Arithmetic Operations

§2.2 Operations of the Computer Hardware

- Add and subtract, three operands
  - Two sources and one destination

```
add a, b, c  // a gets b + c
```

- All arithmetic operations have this form
- *Design Principle 1:* Simplicity favours regularity
  - Regularity makes implementation simpler
  - Simplicity enables higher performance at lower cost

Immediate Operands

- Constant data specified in an instruction

```
addi x22, x22, 4
```

- Make the common case fast
  - Small constants are common
  - Immediate operand avoids a load instruction

Sign Extension

- Representing a number using more bits
  - Preserve the numeric value
- Replicate the sign bit to the left
  - c.f. unsigned values: extend with 0s
- Examples: 8-bit to 16-bit
  - +2: 0000 0010 => 0000 0000 0000 0010
  - –2: 1111 1110 => 1111 1111 1111 1110

- In RISC-V instruction set
  - `lb`: sign-extend loaded byte
  - `lbu`: zero-extend loaded byte

Memory Operand Example

- C code:

```
A[12] = h + A[8];
```

  - h in x21, base address of A in x22

- Compiled RISC-V code:
  - Index 8 requires offset of 64
    - 8 bytes per doubleword

```
ld      x9, 64(x22)
add     x9, x21, x9
sd      x9, 96(x22)
```

Memory Address = Imm. Const. + Register (12-bit Immediate sign-extended)

Uses: StructPointer+Offset; StackPointer+Offset; GlobPtr+Offs

Memory Operands

- Main memory used for composite data
  - Arrays, structures, dynamic data
- To apply arithmetic operations
  - Load values from memory into registers
  - Store result from register to memory
- Memory is byte addressed
  - Each address identifies an 8-bit byte
- RISC-V is Little Endian
  - Least-significant byte at least address of a word
  - *c.f.* Big Endian: most-significant byte at least address
- RISC-V does not require words to be aligned in memory
  - Unlike some other ISAs

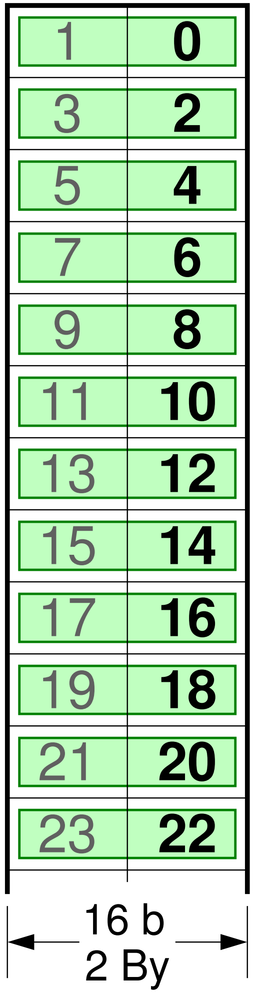
Big–Endian Machine:

	MS			*LS*
word 12:	byte12: 00000000	byte13: 00000000	byte14: 00000111	byte15: 11010011
word 16:	byte16: **k**	byte17: **a**	byte18: **t**	byte19: **e**
word 20:	byte20: **v**	byte21: **e**	byte22: **n**	byte23: **i**
word 24:	byte24: **s**	byte25: **\0**	byte26:	byte27:

Little–Endian Machine:

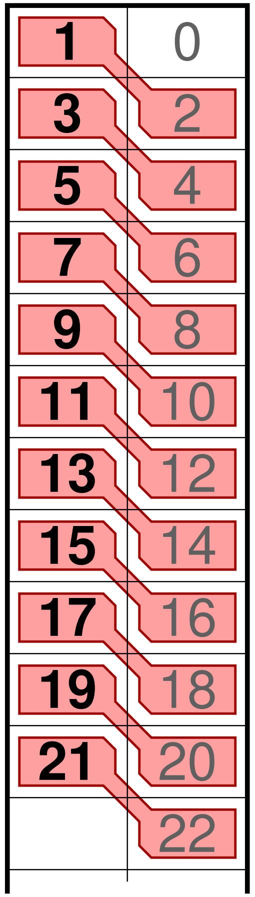
	MS			*LS*
word 12:	byte15: 00000000	byte14: 00000000	byte13: 00000111	byte12: 11010011
word 16:	byte19: **e**	byte18: **t**	byte17: **a**	byte16: **k**
word 20:	byte23: **i**	byte22: **n**	byte21: **e**	byte20: **v**
word 24:	byte27:	byte26:	byte25: **\0**	byte24: **s**

2–Byte "Half Words" Aligned on Addresses that are integer multiples of 2

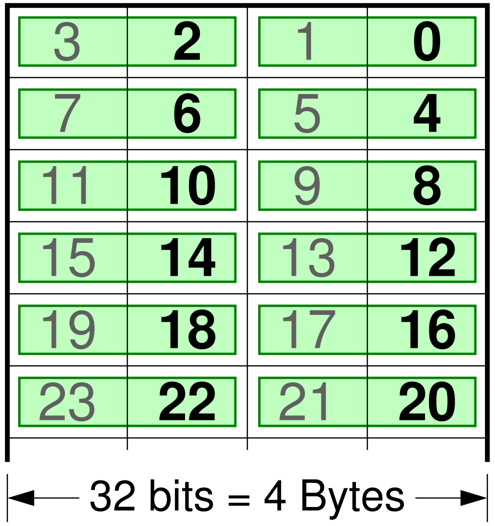


1	**0**
3	**2**
5	**4**
7	**6**
9	**8**
11	**10**
13	**12**
15	**14**
17	**16**
19	**18**
21	**20**
23	**22**

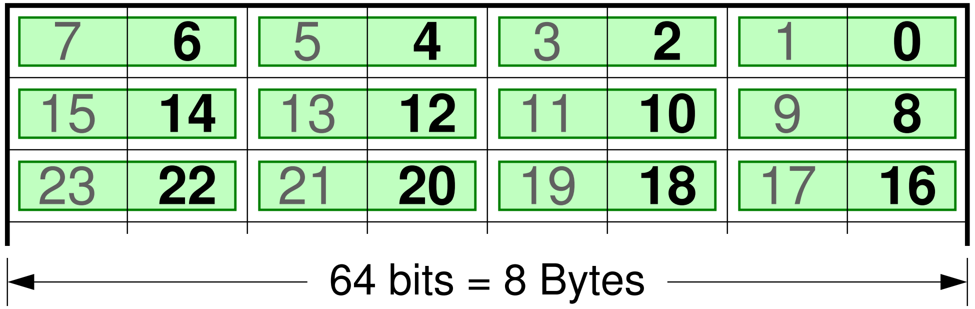
Addresses
drawn
assuming:
Little Endian
layout



1	0
3	2
5	4
7	6
9	8
11	10
13	12
15	14
17	16
19	18
21	20
	22



3	**2**	1	**0**
7	**6**	5	**4**
11	**10**	9	**8**
15	**14**	13	**12**
19	**18**	17	**16**
23	**22**	21	**20**

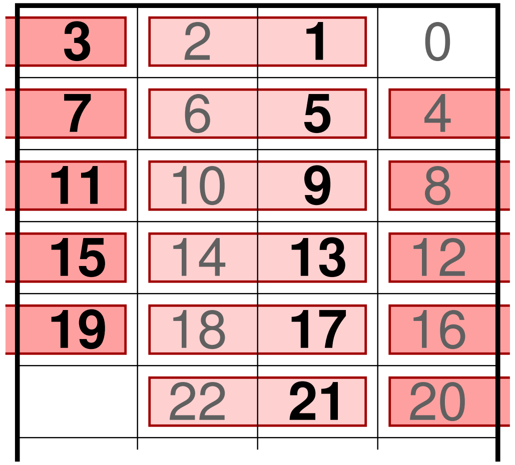


7	**6**	5	**4**	3	**2**	1	**0**
15	**14**	13	**12**	11	**10**	9	**8**
23	**22**	21	**20**	19	**18**	17	**16**

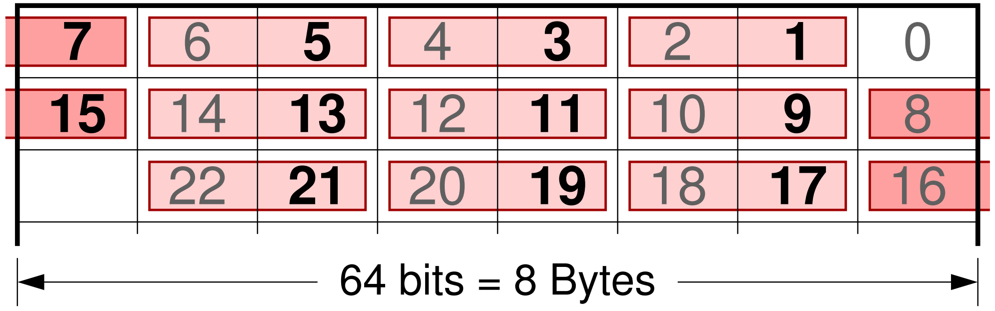
- Numbers inside boxes are Byte Addresses –NOT Contents
- The address of each 2–Byte "half word" is shown in Bold

(the address of a multi–Byte quantity is the address of that Byte inside it that has the smallest address among all the Bytes inside the quantity)

2–Byte "Half Words" at Addresses that are NOT integer multiples of 2



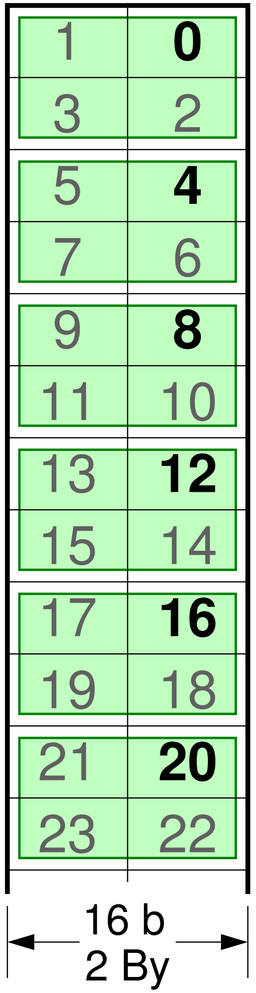
3	2	**1**	0
7	6	**5**	4
11	10	**9**	8
15	14	**13**	12
19	18	**17**	16
	22	**21**	20



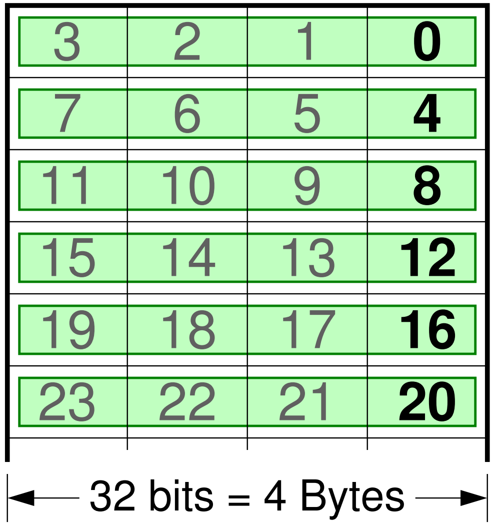
7	6	**5**	4	**3**	2	**1**	0
15	14	**13**	12	**11**	10	**9**	8
	22	**21**	20	**19**	18	**17**	16

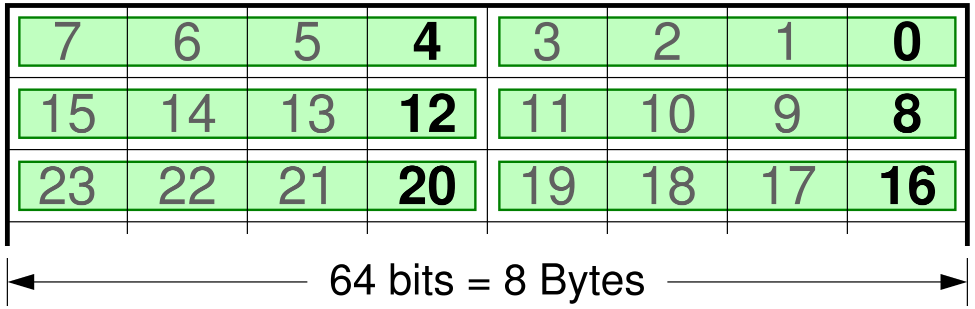
- Some (even if not all) of these 2–Byte half–w. incur a performance penalty when accessed

4–Byte "Words" Aligned on Addresses that are integer multiples of 4



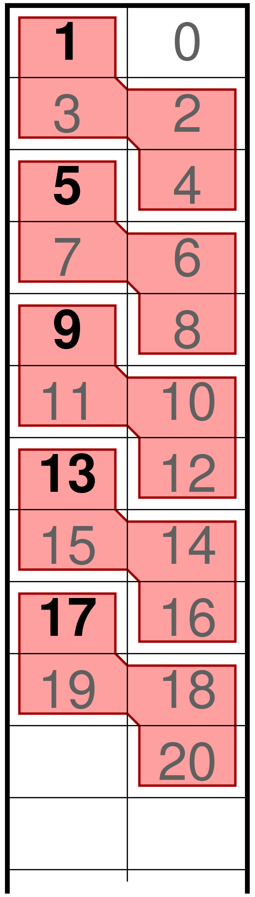
Addresses drawn assuming:
Little Endian layout



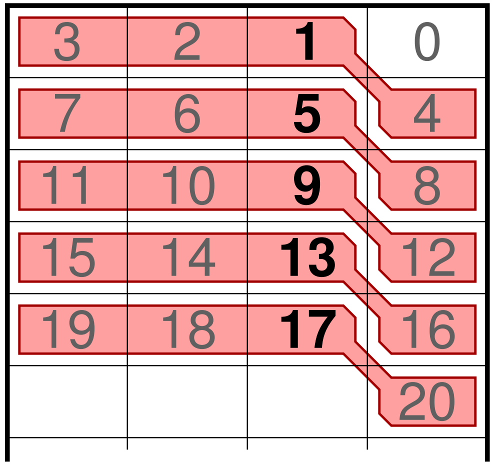


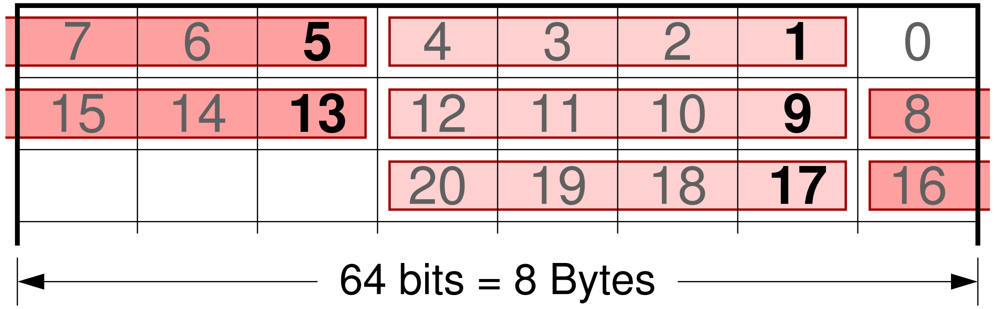
- Numbers inside boxes are Byte Addresses –NOT Contents
- The address of each 4–Byte "word" is shown in Bold

(the address of a multi–Byte quantity is the address of that Byte inside it that has the smallest address among all the Bytes inside the quantity)



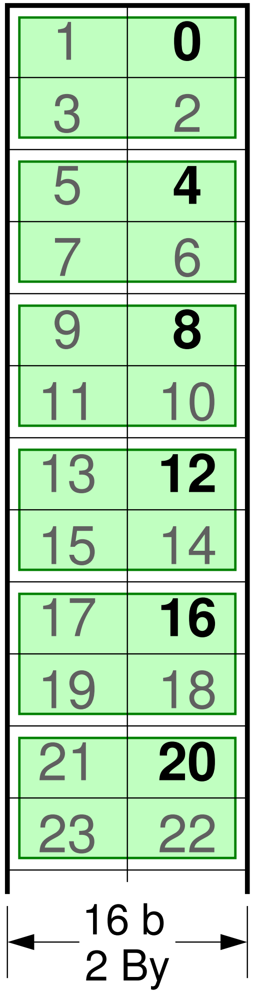
4–Byte "Words" at Addresses that are NOT multiples multiples of 4, but are 1–off, i.e. Addr mod 4 == 1



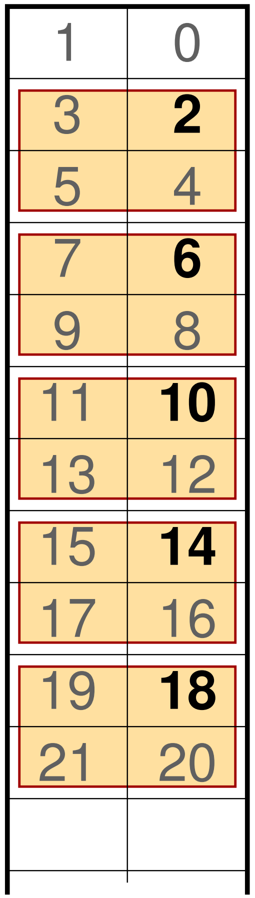


- Some (even if not all) of these 4–Byte words incur a performance penalty when accessed

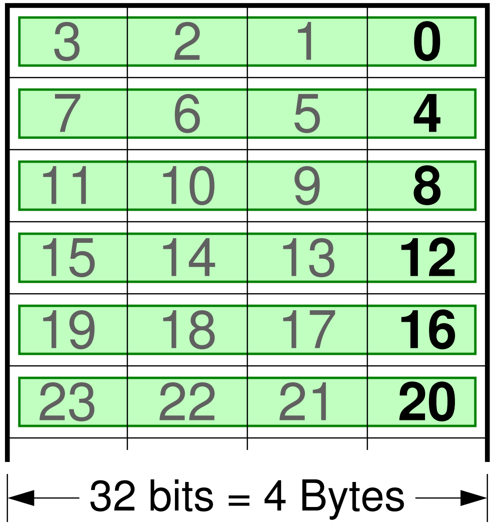
4–Byte "Words" Aligned on Addresses that are integer multiples of 4

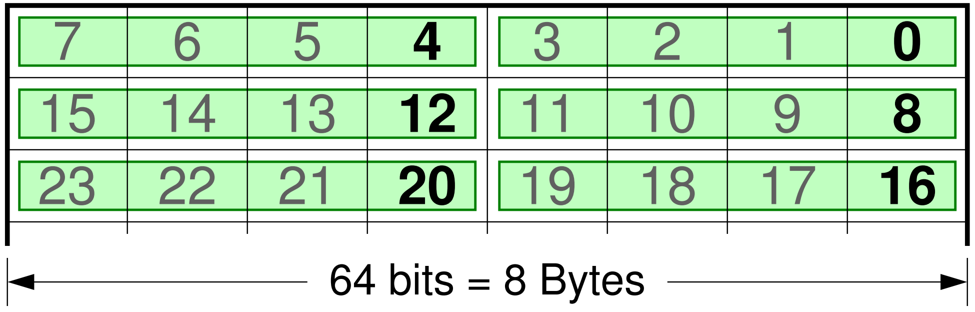


Addresses drawn assuming:
Little Endian layout



4–Byte words at multiples of 2 but not of 4 are OK in 2–Byte wide memories, but NOT in wider!

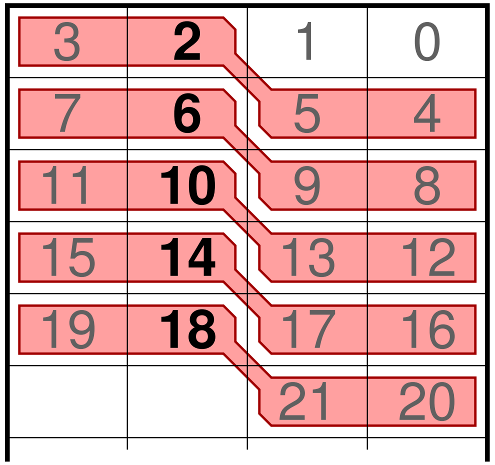


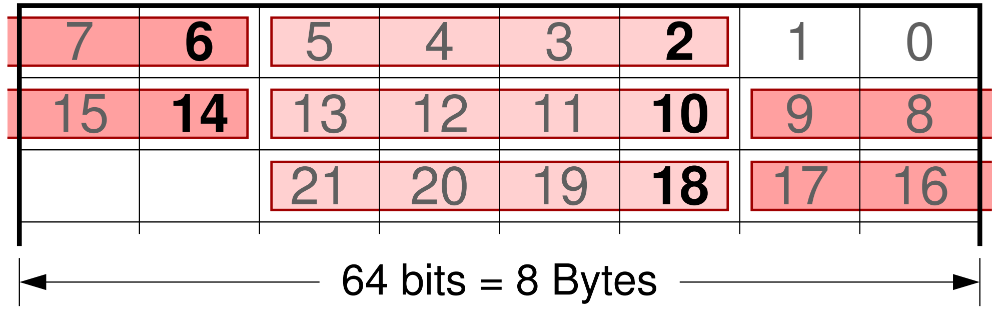


- Numbers inside boxes are Byte Addresses –NOT Contents
- The address of each 4–Byte "word" is shown in Bold

(the address of a multi–Byte quantity is the address of that Byte inside it that has the smallest address among all the Bytes inside the quantity)

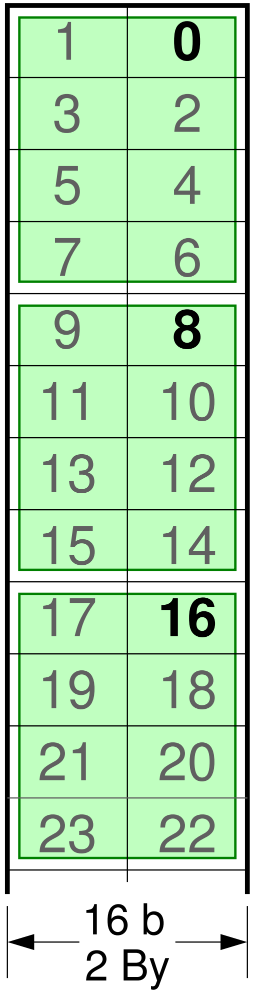
4–Byte "Words" at Addresses that are multiples of 2, but NOT multiples of 4 (i.e. Addr mod 4 == 2)





- Some (even if not all) of these 4–Byte words incur a performance penalty when accessed

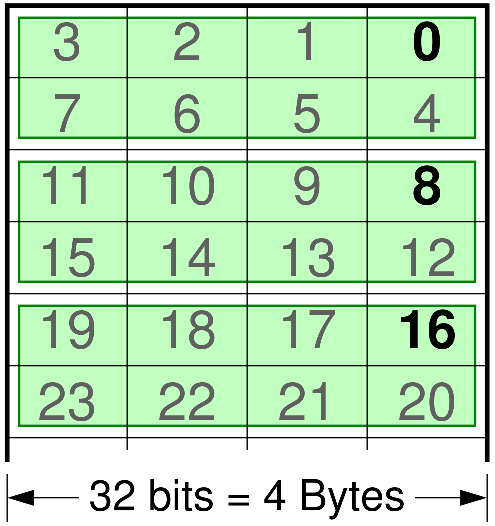
8–Byte "Double Words" Aligned on Addresses that are integer multiples of 8



1	**0**
3	2
5	4
7	6
9	**8**
11	10
13	12
15	14
17	**16**
19	18
21	20
23	22

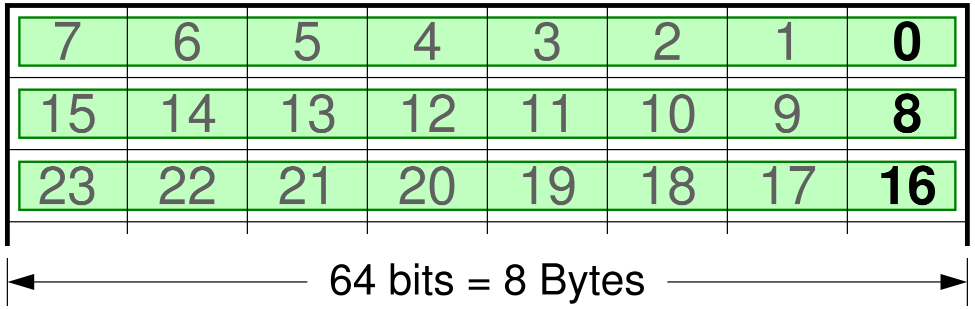
16 b
2 By

Addresses
drawn
assuming:
Little Endian
layout



3	2	1	**0**
7	6	5	4
11	10	9	**8**
15	14	13	12
19	18	17	**16**
23	22	21	20

32 bits = 4 Bytes



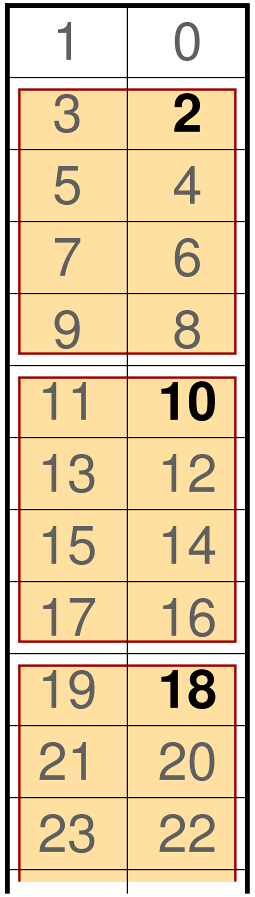
7	6	5	4	3	2	1	**0**
15	14	13	12	11	10	9	**8**
23	22	21	20	19	18	17	**16**

64 bits = 8 Bytes

- Numbers inside boxes are Byte Addresses –NOT Contents
- The address of each 8–Byte "double word" is shown in Bold

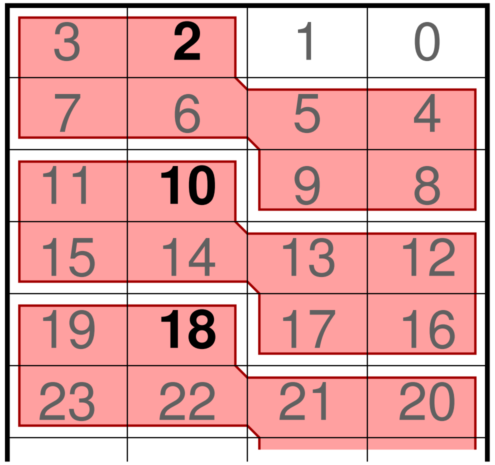
(the address of a multi–Byte quantity is the address of that Byte inside it that has the smallest address among all the Bytes inside the quantity)

8–Byte "Doubles" at Addresses that are multiples of 2, but NOT multiples of 4 or 8 (i.e. Addr mod 8 == 2)

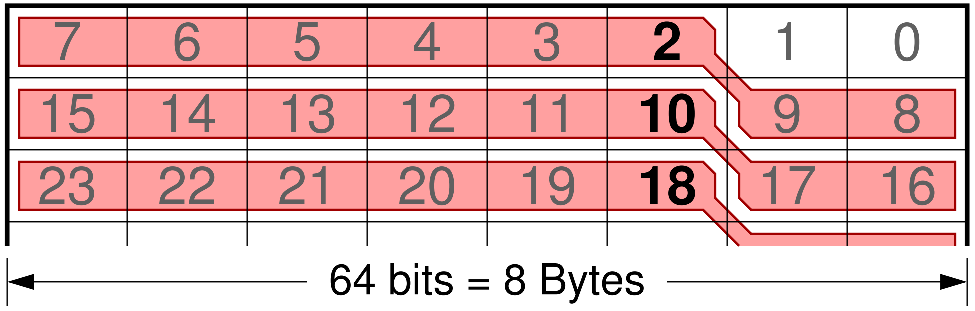


1	0
3	**2**
5	4
7	6
9	8
11	**10**
13	12
15	14
17	16
19	**18**
21	20
23	22

8–Byte doubles
at multiples of 2
but not of 4 or 8
are OK in 2–Byte
wide memories,
but NOT in wider!



3	**2**	1	0
7	6	5	4
11	**10**	9	8
15	14	13	12
19	**18**	17	16
23	22	21	20

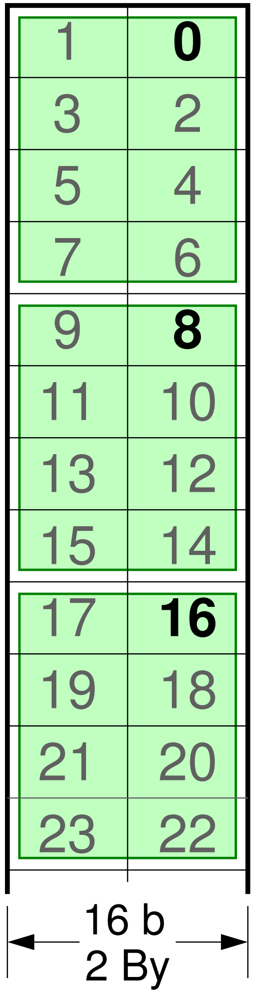


7	6	5	4	3	**2**	1	0
15	14	13	12	11	**10**	9	8
23	22	21	20	19	**18**	17	16

64 bits = 8 Bytes

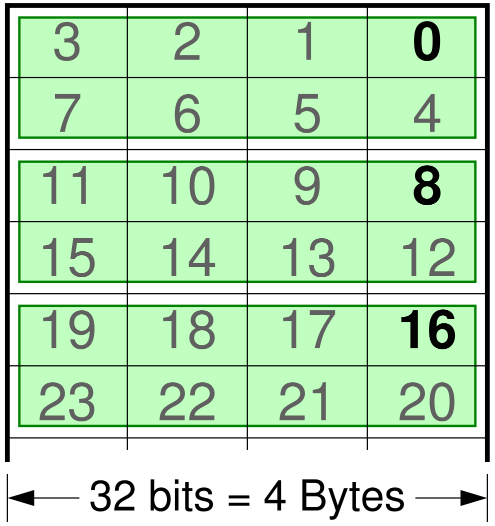
- In 4– and 8–wide memories, all of these doubles incur a performance penalty when accessed

8–Byte "Double Words" Aligned on Addresses that are integer multiples of 8

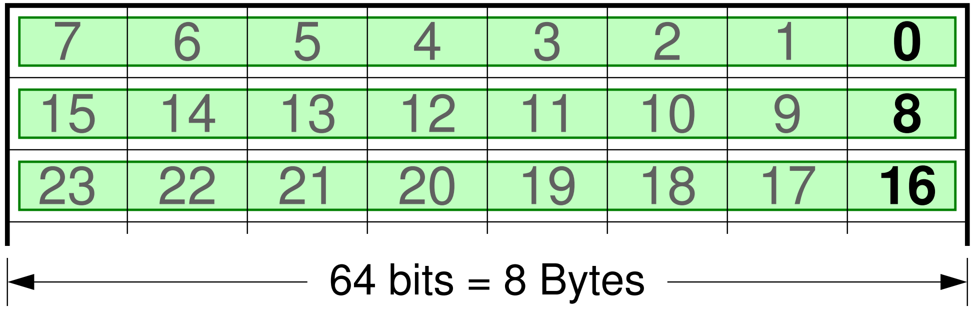


1	**0**
3	2
5	4
7	6
9	**8**
11	10
13	12
15	14
17	**16**
19	18
21	20
23	22

Addresses drawn assuming: *Little Endian layout*



3	2	1	**0**
7	6	5	4
11	10	9	**8**
15	14	13	12
19	18	17	**16**
23	22	21	20

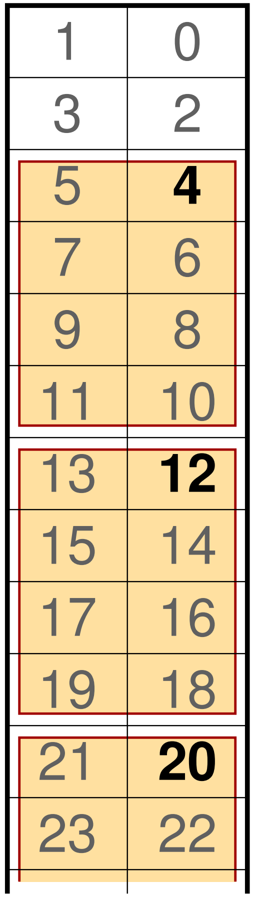


7	6	5	4	3	2	1	**0**
15	14	13	12	11	10	9	**8**
23	22	21	20	19	18	17	**16**

- Numbers inside boxes are Byte Addresses –NOT Contents
- The address of each 8–Byte "Double word" is shown in Bold

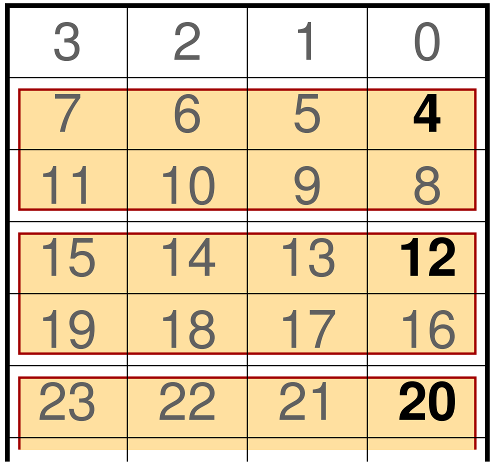
(the address of a multi–Byte quantity is the address of that Byte inside it that has the smallest address among all the Bytes inside the quantity)

8–Byte "Doubles" at Addresses that are multiples of 4, but NOT multiples of 8 (i.e. Addr mod 8 == 4)

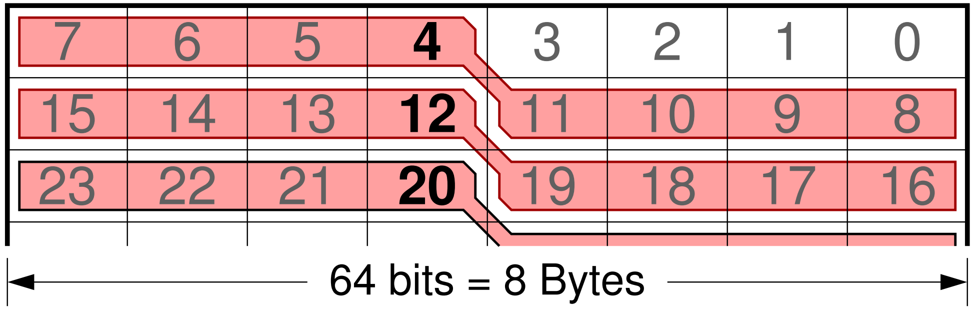


1	0
3	2
5	**4**
7	6
9	8
11	10
13	**12**
15	14
17	16
19	18
21	**20**
23	22

8–Byte Doubles at multiples of 4 but not of 8 are OK in 2– & 4– wide memories, but NOT in wider!



3	2	1	0
7	6	5	**4**
11	10	9	8
15	14	13	**12**
19	18	17	16
23	22	21	**20**



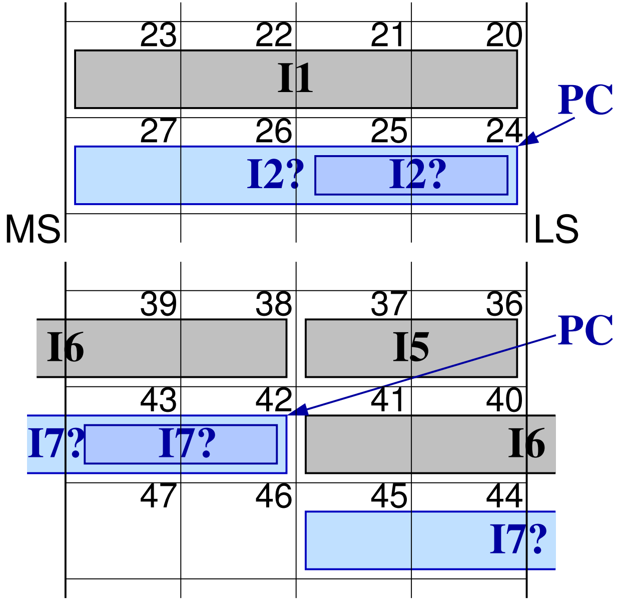
7	6	5	**4**	3	2	1	0
15	14	13	**12**	11	10	9	8
23	22	21	**20**	19	18	17	16

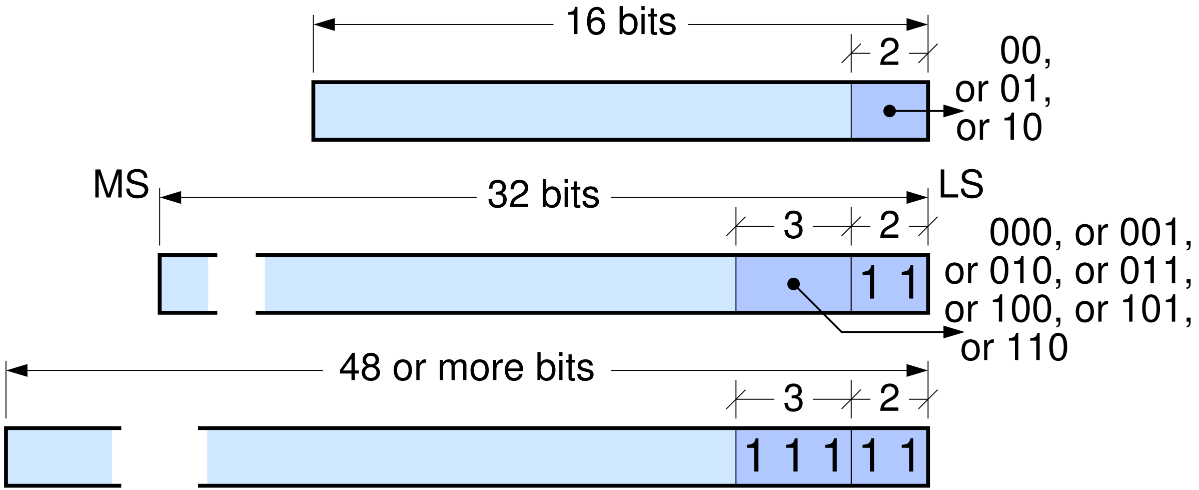
- In 8– Byte wide memories, these Doubles incur a performance penalty when accessed

Variable–size Instructions in Little–Endian Memory:

Opcode must be in LS part of the instruction

(RISC–V allows for optional "C" extension that includes Compact 16–bit instructions)

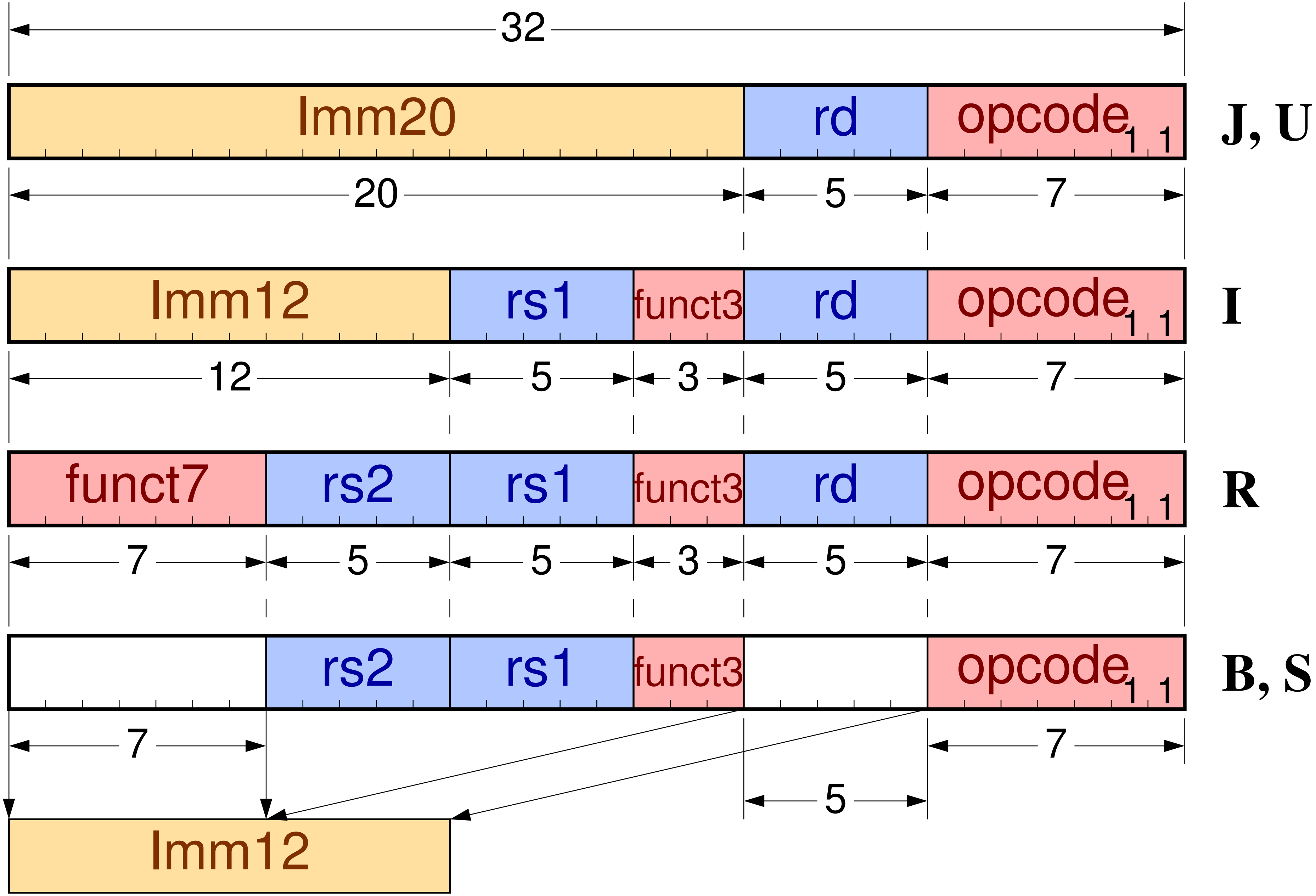




RISC-V R-format Instructions

funct7	rs2	rs1	funct3	rd	opcode
7 bits	5 bits	5 bits	3 bits	5 bits	7 bits

- Instruction fields
  - opcode: operation code
  - rd: destination register number
  - funct3: 3-bit function code (additional opcode)
  - rs1: the first source register number
  - rs2: the second source register number
  - funct7: 7-bit function code (additional opcode)



Source/Destination Register Fields always at fixed locations

so as to check data dependencies with other Instructions fast, and read src reg's fast

Conditional Operations

- Branch to a labeled instruction if a condition is true
  - Otherwise, continue sequentially
- `beq rs1, rs2, L1`
  - if (rs1 == rs2) branch to instruction labeled L1
- `bne rs1, rs2, L1`
  - if (rs1 != rs2) branch to instruction labeled L1

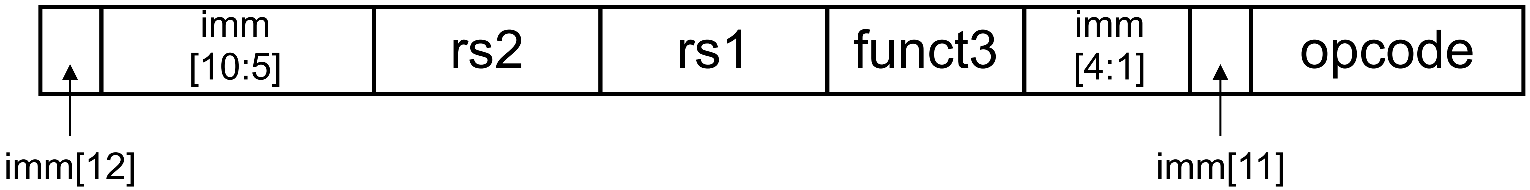
More Conditional Operations

- `blt rs1, rs2, L1`
  - if (rs1 < rs2) branch to instruction labeled L1
- `bge rs1, rs2, L1`
  - if (rs1 >= rs2) branch to instruction labeled L1
- Example
  - if (a > b) a += 1;
  - a in x22, b in x23

```
      bge  x23, x22, Exit     // branch if b >= a
      addi x22, x22, 1
Exit:
```

Branch Addressing

- Branch instructions specify
  - Opcode, two registers, target address
- Most branch targets are near branch
  - Forward or backward
- SB format:



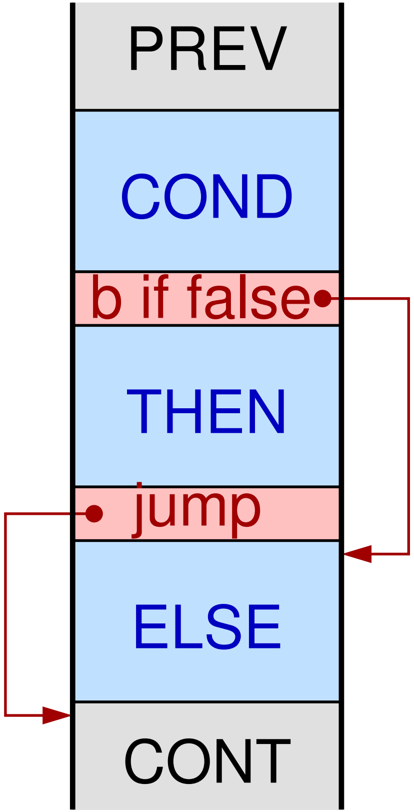
	imm [10:5]	rs2	rs1	funct3	imm [4:1]		opcode

imm[12]

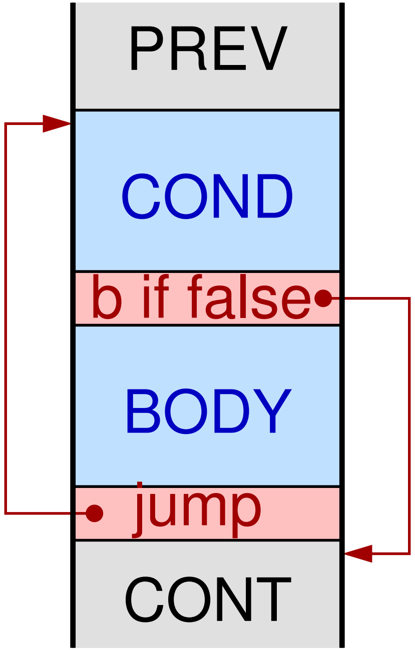
imm[11]

- PC-relative addressing
  - Target address = PC + immediate × 2

```
PREV;
if (COND) {
      THEN;
} else {
      ELSE;
}
CONT;
```



```
PREV;
while (COND) {
      BODY;
}
CONT;
```



Exercise: rearrange COND / BODY so as to execute only one CTI (br/jmp) on most iterations also used as JUMP pseudoinstructions

Procedure Call Instructions

- Procedure call: jump and link

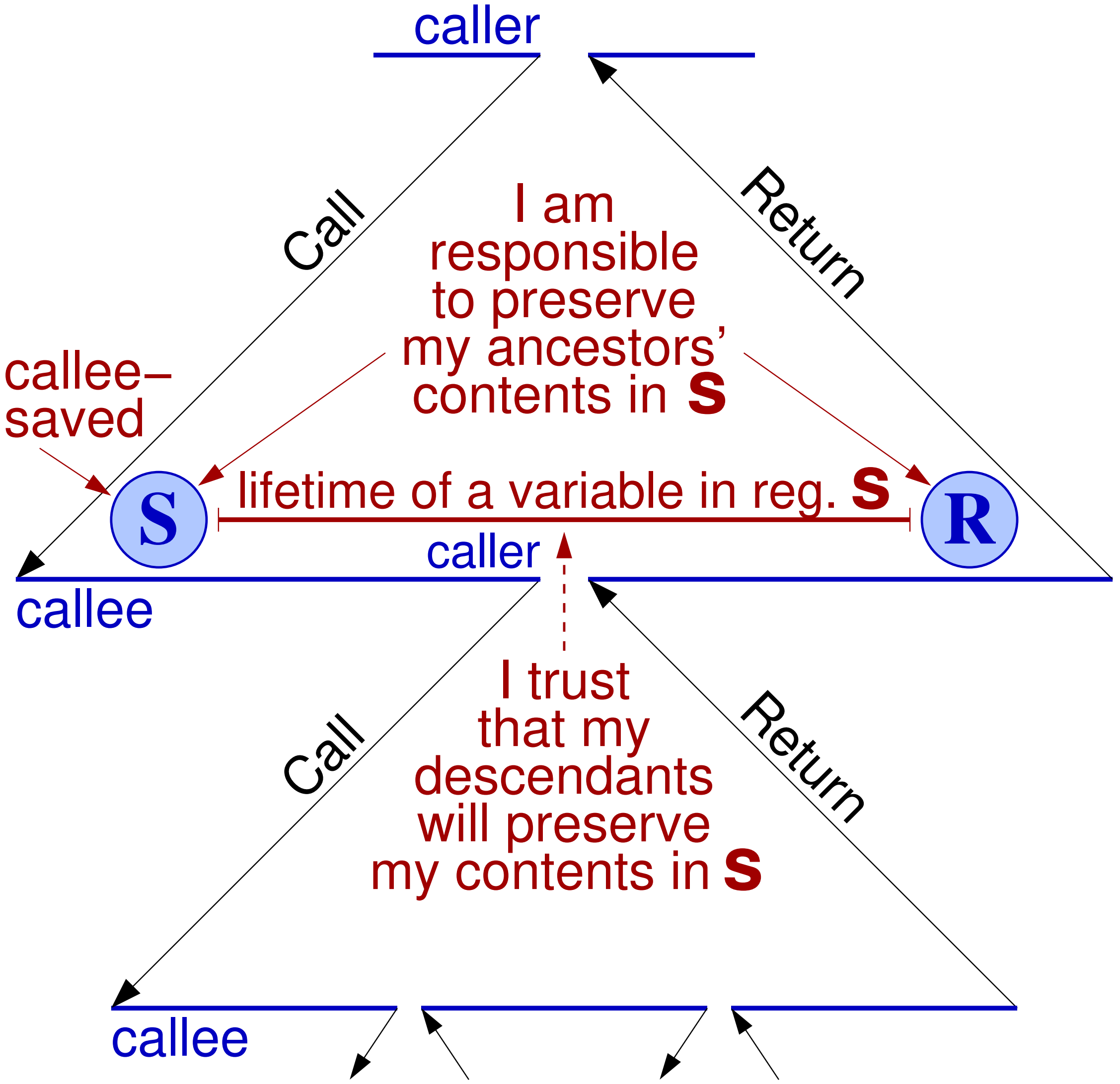
  `jal x1, ProcedureLabel` if rd==x0 ==> Jump

  - Address of following instruction put in x1
  - Jumps to target address = PC + 2 x (Imm20)
- Procedure return: jump and link register

  `jalr x0, 0(x1)` PCnew = rs1 + Immediate12

  - Like jal, but jumps to 0 + address in x1
  - Use x0 as rd (x0 cannot be changed)
  - Can also be used for computed jumps
    - e.g., for case/switch statements

S (saved) Registers: Callee–saved

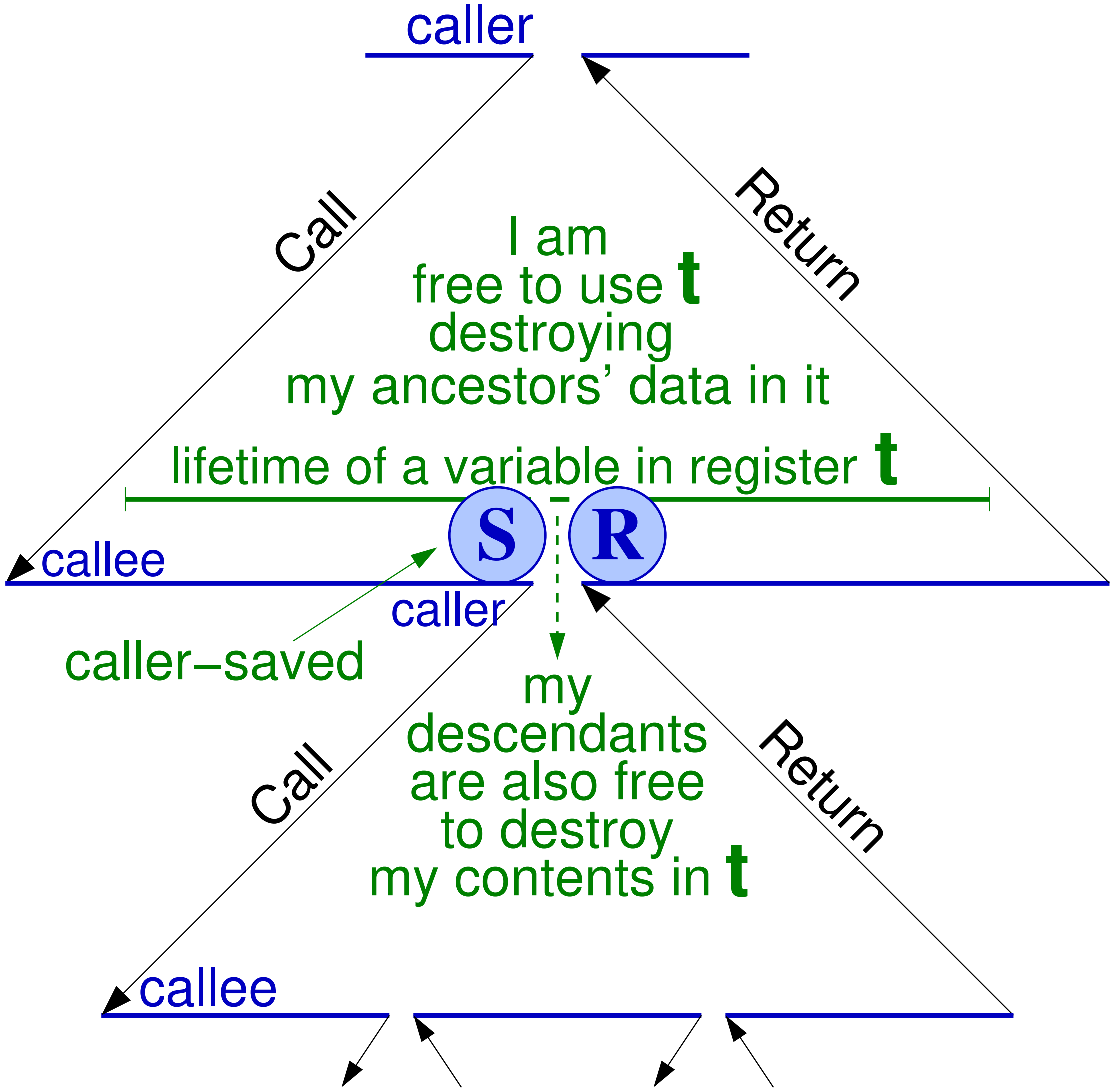


I am responsible to preserve my ancestors' contents in **s**

lifetime of a variable in reg. **s**

I trust that my descendants will preserve my contents in **s**

T (temporary) Registers: Caller–saved



I am free to use **t** destroying my ancestors' data in it

lifetime of a variable in register **t**

my descendants are also free to destroy my contents in **t**

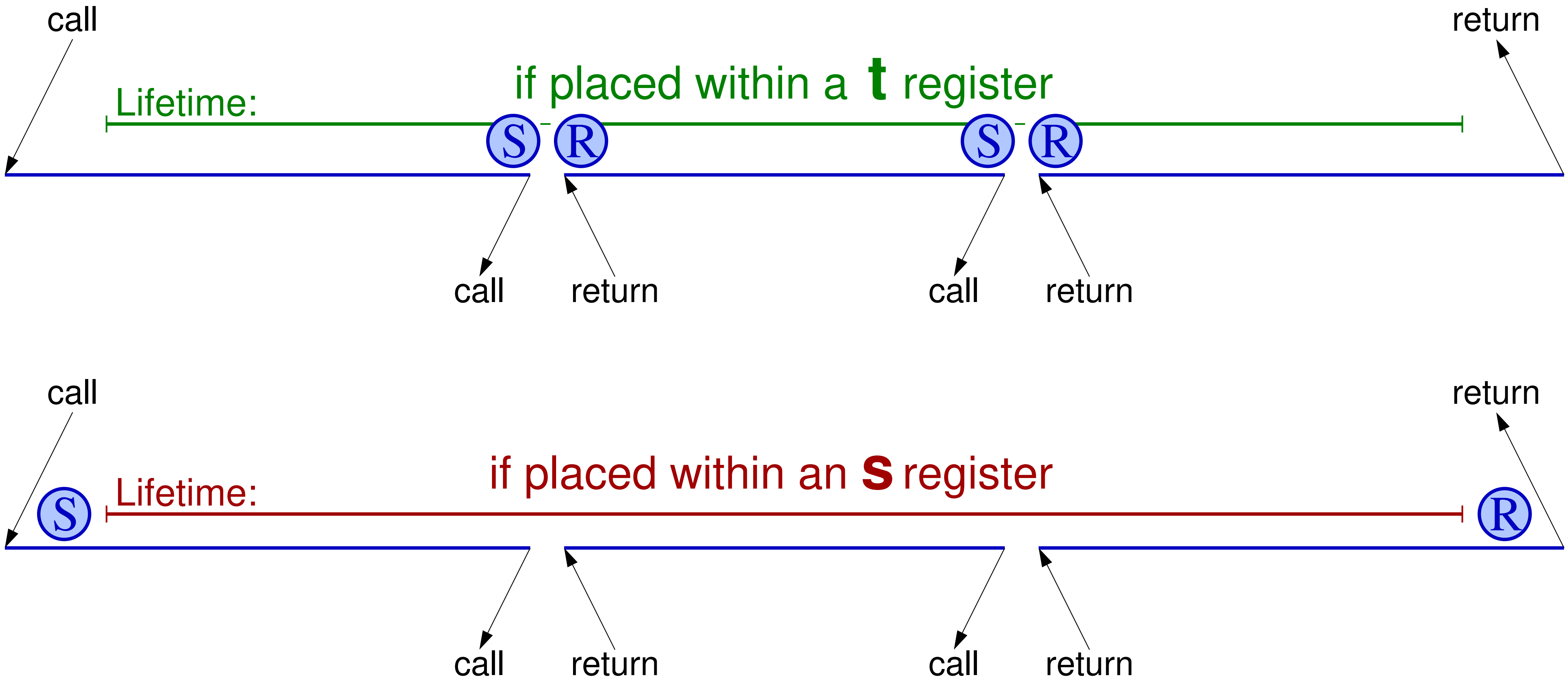
S = save register xi (s or t) on the stack

```
addi  sp,  sp,  -8
sd    xi,  0(sp)
```

R = restore register xi (s or t) from the stack

```
ld    xi,  0(sp)
addi  sp,  sp,  8
```

Lifetimes of variables that span 2 or more procedure Calls

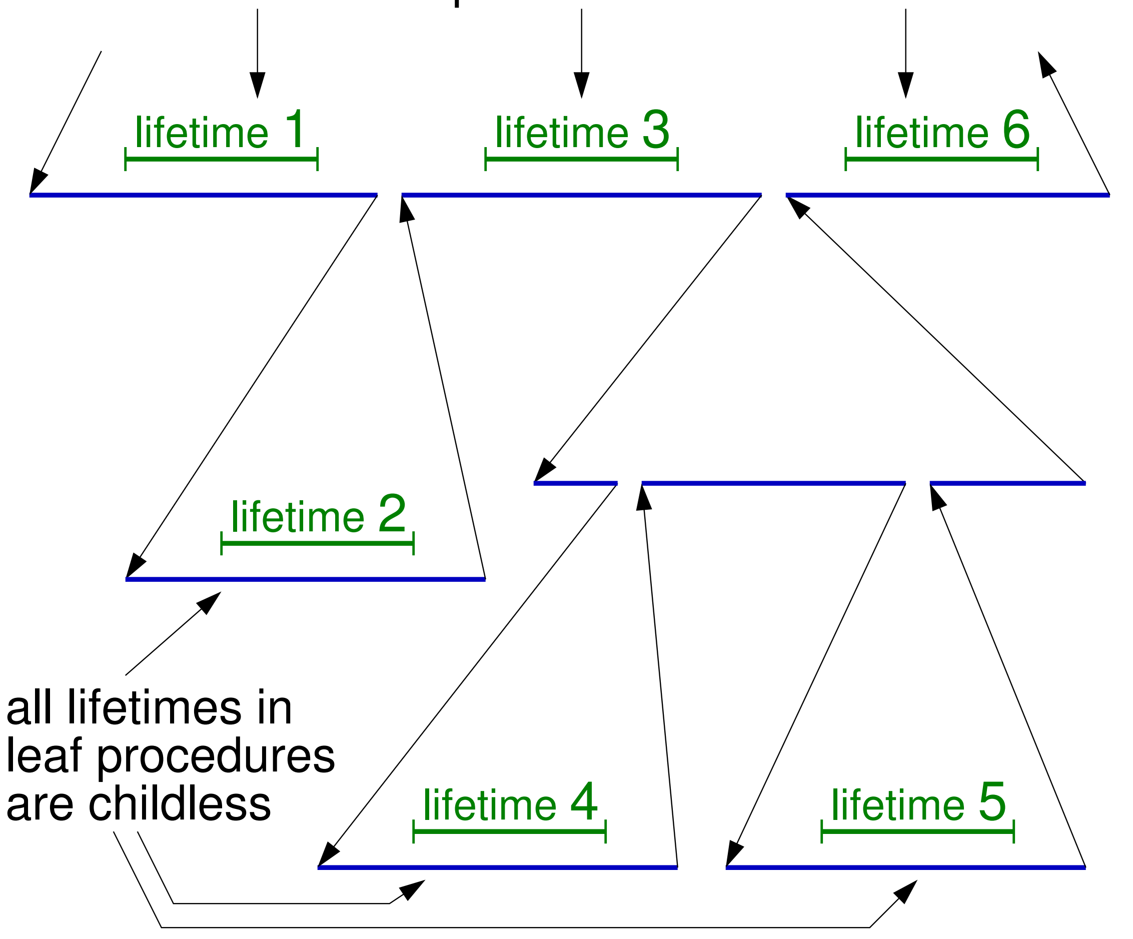


"s" (saved) register is preferable: fewer save–restores to stack

Lifetimes of variables that contain no procedure Calls

if placed within **t** registers:

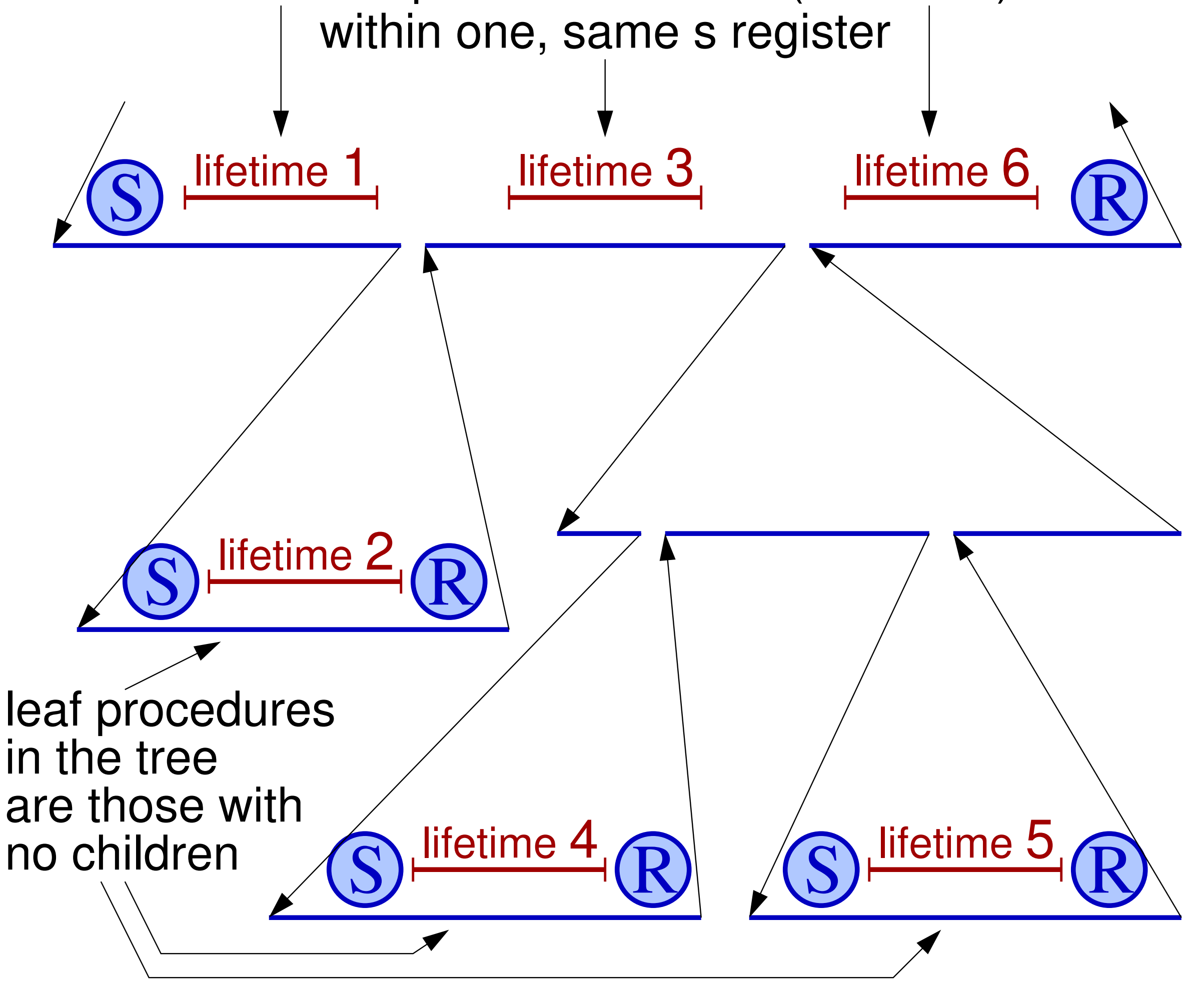
three independent lifetimes (variables):
no data need to be preserved from one to the other



all lifetimes in
leaf procedures
are childless

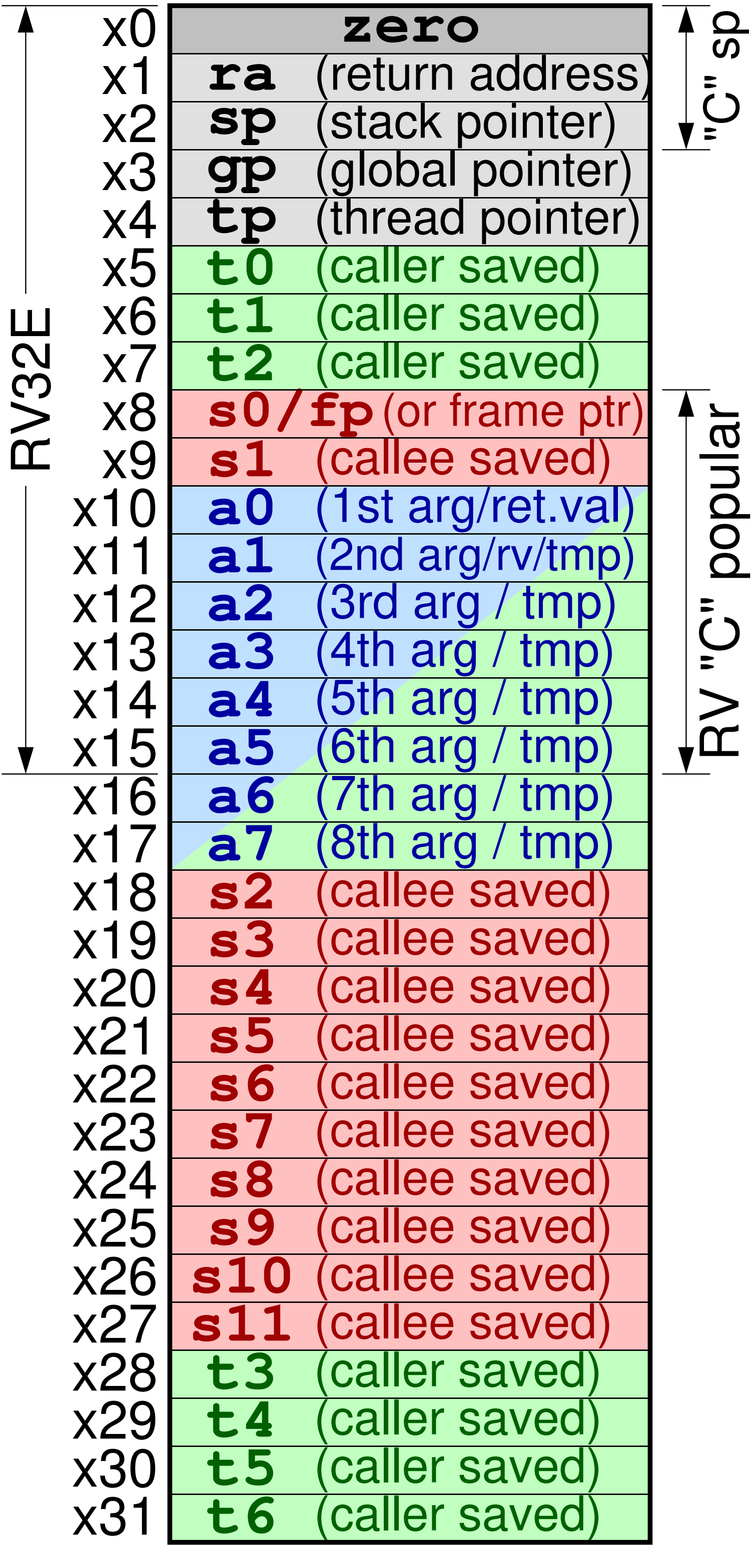
if placed within **s** registers:

three independent lifetimes (variables)
within one, same s register



leaf procedures
in the tree
are those with
no children

"t" registers are preferable for childless lifetimes: no save−restores to stack



x0	**zero**	
x1	**ra**	(return address)
x2	**sp**	(stack pointer)
x3	**gp**	(global pointer)
x4	**tp**	(thread pointer)
x5	**t0**	(caller saved)
x6	**t1**	(caller saved)
x7	**t2**	(caller saved)
x8	**s0/fp**	(or frame ptr)
x9	**s1**	(callee saved)
x10	**a0**	(1st arg/ret.val)
x11	**a1**	(2nd arg/rv/tmp)
x12	**a2**	(3rd arg / tmp)
x13	**a3**	(4th arg / tmp)
x14	**a4**	(5th arg / tmp)
x15	**a5**	(6th arg / tmp)
x16	**a6**	(7th arg / tmp)
x17	**a7**	(8th arg / tmp)
x18	**s2**	(callee saved)
x19	**s3**	(callee saved)
x20	**s4**	(callee saved)
x21	**s5**	(callee saved)
x22	**s6**	(callee saved)
x23	**s7**	(callee saved)
x24	**s8**	(callee saved)
x25	**s9**	(callee saved)
x26	**s10**	(callee saved)
x27	**s11**	(callee saved)
x28	**t3**	(caller saved)
x29	**t4**	(caller saved)
x30	**t5**	(caller saved)
x31	**t6**	(caller saved)

32-bit Constants

- Most constants are small
  - 12-bit immediate is sufficient
- For the occasional 32-bit constant

```
lui rd, constant
```

  - Copies 20-bit constant to bits [31:12] of rd
  - Extends bit 31 to bits [63:32]
  - Clears bits [11:0] of rd to 0

```
lui x19, 976  // 0x003D0
```

0000 0000 0000 0000	0000 0000 0000 0000	0000 0000 0011 1101 0000	0000 0000 0000

```
addi x19,x19,128  // 0x500
```

0000 0000 0000 0000	0000 0000 0000 0000	0000 0000 0011 1101 0000	0101 0000 0000

Other RISC-V Instructions

- Base integer instructions (RV64I)
  - Those previously described, plus
  - auipc rd, immed // rd = (imm<<12) + pc
    - follow by jalr (adds 12-bit immed) for long jump
  - slt, sltu, slti, sltui: set less than (like MIPS)
  - addw, subw, addiw: 32-bit add/sub
  - sllw, srlw, srlw, slliw, srliw, sraiw: 32-bit shift
- 32-bit variant: RV32I
  - registers are 32-bits wide, 32-bit operations