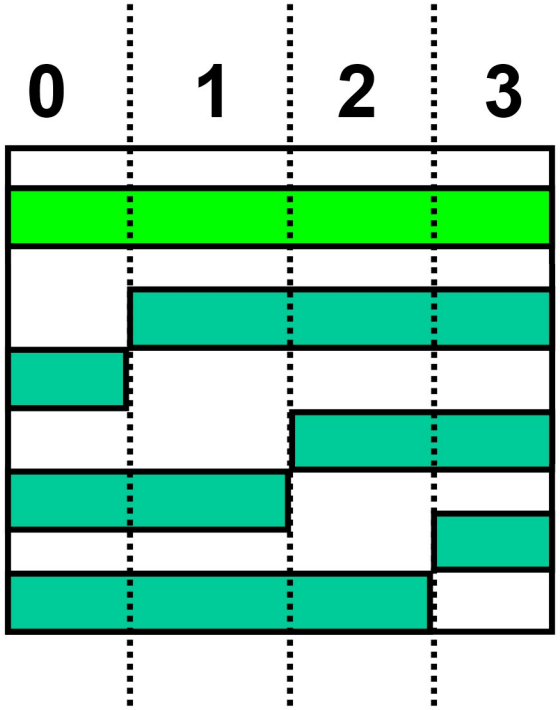
RISC-V Reference Card

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]	

Memory Alignment

- **RISC-V requires that all words start at byte addresses that are multiples of 4 bytes**



Last hex digit of memory address:

Word-aligned: low two bits are 0 (0,4,8,C)

Byte-aligned: anything

Halfword-aligned: low one bit is 0

Byte-aligned

- **Alignment: objects must fall on address that is multiple of their size**
  - For both lw and sw, the sum of the base address and the offset must be a multiple of 4 (to be word aligned)

RARS: Memory Alignment

- **.align <n>**
  - Ensure the next field is aligned to a multiple of 2^n
- **.align 2**
  - gives **word** alignment
  - Useful if you want to start a word array, but not sure if the previous items are of size in multiples of 4

Endian-ness

- **Storage order of bytes**
- **Little-endian (RISC-V)**
  - Store the little end of the number (least significant bits) first
- **Big-endian**
  - Store the big end of the number first
- **Example: 32766 (0x00007ffe)**

Memory address	Little-endian	Big-endian
0x10000003	0x00	0xfe
0x10000002	0x00	0x7f
0x10000001	0x7f	0x00
0x10000000	0xfe	0x00

Pointers vs. Values in Registers

- **Key Concept: A register can hold any 32-bit value.**
  - a (signed) `int`,
  - an `unsigned int`,
  - a pointer (memory address)
  - …
- **`add t2,t1,t0`**
  - `t0` and `t1` better contain **integers**
- **`lw t2,0(t0)`**
  - `t0` better contains a **pointer**

C code:
int d, int *c;
d = *c;

Compile assuming
c in s1
d in s2

RISC-V code:
lw s2, 0(s1)

RARS: Variables

- **All globals**
  - Compilers created them, not a human
- **C code:**

```
int a, b, c = 0;
```

- **RISC-V code:**

```
.data
a: .space 4    # allocate 4 bytes, uninitialized
b: .space 4    # allocate 4 bytes, uninitialized
c: .word 0     # 4 bytes, initialized to 0
```

RARS: Defining Arrays

- **C code:**

```
int ray[100];
char str[128];
int nums[ ] = {3, 0, 1, 2, 6, -2, 4, 7, 3, 7};
```

- **RISC-V code**

```
.data
ray: .space 400  # 100 4-byte locations = 400 bytes
str: .space 128  # 1 byte/char = 128 bytes
nums: .word 3, 0, 1, 2, 6, -2, 4, 7, 3, 7 # 10 words!

.text
…
   la a0, ray  # first address of ray (&ray[0])
   la a1, nums # first address of nums - not nums[0]!
               # memory operations used to get to values
```

RARS: Using Arrays

- **C Code:**

ray [0] = 24;

ray [4] = 48;

- **RISC-V Code:**

```
la a0, ray
addi t0, zero, 24
sw t0, 0(a0)
addi t0, zero, 48
sw t0, 16(a0)
```

- **C Code:**

str [0] = `a';

str [4] = `b';

- **RISC-V Code:**

```
la a0, str
addi t0, zero, `a'
sb t0, 0(a0)
addi t0, zero, `b'
sb t0, 4(a0)
```

Macros in RARS

- **Macros in RARS**
  - See RARS Help
  - Understand example-macros.s from Canvas and use it in your work

Live Coding Time

- **Download code from Canvas**
  - Work on mem.s
  - Work on data-transfer.s
  - Work on printbyte.s

Live Coding Time

- **Turn this C function into a RISC-V program**

```
int sum_pow2(int b, int c) {
  int pow2[] = {1, 2, 4, 8, 16, 32, 64, 128};
  int a;
  a = b + c;
  return pow2[a];
}
```

  - **read in b and c, output value rather than return result - for now**
  - **assume b+c<8 (just make sure you only give good values as input)**