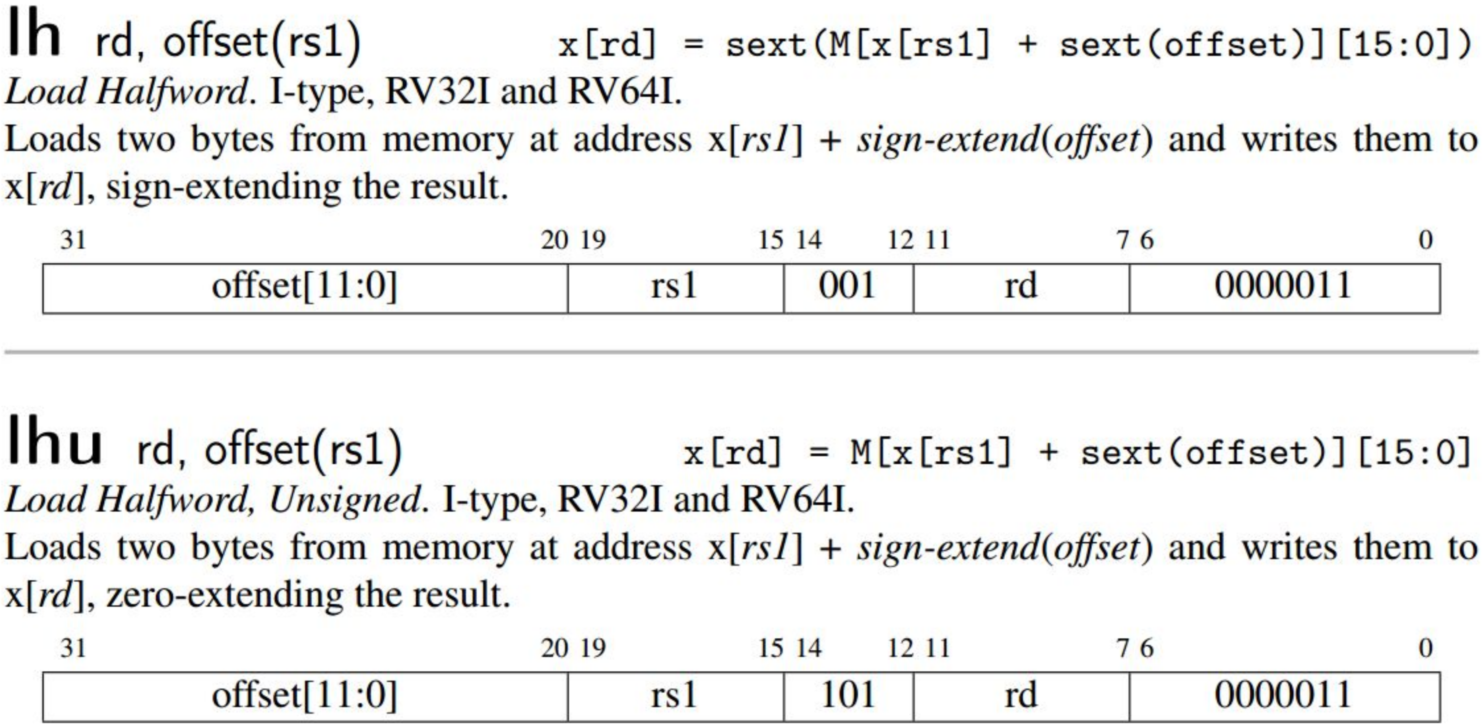
Why RISC-V

- Open source. No need to pay money to build your hardware implementation.
- Modular. A base frozen ISA + optional extensions
    - An incremental ISA, x86-64 has to adopt all archaic instructions of 80x86.
    - For RISC-V, you start with RV64I base, which will never change, and add other extensions (M,A,D,V etc.) according to your hardware support.
    - Or maybe custom extensions for your specific hardware
- V Extension has vector architecture instead of SIMD instructions
    - No need to introduce new instructions when vector registers grow in size/number.

RV64I - Introduction

- 64-bit addressing space & data.
- Unlike x86-64, instructions are fixed size, 32-bits.
- Unlike x86-64, instructions operate on three registers.
    - Easier to convert from TAC :)
- Unlike x86-64, no operations between memory-register except load/store operations.
- 32 64-bit registers x0-x31. One of them (`zero` register) is hardwired to 0
- Immediates within instructions are either 20 bits or 12 bits. They are always sign extended (removes the need to put `subi` instruction etc.)

RV64I - load/store



lh rd, offset(rs1) `x[rd] = sext(M[x[rs1] + sext(offset)][15:0])`

Load Halfword. I-type, RV32I and RV64I.

Loads two bytes from memory at address x[*rs1*] + *sign-extend*(*offset*) and writes them to x[*rd*], sign-extending the result.

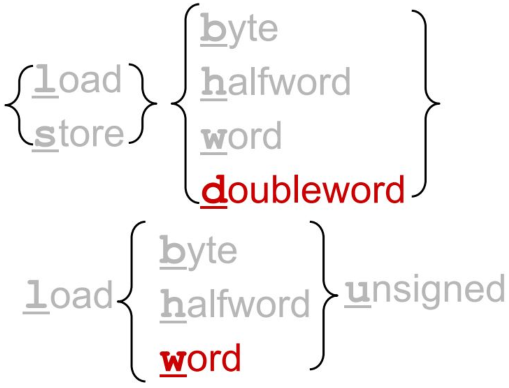
31 20	19 15	14 12	11 7	6 0
offset[11:0]	rs1	001	rd	0000011

lhu rd, offset(rs1) `x[rd] = M[x[rs1] + sext(offset)][15:0]`

Load Halfword, Unsigned. I-type, RV32I and RV64I.

Loads two bytes from memory at address x[*rs1*] + *sign-extend*(*offset*) and writes them to x[*rd*], zero-extending the result.

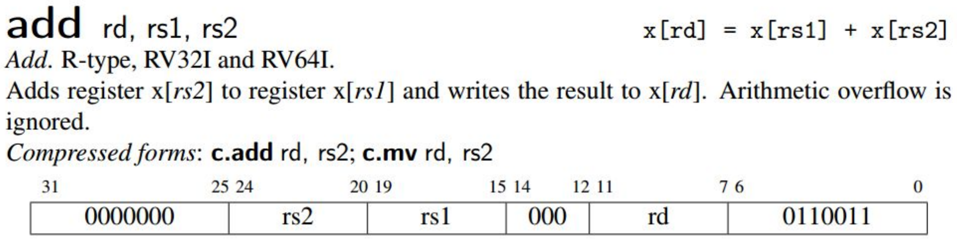
31 20	19 15	14 12	11 7	6 0
offset[11:0]	rs1	101	rd	0000011



Any combination here is possible:
Ld -> loads a doubleword (64-bit)
lbu -> loads a byte, zero extends the result

Halfword: 16-bits, word: 32-bits, doubleword: 64-bits.

RV64I - arithmetic



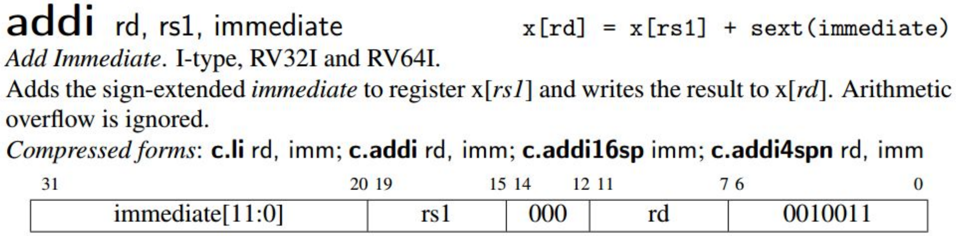
add rd, rs1, rs2 `x[rd] = x[rs1] + x[rs2]`

Add. R-type, RV32I and RV64I.

Adds register x[*rs2*] to register x[*rs1*] and writes the result to x[*rd*]. Arithmetic overflow is ignored.

Compressed forms: **c.add** rd, rs2; **c.mv** rd, rs2

31 25	24 20	19 15	14 12	11 7	6 0
0000000	rs2	rs1	000	rd	0110011



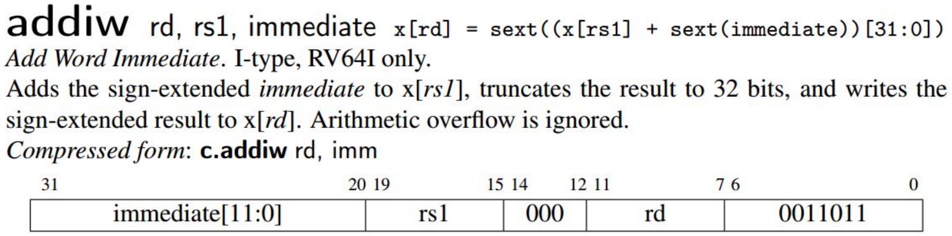
addi rd, rs1, immediate `x[rd] = x[rs1] + sext(immediate)`

Add Immediate. I-type, RV32I and RV64I.

Adds the sign-extended *immediate* to register x[*rs1*] and writes the result to x[*rd*]. Arithmetic overflow is ignored.

Compressed forms: **c.li** rd, imm; **c.addi** rd, imm; **c.addi16sp** imm; **c.addi4spn** rd, imm

31 20	19 15	14 12	11 7	6 0
immediate[11:0]	rs1	000	rd	0010011



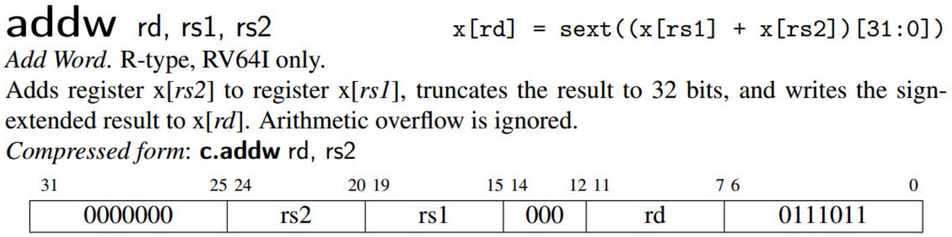
addiw rd, rs1, immediate `x[rd] = sext((x[rs1] + sext(immediate))[31:0])`

Add Word Immediate. I-type, RV64I only.

Adds the sign-extended *immediate* to x[*rs1*], truncates the result to 32 bits, and writes the sign-extended result to x[*rd*]. Arithmetic overflow is ignored.

Compressed form: **c.addiw** rd, imm

31 20	19 15	14 12	11 7	6 0
immediate[11:0]	rs1	000	rd	0011011



addw rd, rs1, rs2 `x[rd] = sext((x[rs1] + x[rs2])[31:0])`

Add Word. R-type, RV64I only.

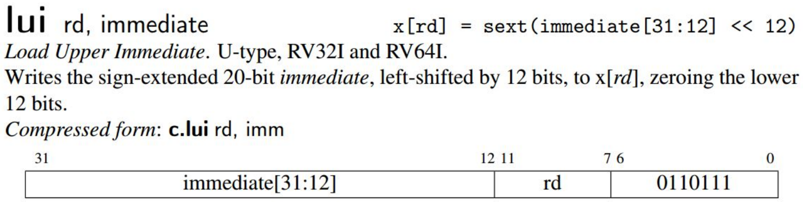
Adds register x[*rs2*] to register x[*rs1*], truncates the result to 32 bits, and writes the sign-extended result to x[*rd*]. Arithmetic overflow is ignored.

Compressed form: **c.addw** rd, rs2

31 25	24 20	19 15	14 12	11 7	6 0
0000000	rs2	rs1	000	rd	0111011

Do we need a register move instruction?

What about moving an (32-bit/64-bit) immediate?



lui rd, immediate `x[rd] = sext(immediate[31:12] << 12)`

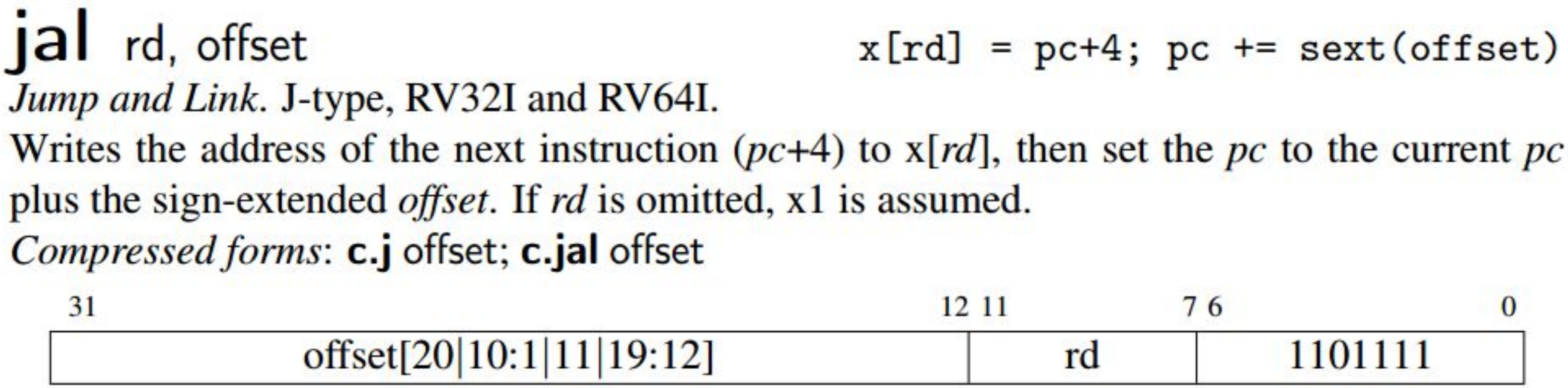
Load Upper Immediate. U-type, RV32I and RV64I.

Writes the sign-extended 20-bit *immediate*, left-shifted by 12 bits, to x[*rd*], zeroing the lower 12 bits.

Compressed form: **c.lui** rd, imm

31 12	11 7	6 0
immediate[31:12]	rd	0110111

RV64I - Unconditional Jumps



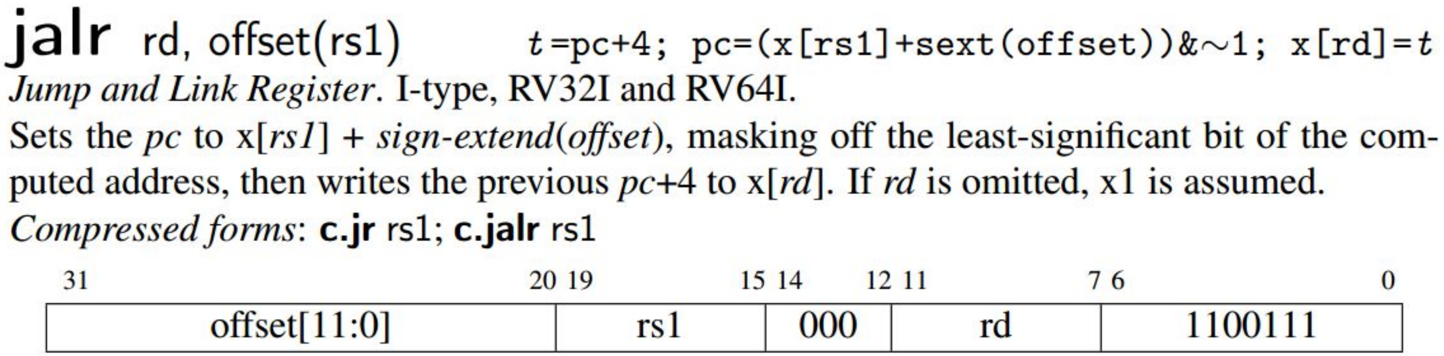
jal rd, offset x[rd] = pc+4; pc += sext(offset)

Jump and Link. J-type, RV32I and RV64I.

Writes the address of the next instruction (*pc*+4) to x[*rd*], then set the *pc* to the current *pc* plus the sign-extended *offset*. If *rd* is omitted, x1 is assumed.

Compressed forms: **c.j** offset; **c.jal** offset

31 12	11 7	6 0
offset[20|10:1|11|19:12]	rd	1101111



jalr rd, offset(rs1) *t* =pc+4; pc=(x[rs1]+sext(offset))&∼1; x[rd]=*t*

Jump and Link Register. I-type, RV32I and RV64I.

Sets the *pc* to x[*rs1*] + *sign-extend*(*offset*), masking off the least-significant bit of the computed address, then writes the previous *pc*+4 to x[*rd*]. If *rd* is omitted, x1 is assumed.

Compressed forms: **c.jr** rs1; **c.jalr** rs1

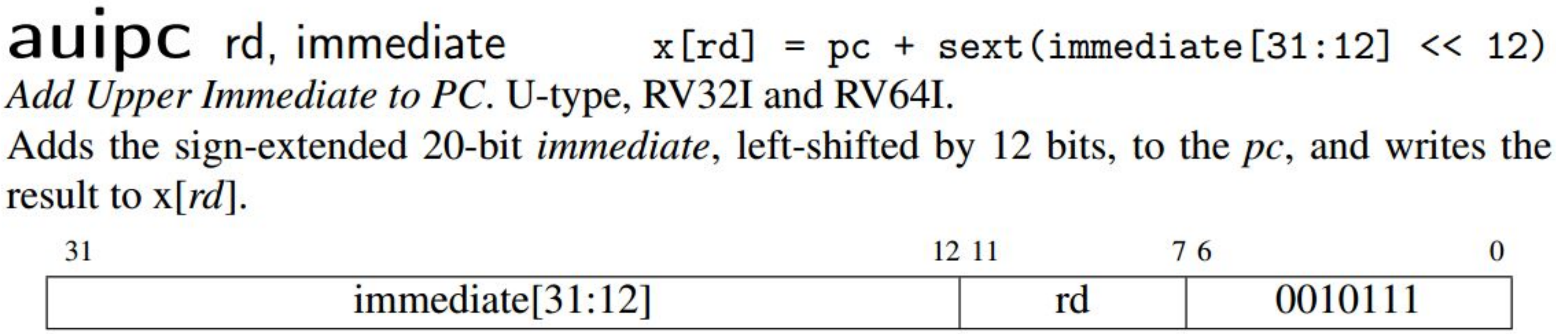
31 20	19 15	14 12	11 7	6 0
offset[11:0]	rs1	000	rd	1100111

Adds rs1+offset to pc and masks off last bit, and saves previous pc + 4 to rd (usually picked as register `ra`, the return register)

Why does it mask off the last bit? What is the point of storing pc+4?

How is the instruction after the jump stored in x86-64?

RV64I - Unconditional Jumps



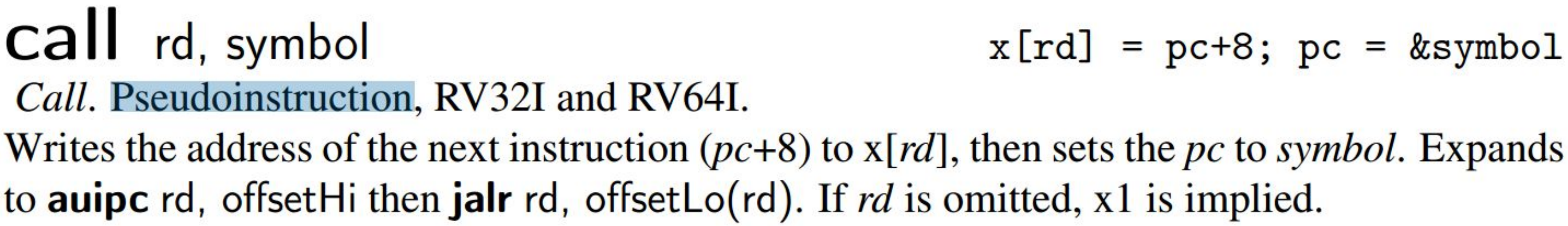
auipc rd, immediate `x[rd] = pc + sext(immediate[31:12] << 12)`

Add Upper Immediate to PC. U-type, RV32I and RV64I.

Adds the sign-extended 20-bit *immediate*, left-shifted by 12 bits, to the *pc*, and writes the result to x[*rd*].

31 – 12	11 – 7	6 – 0
immediate[31:12]	rd	0010111

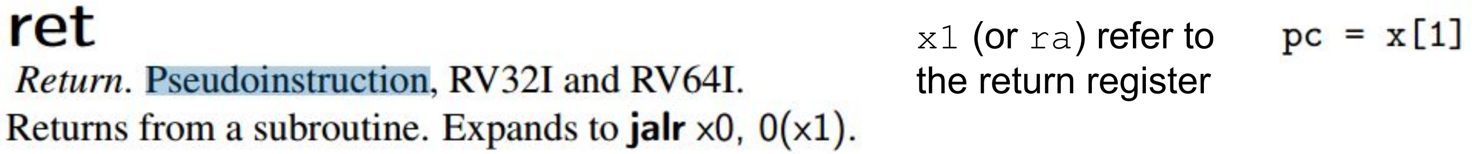
- Position-Independent Code: Jumps are encoded as current PC + offset, so the program would work in any address we load to.
- RISC-V is PIC friendly.
- How do we encode 32-bit offsets?



call rd, symbol `x[rd] = pc+8; pc = &symbol`

Call. Pseudoinstruction, RV32I and RV64I.

Writes the address of the next instruction (*pc*+8) to x[*rd*], then sets the *pc* to *symbol*. Expands to **auipc** rd, offsetHi then **jalr** rd, offsetLo(rd). If *rd* is omitted, x1 is implied.



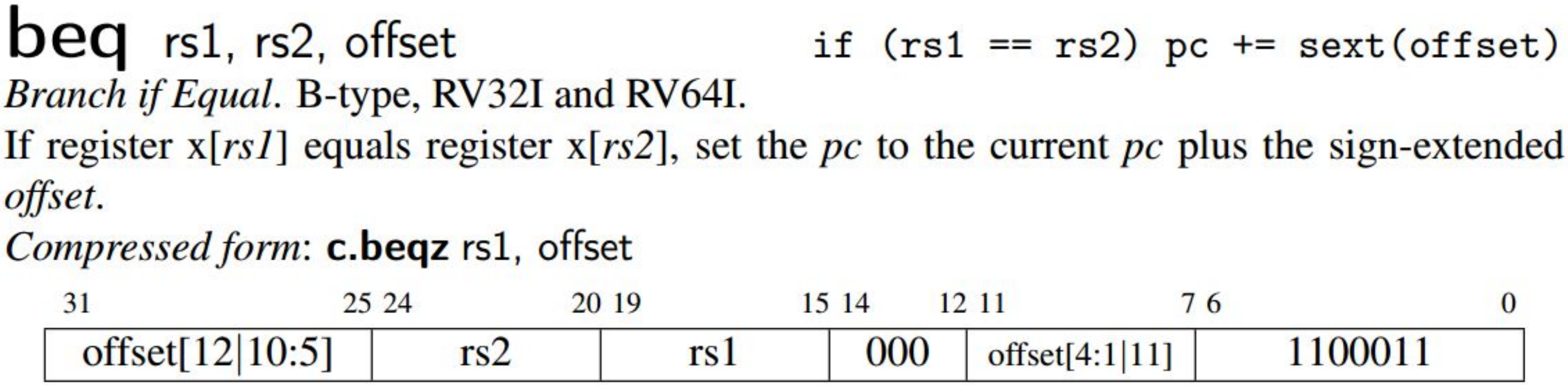
ret `pc = x[1]`

Return. Pseudoinstruction, RV32I and RV64I.

Returns from a subroutine. Expands to **jalr** x0, 0(x1).

`x1` (or `ra`) refer to the return register

RV64I - Conditional Jumps



beq rs1, rs2, offset `if (rs1 == rs2) pc += sext(offset)`

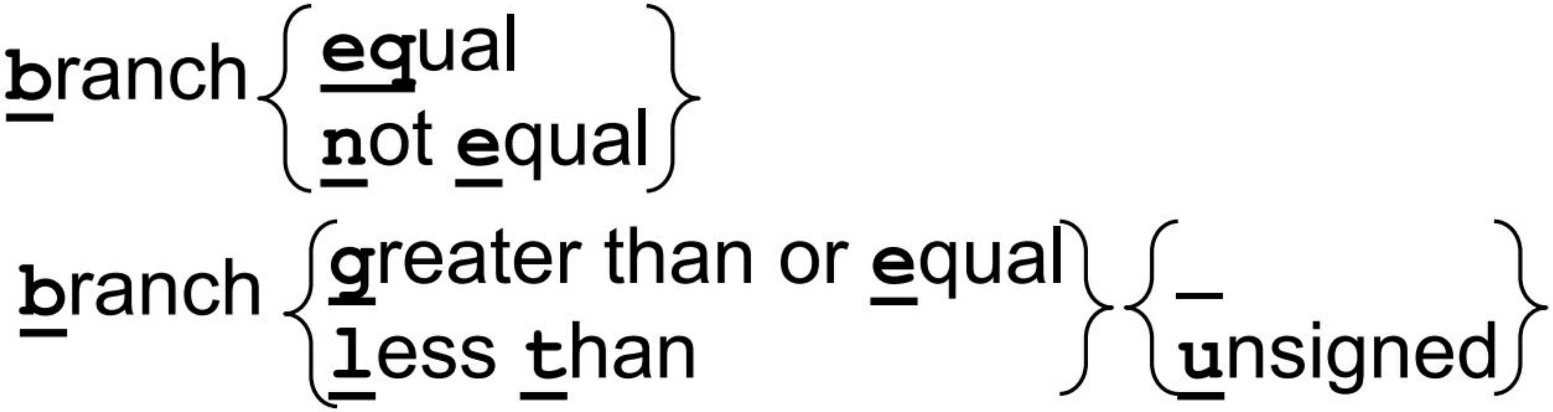
Branch if Equal. B-type, RV32I and RV64I.

If register x[*rs1*] equals register x[*rs2*], set the *pc* to the current *pc* plus the sign-extended *offset*.

Compressed form: **c.beqz** rs1, offset

31 25	24 20	19 15	14 12	11 7	6 0
offset[12|10:5]	rs2	rs1	000	offset[4:1|11]	1100011

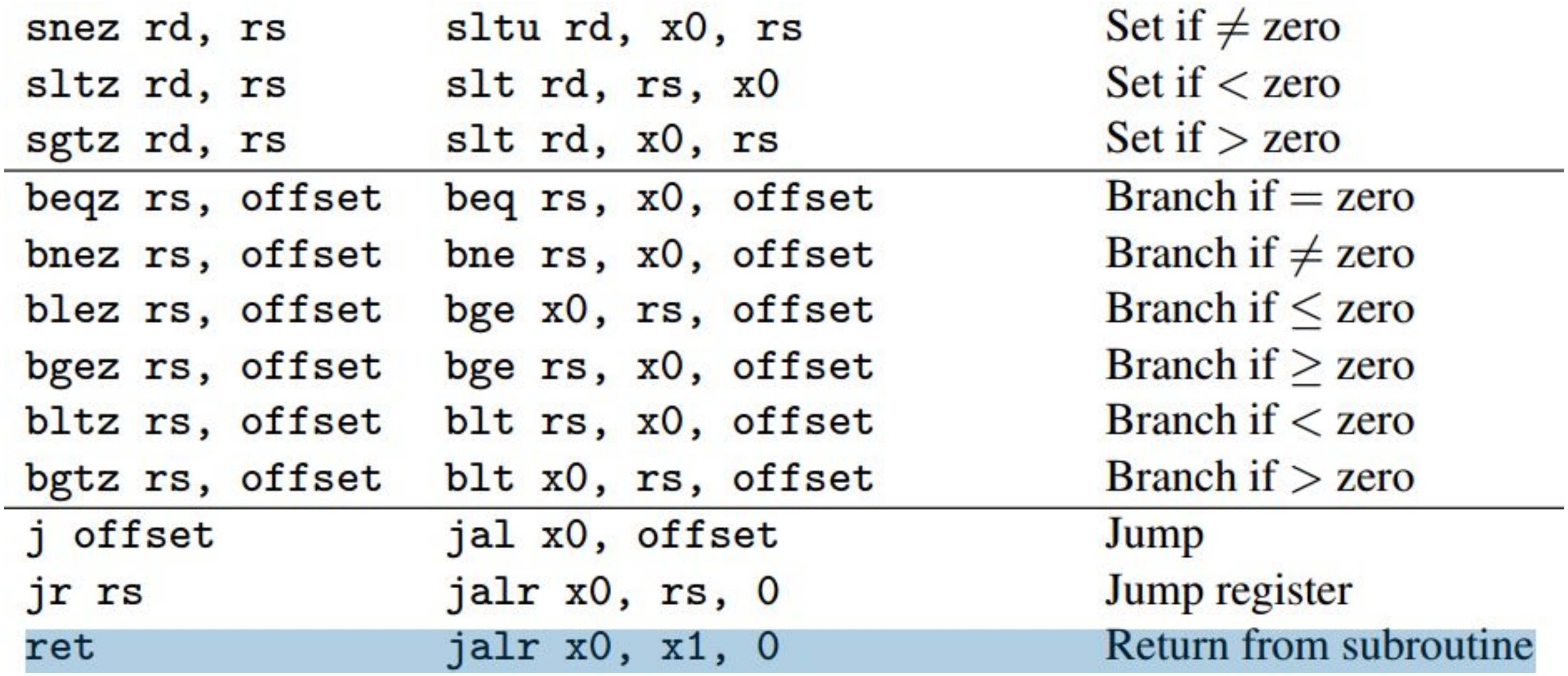
- If rs1 == rs2 goto label, else execute next instruction
- All of these are available



RV64I - Pseudoinstructions

- assembler converts these into a series of instructions / instructions with 0 immediate / instructions that use `zero` register etc.

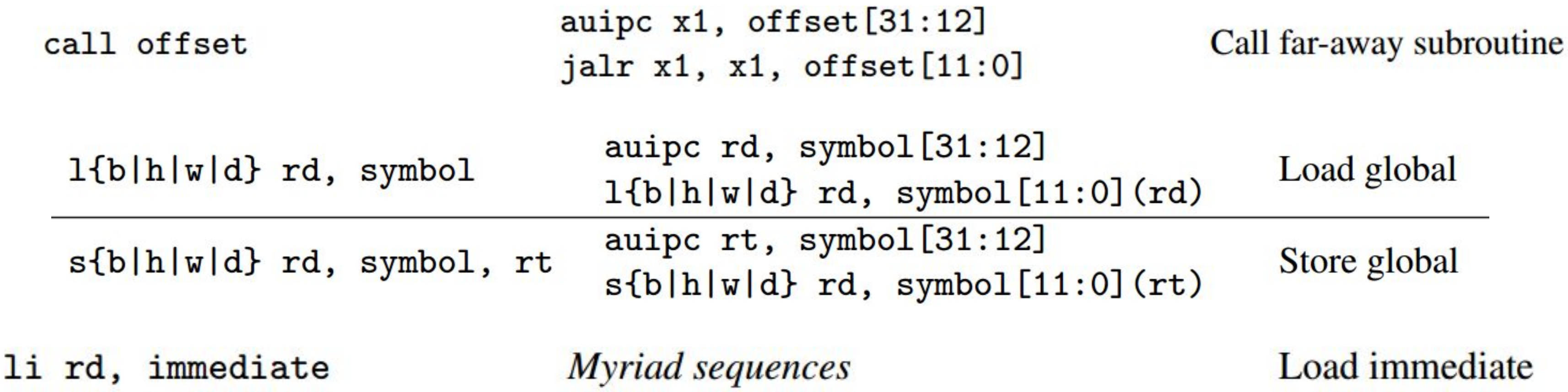
- **`call` and `ret` pseudoinstructions are important! They use the return register (`ra - i.e x1`).**



`snez rd, rs`	`sltu rd, x0, rs`	Set if ≠ zero
`sltz rd, rs`	`slt rd, rs, x0`	Set if < zero
`sgtz rd, rs`	`slt rd, x0, rs`	Set if > zero
`beqz rs, offset`	`beq rs, x0, offset`	Branch if = zero
`bnez rs, offset`	`bne rs, x0, offset`	Branch if ≠ zero
`blez rs, offset`	`bge x0, rs, offset`	Branch if ≤ zero
`bgez rs, offset`	`bge rs, x0, offset`	Branch if ≥ zero
`bltz rs, offset`	`blt rs, x0, offset`	Branch if < zero
`bgtz rs, offset`	`blt x0, rs, offset`	Branch if > zero
`j offset`	`jal x0, offset`	Jump
`jr rs`	`jalr x0, rs, 0`	Jump register
`ret`	`jalr x0, x1, 0`	Return from subroutine

RV64I - Pseudoinstructions

- assembler converts these into a series of instructions / instructions with 0 immediate / instructions that use `zero` register etc.
- **`call` and `ret` pseudoinstructions are important! They use the return register (`ra - i.e x1`).**



<table>
<tr><td><code>call offset</code></td><td><code>auipc x1, offset[31:12]</code>
<code>jalr x1, x1, offset[11:0]</code></td><td>Call far-away subroutine</td></tr>
<tr><td><code>l{b|h|w|d} rd, symbol</code></td><td><code>auipc rd, symbol[31:12]</code>
<code>l{b|h|w|d} rd, symbol[11:0](rd)</code></td><td>Load global</td></tr>
<tr><td><code>s{b|h|w|d} rd, symbol, rt</code></td><td><code>auipc rt, symbol[31:12]</code>
<code>s{b|h|w|d} rd, symbol[11:0](rt)</code></td><td>Store global</td></tr>
<tr><td><code>li rd, immediate</code></td><td><i>Myriad sequences</i></td><td>Load immediate</td></tr>
</table>

- I suggest you avoid the load immediate pseudoinstruction for values larger than 32 bits. Apparently it uses many shifts/luis/addis to compensate. Simply store that value as a global variable in .data section and then load.

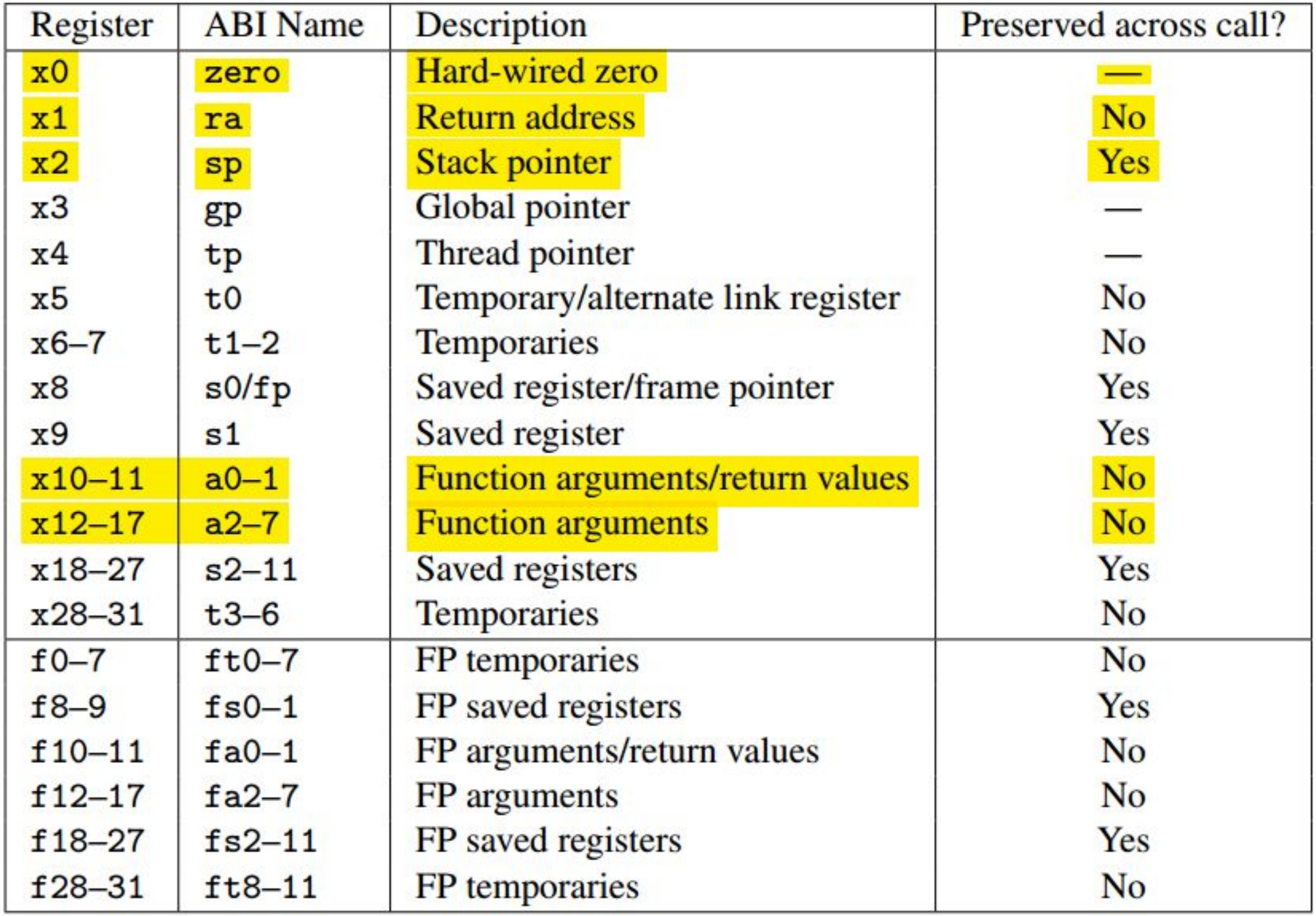
RV64I - Extensions and Other Details

- M extension adds multiplication/division
  - `mul` calculates lower 64-bits of multiplication
  - `mulh` calculates higher 64-bits of multiplication
  - `div` calculates division
  - `rem` calculates remainder
  - There are unsigned variants of these instructions (`mulhu` etc.).
- F/D extensions add float (32-bit) and double (64-bit) support
  - 32 double registers f0-f31 for arithmetic ops. Unlike zero register, f0 is not hardwired to 0.
  - Floating point to integer/ vice versa conversions happen between f and x registers

RV64I - Extensions and Other Details

- A extension adds atomic operations (for multiple hardware threads (harts)). C extension adds support for compressed 16-bit instructions for small immediate loads and common operations.
  - You will probably not need to know these. Assembler handles compression on its own, A extension is mostly reserved for kernel space.
- V extension introduces vector architecture and 32 vector registers.
- For user space, RV64I includes two special instructions, `ecall` for system calls and `ebreak` for debuggers to trap execution.

RV64I - Calling Conventions and ABI



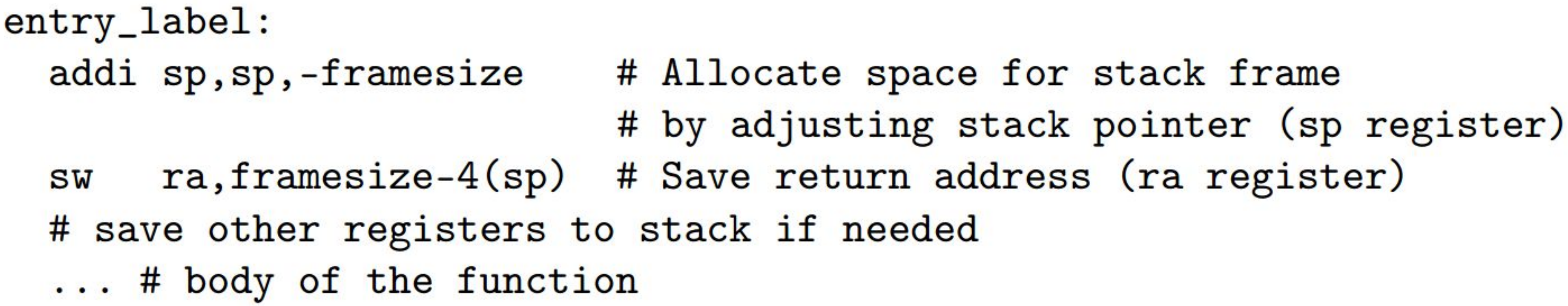
Register	ABI Name	Description	Preserved across call?
x0	zero	Hard-wired zero	—
x1	ra	Return address	No
x2	sp	Stack pointer	Yes
x3	gp	Global pointer	—
x4	tp	Thread pointer	—
x5	t0	Temporary/alternate link register	No
x6–7	t1–2	Temporaries	No
x8	s0/fp	Saved register/frame pointer	Yes
x9	s1	Saved register	Yes
x10–11	a0–1	Function arguments/return values	No
x12–17	a2–7	Function arguments	No
x18–27	s2–11	Saved registers	Yes
x28–31	t3–6	Temporaries	No
f0–7	ft0–7	FP temporaries	No
f8–9	fs0–1	FP saved registers	Yes
f10–11	fa0–1	FP arguments/return values	No
f12–17	fa2–7	FP arguments	No
f18–27	fs2–11	FP saved registers	Yes
f28–31	ft8–11	FP temporaries	No

RV64I - Calling Conventions and ABI

- Registers that start with t are **t**emporary registers. **They are not preserved across a call.**
- Registers that start with s are **s**aved registers. **They are preserved across a call.**
- Function **a**rguments start with a. Are they preserved across a call?
- Function return value is **`a0`**. **It is the first argument and the return value at the same time.** *Do not confuse it with the **r**eturn **a**ddress register **`ra`**!*
- **S**tack **p**ointer is **`sp`**.
- **`zero`** is 0.

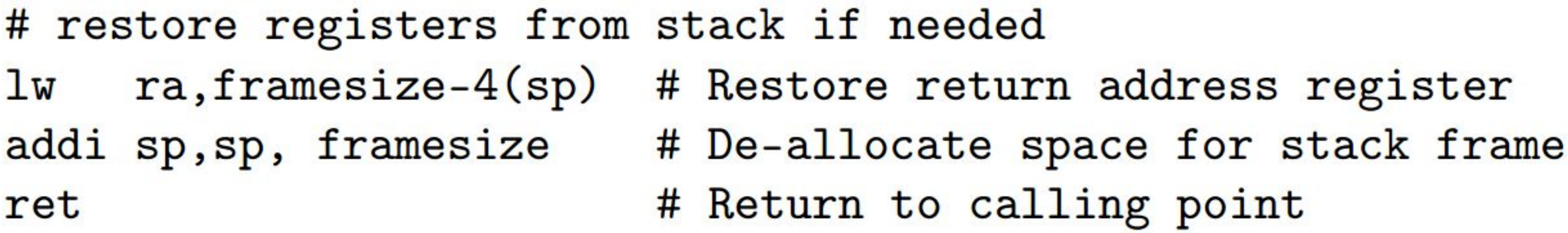
RV64I - Calling Conventions and ABI

- We may need to allocate space for `ra` within the stack in our function calls (Our very own x86-64 style `call`).



```
entry_label:
  addi sp,sp,-framesize    # Allocate space for stack frame
                           # by adjusting stack pointer (sp register)
  sw   ra,framesize-4(sp)  # Save return address (ra register)
  # save other registers to stack if needed
  ... # body of the function
```

- We then pop `ra` and then return, just like x86-64 `ret`.



```
# restore registers from stack if needed
lw   ra,framesize-4(sp)  # Restore return address register
addi sp,sp, framesize    # De-allocate space for stack frame
ret                      # Return to calling point
```

GNU Assembly Quickstart

- .text section: executable code resides here
- .data section: initialized global variables
- .bss section: uninitialized global variables
- Symbols (identifier that ends with “:”) before instructions or data
- Prepend data with .quad / .ascii etc. to mark datatype
- “.align 2” aligns following code to 4 bytes (since unaligned RISC-V instructions might cause traps we need to do this)
- .global <symbol> makes symbol visible to other files for linking
- If symbol name starts with “.L” it is local, objdumping later will not reveal this symbol. Use it for local branches.

GNU Assembly Quickstart

- **If compiled without C libraries, your assembly code should have a global “_start” symbol.** The OS will start your program from this location. **You need to make a syscall to terminate your program if you run your program “naked”** (For details, see Resources).
- **If compiled with C libraries, your assembly code should have a global “main” symbol.** gcc will generate a _start that sets signal handlers and other essential stuff, and then calls main. **Here, act as if you are in a function call, and simply return your termination code within `a0`.**
- **Now let’s write some RISC-V assembly!**

Demo in Progress…

If you could not attend to the recit, here is what happened:

- We first write a program that reads a global variable and returns it as termination code. Since the termination code is not easily visible from qemu, we run gdb with qemu and check the termination code from there. (hello_1.s)
- We than read multiple variables with indirect addressing from the same global variable, this time an array. Then we do arithmetic ops with the values and return it as the termination code. (hello_2.s)
- We add some branching to our code. (hello_3.s)
- We call puts from <stdio.h> in our assembly and print a fixed string, encoded as a global variable (hello_4.s)
- Within assembly, we call our custom c functions to read variables from stdin to do arithmetic ops. Then we print these variables to stdout (hello_5.s)

RV64I - V Extension Overview

- **Main advantage: Change element size & grouping of vector registers on the fly.** If vector register size is different in one architecture, this does not require adding new instructions or recompiling for a new target.
- Contrary to this, x86 introduced new instructions every time vector registers changed.
- **32 vx registers, supports strided, indexed, segmented adressing for vector registers. Supports masked operations.**
- **Similar to other parts of RV64I, arithmetic ops only permitted on vreg-vreg / vreg-xreg / vreg-freg.**
- Due to time restrictions, we won't be able to cover everything, but hopefully make it easier for to understand the V extension specification if you decide to employ an instruction with the terminology here.

RV64I - V Extension Terminology

- **VLEN:** Maximum number of bits a single vector register can hold.
- **ELEN:** Maximum number of bits an element can have in a vector register.
  - VLEN and ELEN are fixed for a particular hardware.
- **SEW:** *Selected Element Width.* A value we pick for subsequent vector operations, representing the number of bits in an element for this vector operation. If VLEN=128 and SEW=32, we can process 4 elements in a single vector register. If we pick SEW=64, we can process 2 elements in a single vector register.
- **LMUL:** *Vector Length Multiplier.* Represents the grouping of vector registers for subsequent vector operations. If LMUL = 2, v0-v1, v2-v3 … are grouped. Referencing odd numbered instructions when LMUL = 2 is reserved to hardware.

RV64I - V Extension Terminology

- **EEW:** *Effective Element Width.* Some instructions can specify element widths that override SEW, or they might require multiple element width specifications (narrowing/widening ops). These are regarded as EEW.
- **EMUL:** *Effective LMUL.* The grouping of registers implicitly grow/shrink with different EEWs within instructions, i.e. **EMUL/EEW = LMUL/SEW = VLMAX.** Note that referencing registers that does not start a grouping for an EMUL or implicit EMUL values that are not realizable for the hardware is reserved.

RV64I - V Extension Terminology

- We pick SEW and LMUL ourselves by modifying the **vtype** register, executing **`vsetvli`** instruction.
- SEW and LMUL in turn determine **VLMAX**, the maximum number of elements the vector register grouping can accept.
- Depending on the value of VLMAX and the number of elements we would like to process, the hardware supplies us *the number of items elements will be processed*, **vl,** *as the output of* ***`vsetvli`*** *instruction.*
- The vl value is saved to an x register we specify. We then, until we run out of elements to process, process vl number of elements, and subtract vl from the number of elements that needs to be processed.
- **Most vector operations fall into the following loop:**

RV64I - V Extension Stripmining Logic

```
arr[num_elements] inp;
arr[num_elements] out;
ind = 0;
while(num_elements > 0){
    rs = num_elements; //load the number of elements we want to
                      //process to an x register
    rd = vsetvli(rs, settings); // run vsetvli instruction with
                               // certain settings you want.
                              //vl is saved into register rd.
    load inp[ind:ind+rd] to some vector register;
    do some vector arithmetic, etc; //will process vl elements
    save from vector register to out[ind:ind+rd];
    num_element -= rd;
    ind += rd;
}
```

RV64I - vsetvli

```
e8     # SEW=8b
e16    # SEW=16b
e32    # SEW=32b
e64    # SEW=64b

mf8  # LMUL=1/8
mf4  # LMUL=1/4
mf2  # LMUL=1/2
m1   # LMUL=1, assumed if m setting absent
m2   # LMUL=2
m4   # LMUL=4
m8   # LMUL=8
```

`vl` saved to `t0`

We supply `num_elements` in `a0`

```
Examples:
    vsetvli t0, a0, e8           # SEW= 8, LMUL=1
    vsetvli t0, a0, e8, m2       # SEW= 8, LMUL=2
    vsetvli t0, a0, e32, mf2     # SEW=32, LMUL=1/2
```

RV64I - V Extension Stripmining Logic

- The vl vsetvli supplies is guaranteed to exhaust num_elements. Do not worry, it will not cause it to be less than zero.
- *Do not assume it is greedily set to VLMAX however! The logic is kept vague on purpose in the specification (It might be trying to evenly balance the last two loads instead of handling the tail only in the last load).*
- Where does EEW/EMUL come into play?
    - *If you must ask*, what basically happens is the same number of elements are processed (vl number of elements), but the element width is overriden by the EEW in the instruction. More/less space might be required in the vector register. Therefore the vector registers are grouped or fractioned for the instruction with the special EEW. (For example, let SEW be 32-bits, LMUL=1 (no grouping) and vl we received be 8 for a vector register of width 256-bits (VLEN=256). So vl = vlmax, the register is to be fully loaded. We then execute a vle64.v v0, instruction, which loads 64-bit elements to v0 (its EEW is 64-bits). However, since we have to load 8 elements, v0 itself won’t be enough. Therefore v0-v1 are grouped for this instruction. Elements are loaded to v1 as well.

RV64I - V Extension Memory Access Types

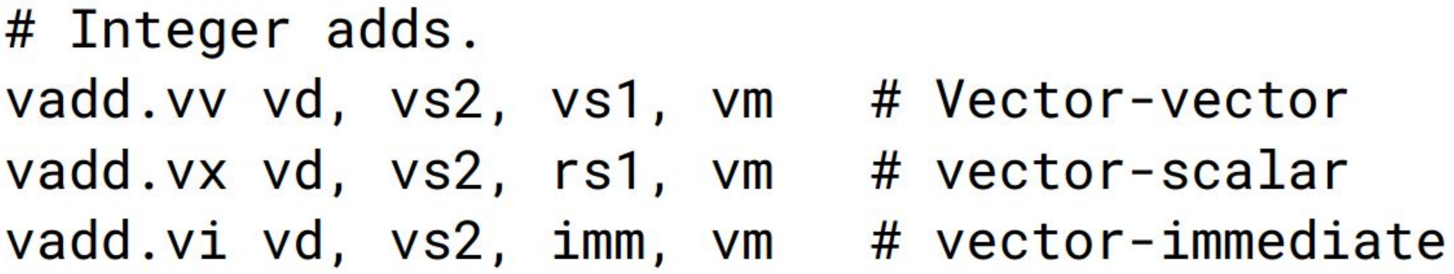
The V extension provides a dizzying variety of read/store types:

In Numpy notation:

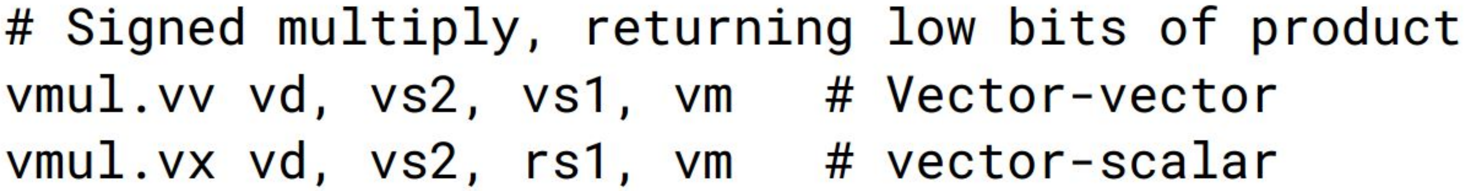
- Unit stride load/store: `vreg[0:n] = arr[0:n]`
- Strided load/store: `vreg[0:n] = arr[0:m:stride]`
- Indexed load/store: `vreg1[vreg2] = arr[vreg2]`, `vreg2` is a vector of byte offsets
- Segmented load/store: Let each element of arr be a tuple `(x,y)`. All `x` within `arr[0:n]` go to `vreg1[0:n]` and all `y` within `arr[0:n]` go to `vreg2[0:n]`
- For all operations, you can specify a bitmask within `v0`. The locations corresponding to the masked elements in vector registers are not modified.

Some Arithmetic Operations (There are lots of them)

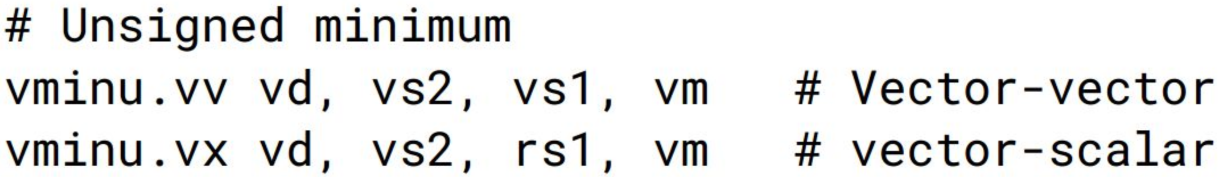
- Within the while loop mentioned in the Stripmining part, you can execute the following (vm is the mask register, you are free to not specify it):



```
# Integer adds.
vadd.vv vd, vs2, vs1, vm   # Vector-vector
vadd.vx vd, vs2, rs1, vm   # vector-scalar
vadd.vi vd, vs2, imm, vm   # vector-immediate
```



```
# Signed multiply, returning low bits of product
vmul.vv vd, vs2, vs1, vm   # Vector-vector
vmul.vx vd, vs2, rs1, vm   # vector-scalar
```



```
# Unsigned minimum
vminu.vv vd, vs2, vs1, vm   # Vector-vector
vminu.vx vd, vs2, rs1, vm   # vector-scalar
```

Demo in Progress…

- If you were not able to attend the recit, we analyzed a vector integer summation function we wrote in assembly (vector_long_sum.s). Note that the stripmining takes care of the tail of the array that is smaller than the vector.
- Fire up gdb and observe the value of `vl` saved in `t0`.

Resources

- RISC-V Reader. (Warning: V-Extension instruction names are deprecated in the book! But the stripmining logic is explained well)
- RISC-V Unprivileged Specifications
  - https://riscv.org/technical/specifications/
- RISC-V Assembly Programmer's Manual
  - https://github.com/riscv-non-isa/riscv-asm-manual/blob/master/riscv-asm.md
- RISC-V Hello World (without C libraries)
  - https://smist08.wordpress.com/2019/09/07/risc-v-assembly-language-hello-world/
- GNU Assembler Examples (written for x86-64 but you will be fine)
  - https://cs.lmu.edu/~ray/notes/gasexamples/
- RISC-V V Extension Specifications (hopefully these slides will help you understand it)
  - https://github.com/riscv/riscv-v-spec/releases/download/v1.0/riscv-v-spec-1.0.pdf