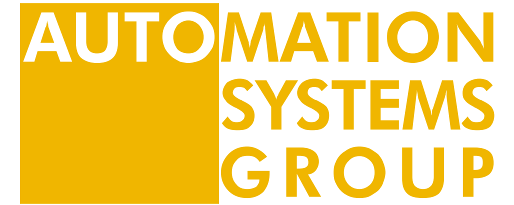
Informatics



Computersysteme

Architecture

Markus Bader

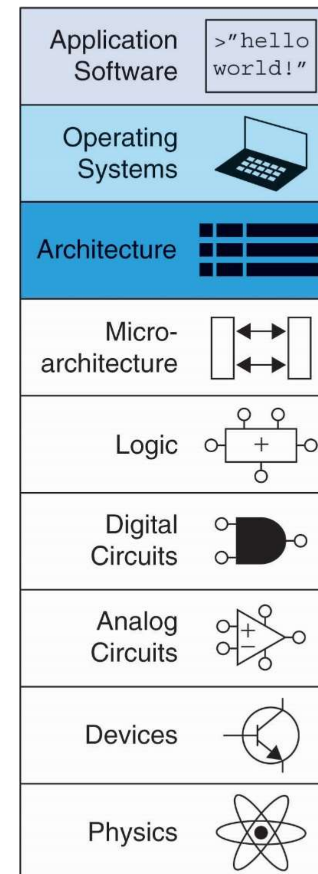
11.03.2024

RISC-V Architecture Introduction

DDCA Ch6 - Part 1: Architecture Introduction https://www.youtube.com/watch?v=uYmEYx5UeHo

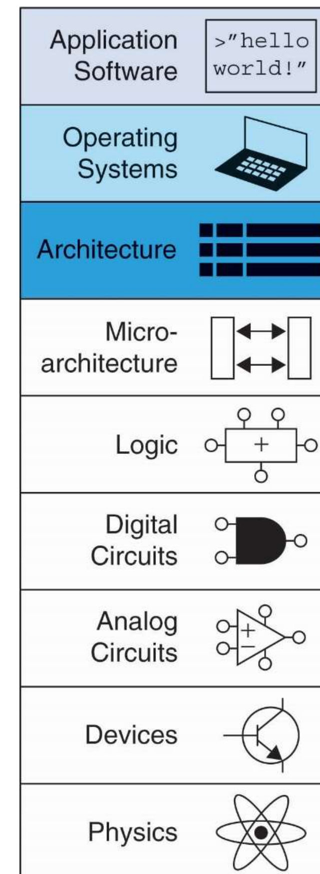
Chapter 6 :: Topics

- Introduction
- Assembly Language
- Programming
- Machine Language
- Addressing Modes
- Lights, Camera, Action: Compiling, Assembly, & Loading
- Odds & Ends



Introduction

- Jumping up a few levels of abstraction
- **Architecture**: programmer's view of computer
- Defined by instructions & operand locations
- **Microarchitecture**: how to implement an architecture in hardware (covered in Chapter 7)



Assembly Language

- **Instructions**: commands in a computer's language
  - **Assembly language**: human-readable format of instructions
  - **Machine language**: computer-readable format (1's and 0's)
- RISC-V architecture:
  - Developed by Krste Asanovic, David Patterson and their colleagues at UC Berkeley in 2010.
  - First widely accepted open-source computer architecture

Once you've learned one architecture, it's easier to learn others

Kriste Asanovic

- Professor of Computer Science at the University of California, Berkeley
- Developed RISC-V during one summer
- Chairman of the Board of the RISC-V Foundation
- Co-Founder of SiFive, a company that commercializes and develops supporting tools for RISC-V

Kriste Asanovic

- Co-founded SiFive with Krste Asanovic
- Weary of existing instruction set architectures (ISAs), he co-designed the RISC-V architecture and the first RISC-V cores
- Earned his PhD in computer science from UC Berkeley in 2016

David Patterson

- Professor of Computer Science at the University of California, Berkeley since 1976
- Coinvented the Reduced Instruction Set Computer (RISC) with John Hennessy in the 1980's
- Founding member of RISC-V team.
- Was given the Turing Award (with John Hennessy) for pioneering a quantitative approach to the design and evaluation of computer architectures.

John Hennessy

- President of Stanford University from 2000 - 2016.
- Professor of Electrical Engineering and Computer Science at Stanford since 1977
- Coinvented the Reduced Instruction Set Computer (RISC) with David Patterson in the 1980’s
- Was given the Turing Award (with David Patterson) for pioneering a quantitative approach to the design and evaluation of computer architectures.

Architecture Design Principles

Underlying design principles, as articulated by Hennessy and Patterson:

1. Simplicity favors regularity

2. Make the common case fast

3. Smaller is faster

4. Good design demands good compromises

Instructions

DDCA Ch6 - Part 2: Instructions https://www.youtube.com/watch?v=6a21EM5AXvs

Instructions: Addition

C Code

```
a = b + c;
```

RISC-V assembly code

```
add a, b, c
```

- `add` mnemonic indicates operation to perform
- `b, c` source operands (on which the operation is performed)
- `a` destination operand (to which the result is written)

Instructions: Subtraction

C Code

```
a = b - c;
```

RISC-V assembly code

```
sub a, b, c
```

- **`sub`** mnemonic indicates operation to perform
- **`b, c`** source operands (on which the operation is performed)
- **`a`** destination operand (to which the result is written)

Design Principle 1

Simplicity favors regularity

- Consistent instruction format
- Same number of operands (two sources and one destination)
- Easier to encode and handle in hardware

Multiple Instructions

C Code

```
a = b - c - d;
```

RISC-V assembly code

```
add t, b, c   # t = b + c
sub a, t, d   # a = t - d
```

Design Principle 2

Make the common case fast

- RISC-V includes only simple, commonly used instructions
- Hardware to decode and execute instructions can be simple, small, and fast
- More complex instructions (that are less common) performed using multiple simple instructions
- RISC-V is a ***reduced instruction set computer*** (RISC), with a small number of simple instructions
- Other architectures, such as Intel's x86, are ***complex instruction set computers*** (CISC)

Operands

DDCA Ch6 - Part 3: Operands https://www.youtube.com/watch?v=21fFcoQr6rg

Operands

- **Operand location**: physical location in computer
  - Registers
  - Memory
  - Constants (also called immediates)

Operands: Registers

- RISC-V has 32 32-bit registers
- Registers are faster than memory
- RISC-V called “32-bit architecture” because it operates on 32-bit data

Design Principle 3

Smaller is Faster

- RISC-V includes only a small number of registers

RISC-V Register Set

Name	Register Number	Usage
zero	x0	Constant value 0
ra	x1	Return address
sp	x2	Stack pointer
gp	x3	Global pointer
tp	x4	Thread pointer
t0-2	x5-7	Temporaries
s0/fp	x8	Saved register / Frame pointer
s1	x9	Saved register
a0-1	x10-11	Function arguments / return values
a2-7	x12-17	Function arguments
s2-11	x18-27	Saved registers
t3-6	x28-31	Temporaries

Operands: Registers

- **Registers:**
  - Can use either name (i.e., `ra, zero`) or x0, x1, etc.
  - Using name is preferred
- Registers used for specific purposes:
  - zero always holds the **constant value 0**.
  - the **saved registers**, `s0-s11`, used to hold variables
  - the **temporary registers**, `t0-t6`, used to hold intermediate values during a larger computation
  - Discuss others later

Instructions with Registers

C Code

```
a = b + c;
```

RISC-V assembly code

```
# s0 = a, s1 = b, s2 = c
add s0, s1, s2
```

# indicates a single-line comment

Instructions with Constants

C Code

```
a = b + 6;
```

RISC-V assembly code

```
# s0 = a, s1 = b
addi s0, s1, 6
```

addi mnemonic indicates operation to perform

Adds a constant to the source operator

- constant (immediates) with 12 bit
- sign extended

Memory Operands

DDCA Ch6 - Part 4: Memory https://www.youtube.com/watch?v=wFhbmPuykWQ

Operands: Memory

- Too much data to fit in only 32 registers
- Store more data in memory
- Memory is large, but slow
- Commonly used variables kept in registers

Memory

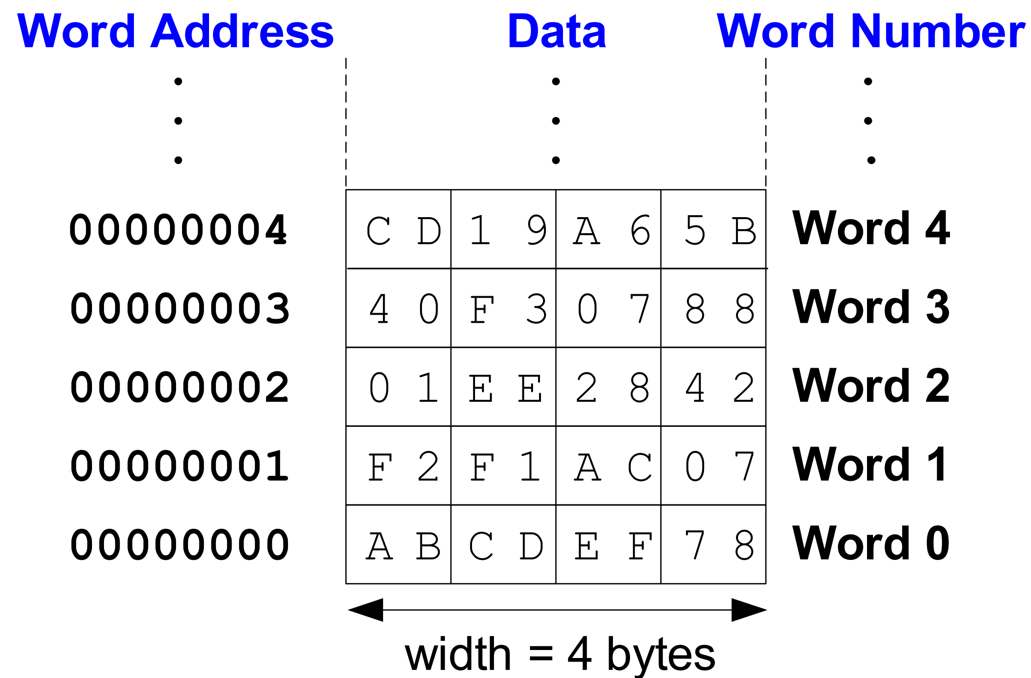
- First, we'll discuss **word-addressable** memory
- Then we'll discuss **byte-addressable** memory

RISC-V is **byte-addressable**

Word-Addressable Memory

RISC-V uses **byte-addressable** memory, which we'll talk about next.

- Each 32-bit data word has a unique address



Reading Word-Addressable Memory

RISC-V uses **byte-addressable** memory, which we'll talk about next.

- Memory read called load
- **Mnemonic:** load word (`lw`)
- **Format:**

```
lw t1, 5(s0)
lw destination, offset(base)
```

- **Address calculation:**
  - add base address (`s0`) to the offset (`5`)
  - address = (`s0 + 5`)
- **Result:**
  - `t1` holds the data value at address (`s0 + 5`)
- **Any register** may be used as base address

Example: read a word of data at memory address 1 into s3

- Assembly code
- Result

Word Address	Data	Word Number
⋮	⋮	⋮
00000004	C D 1 9 A 6 5 B	Word 4
00000003	4 0 F 3 0 7 8 8	Word 3
00000002	0 1 E E 2 8 4 2	Word 2
00000001	**F 2 F 1 A C 0 7**	Word 1
00000000	A B C D E F 7 8	Word 0

Reading Word-Addressable Memory

RISC-V uses **byte-addressable** memory, which we'll talk about next.

- Memory read called load
- **Mnemonic:** load word (`lw`)
- **Format:**

```
lw t1, 5(s0)
lw destination, offset(base)
```

- **Address calculation:**
  - add base address (`s0`) to the offset (`5`)
  - address = (`s0 + 5`)
- **Result:**
  - `t1` holds the data value at address (`s0 + 5`)

- **Any register** may be used as base address

Example: read a word of data at memory address 1 into s3

- Assembly code

```
# read memory word 1 into s3
lw s3, 1(zero)
```

- Result

`s3 = 0xF2F1AC07` after load

Word Address	Data	Word Number
⋮	⋮	⋮
00000004	C D 1 9 A 6 5 B	Word 4
00000003	4 0 F 3 0 7 8 8	Word 3
00000002	0 1 E E 2 8 4 2	Word 2
00000001	**F 2 F 1 A C 0 7**	Word 1
00000000	A B C D E F 7 8	Word 0

Writing Word-Addressable Memory

RISC-V uses **byte-addressable** memory, which we'll talk about next.

- Memory write is called a **store**
- **Mnemonic:** store word (`lw`)
- **Format:**

```
# write the value in t4 to memory word 3
sw t4, 0x3(zero)
```

Address calculation:

- add base the address (`zero`) to the offset (`0x3`)
- Offset can be written in decimal (default) or hexadecimal

- **Result:**
  - for example, if `t4` holds the value `0xFEEDCABB`

Word Address	Data	Word Number
⋮	⋮	⋮
00000004	C D 1 9 A 6 5 B	Word 4
00000003	4 0 F 3 0 7 8 8	Word 3
00000002	0 1 E E 2 8 4 2	Word 2
00000001	F 2 F 1 A C 0 7	Word 1
00000000	A B C D E F 7 8	Word 0

- **Any register** may be used as base address

Writing Word-Addressable Memory

RISC-V uses **byte-addressable** memory, which we'll talk about next.

- Memory write is called a **store**
- **Mnemonic:** store word (`lw`)
- **Format:**

```
# write the value in t4 to memory word 3
sw t4, 0x3(zero)
```

Address calculation:

- add base the address (`zero`) to the offset (`0x3`)
- Offset can be written in decimal (default) or hexadecimal

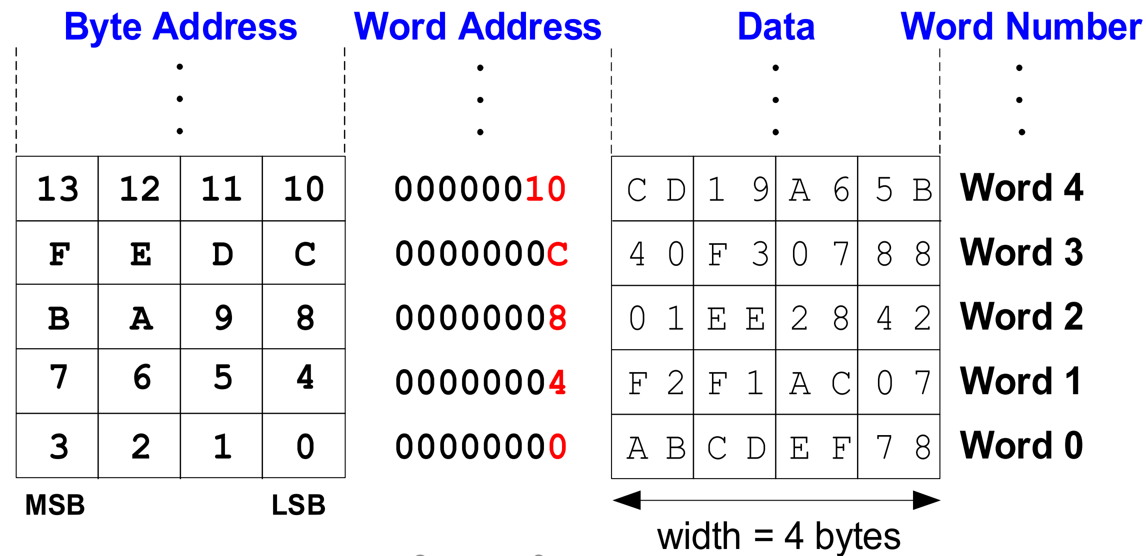
- **Result:**
  - for example, if `t4` holds the value `0xFEEDCABB`
  - then after this instruction completes, word 3 in memory will contain that value

- **Any register** may be used as base address

Word Address	Data	Word Number
⋮	⋮	⋮
00000004	C D 1 9 A 6 5 B	Word 4
00000003	**F E E D C A B B**	Word 3
00000002	0 1 E E 2 8 4 2	Word 2
00000001	F 2 F 1 A C 0 7	Word 1
00000000	A B C D E F 7 8	Word 0

Byte-Addressable Memory

- Each data byte has a unique address
- Load/store words or single bytes: load byte (`lb`) and store byte (`sb`)
- 32-bit word = 4 bytes, so word address **increments by 4**



Reading Byte-Addressable Memory

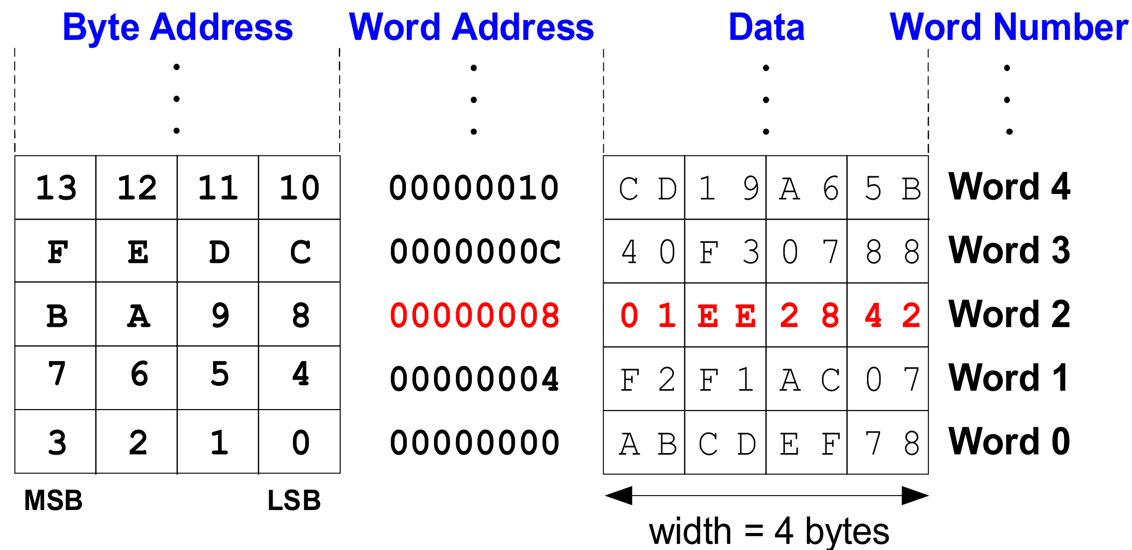
- The address of a memory word must now be multiplied by 4. For example,
  - the address of memory word 2 is
    $2 \times 4 = 8$
  - the address of memory word 10 is
    $10 \times 4 = 40$ (0x28)

RISC-V is **byte-addressed**, not word-addressed

Reading Byte-Addressable Memory

- **Example**: Load a word of data at memory address 8 into `s3`.
- s3 holds the value `0x1EE2842` after load
- RISC-V assembly code

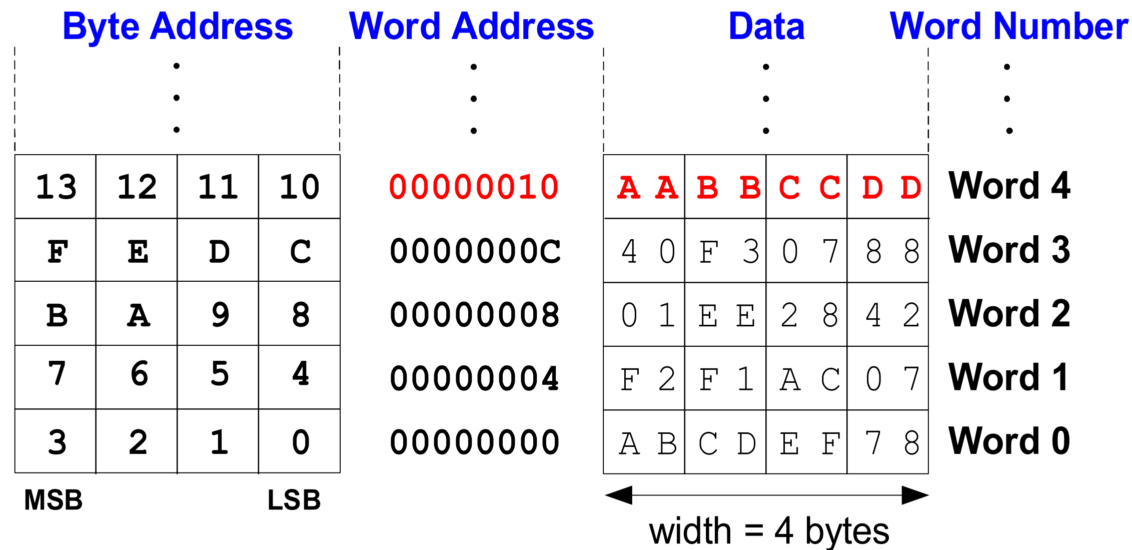
```
lw s3, 8(zero)   # read word at address 8 into s3
```



Writing Byte-Addressable Memory

- **Example**: store the value held in `t7` into memory address `0x10` (16)
- if `t7` holds the value `0xAABBCCDD`, then after the `sw` completes, word 4 (at address `0x10`) in memory will contain that value
- **RISC-V assembly code**

```
sw t7, 0x10(zero)   # write t7 into address 16
```



Generating 12-Bit Constants

DDCA Ch6 - Part 5: Generating Constants https://www.youtube.com/watch?v=VDrOoVTGhTc&t

Generating 12-Bit Constants

Generating 12-Bit Constants

C Code

```
// int is a 32-bit signed word
int a = -372;
int b = a + 6;
```

RISC-V assembly code

```
# s0 = a, s1 = b
addi s0, zero, -372
addi s1, s0, 6
```

- 12-bit signed constants (immediates) using `addi`
- Immediates are **sign-extended** from 12 to 32 bits
  - $+21_{10}$ → $0000\ 0001\ 0101_2$ → $00000000\ 00000000\ 00000000\ 00010101_2$
  - -21_{10} → $1111\ 1110\ 1011_2$ → $11111111\ 11111111\ 11111111\ 11101011_2$
- Any immediate that needs more than 12 bits **cannot** use this method.

Generating 32-bit Constants

C Code

```
int a = 0xFEDC8765;
```

RISC-V assembly code

```
# s0 = a
lui  s0, 0xFEDC8
addi s0, s0, 0x765
```

- Use load upper immediate (`lui`) and `addi`
- `lui`: puts an immediate in the upper 20 bits of destination register and 0's in lower 12 bits
- Remember that `addi` sign-extends its 12-bit immediate

Generating 32-bit Constants

C Code

```
int a = 0xFEDC8E65;
```

RISC-V assembly code

```
# s0 = 0xFEDC9000
lui  s0, 0xFEDC9

# s0 = 0xFEDC9000 + 0xFFFFFEAB
#    = 0xFEDC8EAB
addi s0, s0, -341
```

- **Note:** 0xEAB != 3755 because the bit 11 is used as sign 1 → **-341**
- Immediates are **sign-extended** from 12 to 32 bits
  - 0xEAB → `1110 1010 1011`→ `1111 1111 1111 1111 1111 1110 1010 1011`
- If bit 11 of 32-bit constant is **1**, increment upper 20 bits by **1** in `lui`

Logical / Shift Instructions

DDCA Ch6 - Part 6: Logical & Shift Instructions https://www.youtube.com/watch?v=TTK-RGTCYqE

Programming

- **High-level languages:**
  - e.g., C, Java, Python
  - Written at higher level of abstraction
- **High-level constructs:**
  - loops, conditional statements, arrays, function calls
- **First, introduce instructions that support these:**
  - Logical operations
  - Shift instructions
  - Multiplication & division
  - Branches & Jumps

Ada Lovelace, 1815-1852

- Wrote the first computer program
- Her program calculated the Bernoulli numbers on Charles Babbage's Analytical Engine
- She was the daughter of the poet Lord Byron

Logical Instructions

`and, or, xor`

- `and`: useful for masking bits
  - Masking all but the least significant byte of a value:

    `0xF234012F AND 0x000000FF = 0x0000002F`

- `or`: useful for combining bit fields
  - Combine 0xF2340000 with 0x000012BC:

    `0xF2340000 OR 0x000012BC = 0xF23412BC`

- `xor`: useful for inverting bits:

    `A XOR -1 = NOT A   (remember that -1 = 0xFFFFFFFF)`

Logical Instructions: Example 1

Source Registers

s1	0100 0110	1010 0001	1111 0001	1011 0111
s2	1111 1111	1111 1111	0000 0000	0000 0000

Assembly Code

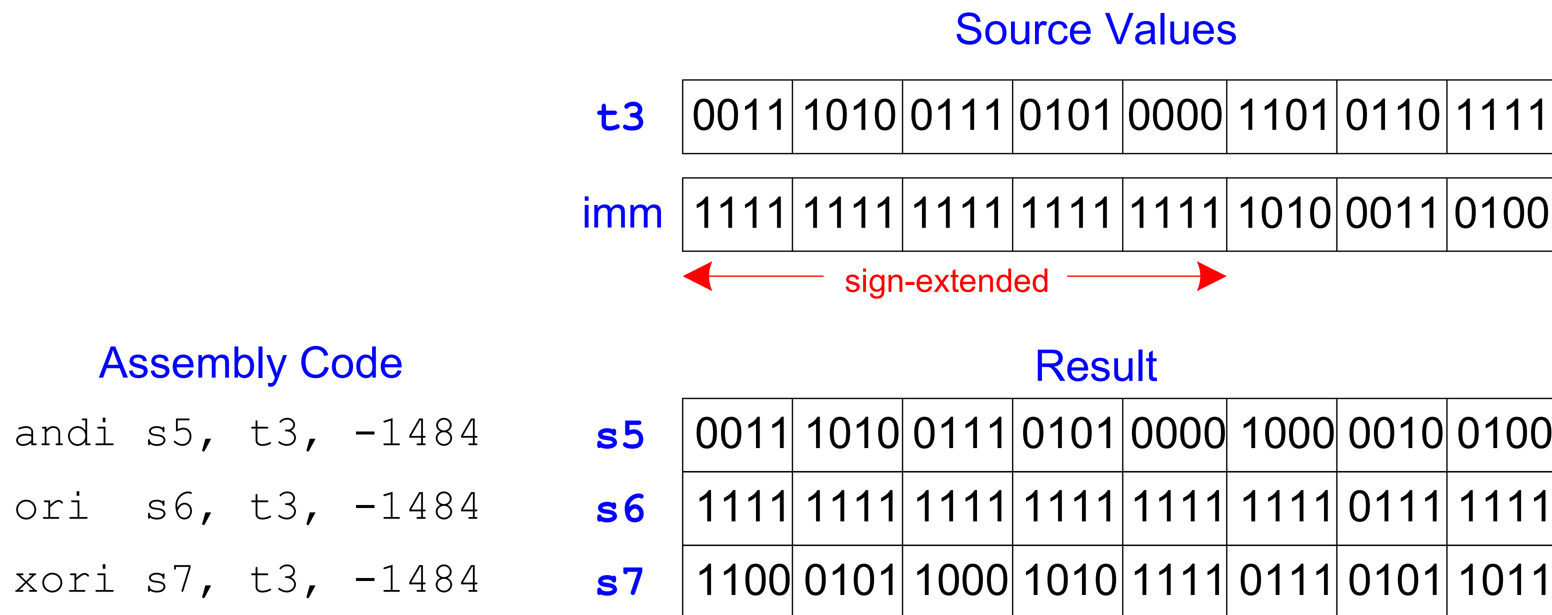
```
and s3, s1, s2
or  s4, s1, s2
xor s5, s1, s2
```

Result

s3	0100 0110	1010 0001	0000 0000	0000 0000
s4	1111 1111	1111 1111	1111 0001	1011 0111
s5	1011 1001	0101 1110	1111 0001	1011 0111

Logical Instructions: Example 2

- -1484 = **0xA34** in 12-bit 2's complement representation.



Shift Instructions

Shift amount is in (lower 5 bits of) a register

- `sll`: shift left logical
  - Example:

```
sll t0, t1, t2 # t0 = t1 << t2
```

- `srl`: shift right logical
  - Example:

```
srl t0, t1, t2 # t0 = t1 >> t2
```

- `sra`: shift right arithmetic
  - Example:

```
sra t0, t1, t2 # t0 = t1 >>> t2
```

Immediate Shift Instructions

Shift amount is an immediate between 0 to 31

- `slli`: shift left logical immediate
  - Example:

```
slli t0, t1, 23 # t0 = t1 << 23
```

- `srli`: shift right logical immediate
  - Example:

```
srli t0, t1, 18 # t0 = t1 >> 18
```

- `srai`: shift right arithmetic immediate
  - Example:

```
srai t0, t1, 5 # t0 = t1 >>> 5
```

Multiplication and Division

DDCA Ch6 - Part 7: Multiplication & Division Instructions https://www.youtube.com/watch?v=UDyVGVzdBuQ

Multiplication

32 × 32 multiplication → 64 bit result

- `mul s3, s1, s2`
  - s3 = lower 32 bits of result
- `mulh s4, s1, s2`
  - s4 = upper 32 bits of result, treats operands as signed
- `{s4, s3} = s1 x s2`

Example:

```
s1 = 0x40000000 = 230; s2 = 0x80000000 = -231
s1 x s2 = -261 = 0xE0000000 00000000
s4 = 0xE0000000; s3 = 0x00000000
```

Division

- 32-bit division → 32-bit quotient & remainder

```
div  s3, s1, s2  # s3 = s1/s2
rem  s4, s1, s2  # s4 = s1%s2
```

Example:

```
s1 = 0x00000011 = 17; s2 = 0x00000003 = 3
s1 / s2 = 5
s1 % s2 = 2
s3  = 0x00000005; s4 = 0x00000002
```

Branches & Jumps

DDCA Ch6 - Part 8: Branches & Jumps https://www.youtube.com/watch?v=8v-XAmqlZCo

Branching

- Execute instructions out of sequence
- Types of branches:
  - **Conditional**
    - branch if equal (`beq`)
    - branch if not equal (`bne`)
    - branch if less than (`blt`)
    - branch if greater than or equal (`bge`)
  - **Unconditional**
    - jump (`j`)
    - jump register (`jr`)
    - jump and link (`jal`)
    - jump and link register (`jalr`)

We'll talk about these when discuss function calls

Conditional Branching

 # RISC-V assembly

```
    addi s0, zero, 4     # s0 = 0 + 4 = 4
    addi s1, zero, 1     # s1 = 0 + 1 = 1
    slli s1, s1, 2       # s1 = 1 << 2 = 4
    beq  s0, s1, target  # branch is taken
    addi s1, s1, 1       # not executed
    sub  s1, s1, s0      # not executed

  target:                # label
    add  s1, s1, s0      # s1 = 4 + 4 = 8
```

Labels indicate instruction location.

- They can’t be reserved words and must be followed by a colon (:)

 # RISC-V assembly

```
    addi s0, zero, 4     # s0 = 0 + 4 = 4
    addi s1, zero, 1     # s1 = 0 + 1 = 1
    slli s1, s1, 2       # s1 = 1 << 2 = 4
    bne  s0, s1, target  # branch not taken
    addi s1, s1, 1       # s1 = 4 + 1 = 5
    sub  s1, s1, s0      # s1 = 5 – 4 = 1

  target:                # label
    add  s1, s1, s0      # s1 = 1 + 4 = 5
```

The code after the label is executed as well!

Conditional Branching

 # RISC-V assembly

```
 j         target          # jump to target
 srai      s1, s1, 2       # not executed
 addi      s1, s1, 1       # not executed
 sub       s1, s1, s0      # not executed

target:
 add      s1, s1, s0       # s1 = 1 + 4 = 5
```

Conditional Statements & Loops

DDCA Ch6 - Part 9: Conditional Statements & Loops https://www.youtube.com/watch?v=2txp2sSevW8&t

Conditional Statements & Loops

- Conditional Statements
  - `if` statements
  - `if/else` statements
- Loops
  - `while` loops
  - `for` loops

If Statement

C Code

```
if (i == j)
  f = g + h;

f = f - i;
```

RISC-V assembly code

```
# s0 = f, s1 = g, s2 = h
# s3 = i, s4 = j
    bne s3, s4, L1
    add s0, s1, s2

L1:
    sub s0, s0, s3
```

Assembly tests opposite case `(i != j)` of high-level code `(i == j)`

If/Else Statement

C Code

```
if (i == j)
  f = g + h;


else
  f = f - i;
```

RISC-V assembly code

```
# s0 = f, s1 = g, s2 = h
# s3 = i, s4 = j
    bne s3, s4, L1
    add s0, s1, s2
    j    done

L1:
    sub s0, s0, s3
```

While Loops

C Code

```
// determines the power
// of x such that 2^x = 128
int pow = 1;
int x   = 0;

while (pow != 128) {
  pow = pow * 2;
  x = x + 1;
}
```

RISC-V assembly code

```
# s0 = pow, s1 = x

       addi s0, zero, 1
       add  s1, zero, zero
       addi t0, zero, 128
while:
       beq  s0, t0, done
       slli s0, s0, 1
       addi s1, s1, 1
       j    while
done:
```

Assembly tests opposite case `(pow == 128)` of high-level code `( pow != 128)`

For Loops

Syntax:

```
for (initialization; condition; loop operation)
    statement
```

- **`initialization`**:
  executes before the loop begins
- **`condition`**:
  is tested at the beginning of each iteration
- **`loop operation`**:
  executes at the end of each iteration
- **`statement`**:
  executes each time the condition is met

For Loops

C Code

```c
// add the numbers from 0 to 9
int sum = 0;
int i;

for (i=0; i!=10; i = i+1) {
  sum = sum + i;
}
```

RISC-V assembly code

```
# s0 = i, s1 = sum
       addi s1, zero, 0
       add  s0, zero, zero
       addi t0, zero, 10
for:
       beq  s0, t0, done
       add  s1, s1, s0
       addi s0, s0, 1
       j    for
done:
```

Less Than Comparison

C Code

```
// add the powers of 2 from 1
// to 100
int sum = 0;
int i;


for (i=1; i < 101; i = i*2) {
  sum = sum + i;
}
```

RISC-V assembly code

```
# s0 = i, s1 = sum
       addi  s1, zero, 0
       addi  s0, zero, 1
       addi  t0, zero, 101

loop:
       bge   s0, t0, done
       add   s1, s1, s0
       slli  s0, s0, 1
       j     loop
done:
```

Less Than Comparison: Version 2

C Code

```
// add the powers of 2 from 1
// to 100
int sum = 0;
int i;


for (i=1; i < 101; i = i*2) {
  sum = sum + i;
}
```

slt: set if less than instruction

```
slt t2, s0, t0 # if s0 < t0, t2 = 1
               # otherwise   t2 = 0
```

RISC-V assembly code

```
# s0 = i, s1 = sum
       addi  s1, zero, 0
       addi  s0, zero, 1
       addi  t0, zero, 101
loop:
       slt   t2, s0, t0
       beq   t2, zero, done
       add   s1, s1, s0
       slli  s0, s0, 1
       j     loop
done:
```

Arrays

DDCA Ch6 - Part 10: Arrays

- https://www.youtube.com/watch?v=y76IlRRNaRg
- https://www.youtube.com/watch?v=XQDKFlPE_mo (Accessing Arrays of Characters)

Arrays

- Access large amounts of similar data
- **Index**: access each element
- **Size**: number of elements

Arrays

- 5-element array
- **Base address** = 0x123B4780 (address of first element, `array[0]`)
- First step in accessing an array: load base address into a register

Address	Data
123B4790	array[4]
123B478C	array[3]
123B4788	array[2]
123B4784	array[1]
123B4780	array[0]

Main Memory

C Code

```c
int array[5];
array[0] = array[0] * 2;
array[1] = array[1] * 2;
```

RISC-V assembly code

```
  # s0 = array base address
    lui   s0, 0x123B4           # 0x123B4 in upper 20 bits of s0
    addi  s0, s0, 0x780         # s0 = 0x123B4780

    lw    t1, 0(s0)             # t1 = array[0]
    slli  t1, t1, 1             # t1 = t1 * 2
    sw    t1, 0(s0)             # array[0] = t1

    lw    t1, 4(s0)             # t1 = array[1]
    slli  t1, t1, 1             # t1 = t1 * 2
    sw    t1, 4(s0)             # array[1] = t1
```

Address	Data
123B4790	array[4]
123B478C	array[3]
123B4788	array[2]
123B4784	array[1]
123B4780	array[0]

Main Memory

Accessing Arrays Using For Loops

C Code

```
int array[1000];
int i;
for (i=0; i < 1000; i = i + 1)
   array[i] = array[i] * 8;
```

RISC-V assembly code

```
# s0 = array base address, s1 = I
# initialization code
  lui  s0, 0x23B8F            # s0 = 0x23B8F000
  ori  s0, s0, 0x400          # s0 = 0x23B8F400
  addi s1, zero, 0            # i = 0
  addi t2, zero, 1000         # t2 = 1000

loop:
  bge  s1, t2, done           # if not then done
  slli t0, s1, 2              # t0 = i * 4 (byte offset)
  add  t0, t0, s0             # address of array[i]
  lw   t1, 0(t0)              # t1 = array[i]
  slli t1, t1, 3              # t1 = array[i] * 8
  sw   t1, 0(t0)              # array[i] = array[i] * 8
  addi s1, s1, 1              # i = i + 1
  j    loop                   # repeat
done:
```

Accessing Arrays Using For Loops (Optimized)

C Code

```
int array[1000];
int i;
for (i=0; i < 1000; i = i + 1)
   array[i] = array[i] * 8;
```

RISC-V assembly code

```
# s0 = array base address, s1 = I
# initialization code
  lui   s0, 0x23B8F          # s0 = 0x23B8F000
  ori   s0, s0, 0x400        # s0 = 0x23B8F400
  addi  s1, zero, 0          # i = 0
  addi  t2, zero, 4000       # t2 = 4000

loop:
  bge   s1, t2, done         # if not then done

  addi  s1, s1, 4            # address of array[i]
  lw    t1, 0(s1)            # t1 = array[i]
  slli  t1, t1, 3            # t1 = array[i] * 8
  sw    t1, 0(s1)            # array[i] = array[i] * 8

  j     loop                 # repeat
done:
```

ASCII Code

- **ASCII:** American Standard Code for Information Interchange
- Each text character has unique byte value
  - For example, `S = 0x53, a = 0x61, A = 0x41`
  - Lower-case and upper-case differ by `0x20 (32)`

#	Char	#	Char	#	Char	#	Char	#	Char	#	Char
20	space	**30**	0	**40**	@	**50**	P	**60**	`	**70**	p
21	!	**31**	1	**41**	A	**51**	Q	**61**	a	**71**	q
22	“	**32**	2	**42**	B	**52**	R	**62**	b	**72**	r
23	#	**33**	3	**43**	C	**53**	S	**63**	c	**73**	s
24	$	**34**	4	**44**	D	**54**	T	**64**	d	**74**	t
25	%	**35**	5	**45**	E	**55**	U	**65**	e	**75**	u
26	&	**36**	6	**46**	F	**56**	V	**66**	f	**76**	v
27	‘	**37**	7	**47**	G	**57**	W	**67**	g	**77**	w
28	(	**38**	8	**48**	H	**58**	X	**68**	h	**78**	x
29	)	**39**	9	**49**	I	**59**	Y	**69**	i	**79**	y
2A	*	**3A**	:	**4A**	J	**5A**	Z	**6A**	j	**7A**	z
2B	+	**3B**	;	**4B**	K	**5B**	[	**6B**	k	**7B**	{
2C	,	**3C**	<	**4C**	L	**5C**	\	**6C**	l	**7C**	|
2D	-	**3D**	=	**4D**	M	**5D**	]	**6D**	m	**7D**	}
2E	.	**3E**	>	**4E**	N	**5E**	^	**6E**	n	**7E**	~
2F	/	**3F**	?	**4F**	O	**5F**	_	**6F**	o		

#	Char	#	Char	#	Char	#	Char	#	Char	#	Char
20	space	**30**	0	**40**	@	**50**	P	**60**	`	**70**	p
21	!	**31**	1	**41**	A	**51**	Q	**61**	a	**71**	q
22	“	**32**	2	**42**	B	**52**	R	**62**	b	**72**	r
23	#	**33**	3	**43**	C	**53**	S	**63**	c	**73**	s
24	$	**34**	4	**44**	D	**54**	T	**64**	d	**74**	t
25	%	**35**	5	**45**	E	**55**	U	**65**	e	**75**	u
26	&	**36**	6	**46**	F	**56**	V	**66**	f	**76**	v
27	‘	**37**	7	**47**	G	**57**	W	**67**	g	**77**	w
28	(	**38**	8	**48**	H	**58**	X	**68**	h	**78**	x
29	)	**39**	9	**49**	I	**59**	Y	**69**	i	**79**	y
2A	*	**3A**	:	**4A**	J	**5A**	Z	**6A**	j	**7A**	z
2B	+	**3B**	;	**4B**	K	**5B**	[	**6B**	k	**7B**	{
2C	,	**3C**	<	**4C**	L	**5C**	\	**6C**	l	**7C**	|
2D	-	**3D**	=	**4D**	M	**5D**	]	**6D**	m	**7D**	}
2E	.	**3E**	>	**4E**	N	**5E**	^	**6E**	n	**7E**	~
2F	/	**3F**	?	**4F**	O	**5F**	_	**6F**	o		

Accessing Arrays of Characters

// C Code

```
char str[80] = "CAT";
int len = 0;

// compute length of string
while (str[len]) len++;
```

RISC-V assembly code

```
# s0 = array base address, s1 = len

        addi s1, zero, 0         # len = 0
while:  add t0, s0, s1           # address of str[len]
        lb t1, 0(t0)             # load str[len]   load byte!
        beq  t1, zero, done      # are we at the end of the string?
        addi s1, s1, 1           # len++
        j while                  # repeat while loop
done:
```

Function Calls

DDCA Ch6 - Part 11: Function Calls https://www.youtube.com/watch?v=XkC65EhgVmA

- **Caller:**
  calling function
  (in this case, `main`)
- **Callee:**
  called function
  (in this case, `sum`)

// C Code

```
void main()
{
  int y;
  y = sum(42, 7);
  ...
}

int sum(int a, int b)
{
  return (a + b);
}
```

Simple Function Call

C Code

```
int main() {
  simple();
  a = b + c;
}

void simple() {
  return;
}
```

RISC-V assembly code

```
0x00000300 main:   jal  simple      # call
0x00000304         add  s0, s1, s2
...               ...


0x0000051c simple: jr   ra           #
return
```

`void` means that simple doesn't return a value

`jal simple:`

`ra` = PC + 4 (0x00000304)

jumps to `simple` label (PC = 0x0000051c)

`jr ra:`

PC = `ra` (0x00000304)

Function Calling Conventions

- **Caller**:
  - passes **arguments** to callee
  - **jumps** to callee
- **Callee**:
  - **performs** the function
  - **returns** result to caller
  - **jumps** to point of call
  - **must** not overwrite registers or memory needed by caller

RISC-V Function Calling Conventions

- **Call Function**: jump and link (`jal func`)
- **Return** from function: jump register (`jr ra`)
- **Arguments**: `a0 - a7`
- **Return value**: `a0`

Name	Register Number	Usage
zero	x0	Constant value 0
ra	x1	Return address
sp	x2	Stack pointer
gp	x3	Global pointer
tp	x4	Thread pointer
t0-2	x5-7	Temporaries
s0/fp	x8	Saved register / Frame pointer
s1	x9	Saved register
a0-1	x10-11	Function arguments / return values
a2-7	x12-17	Function arguments
s2-11	x18-27	Saved registers
t3-6	x28-31	Temporaries

Input Arguments & Return Value

C Code

```
int main()
{
  int y;
  ...
  y = diffofsums(2, 3, 4, 5);  // 4 arguments
  ...
}
...
int diffofsums(int f, int g, int h, int i)
{
  int result;
  result = (f + g) - (h + i);
  return result;            // return value
}
```

RISC-V assembly code

```
# s7 = y
main:
. . .
addi a0, zero, 2   # argument 0 = 2
addi a1, zero, 3   # argument 1 = 3
addi a2, zero, 4   # argument 2 = 4
addi a3, zero, 5   # argument 3 = 5
jal  diffofsums    # call function
add  s7, a0, zero # y = returned value
. . .
# s3 = result
diffofsums:
add  t0, a0, a1    # t0 = f + g
add  t1, a2, a3    # t1 = h + i
sub  s3, t0, t1    # result=(f+g)-(h+i)
add  a0, s3, zero # put return value in a0
jr   ra            # return to caller
```

Input Arguments & Return Value

RISC-V assembly code

```
# s7 = y
main:
. . .
addi a0, zero, 2  # argument 0 = 2
addi a1, zero, 3  # argument 1 = 3
addi a2, zero, 4  # argument 2 = 4
addi a3, zero, 5  # argument 3 = 5
jal  diffofsums   # call function
add  s7, a0, zero # y = returned value
. . .
# s3 = result
diffofsums:
add  t0, a0, a1   # t0 = f + g
add  t1, a2, a3   # t1 = h + i
sub  s3, t0, t1   # result=(f+g)-(h+i)
add  a0, s3, zero # return value in a0
jr   ra           # return to caller
```

What's the issue?

- `diffofsums`
  overwrote 3 registers: t0, t1, s3
- `diffofsums`
  can use **stack** to temporarily store registers

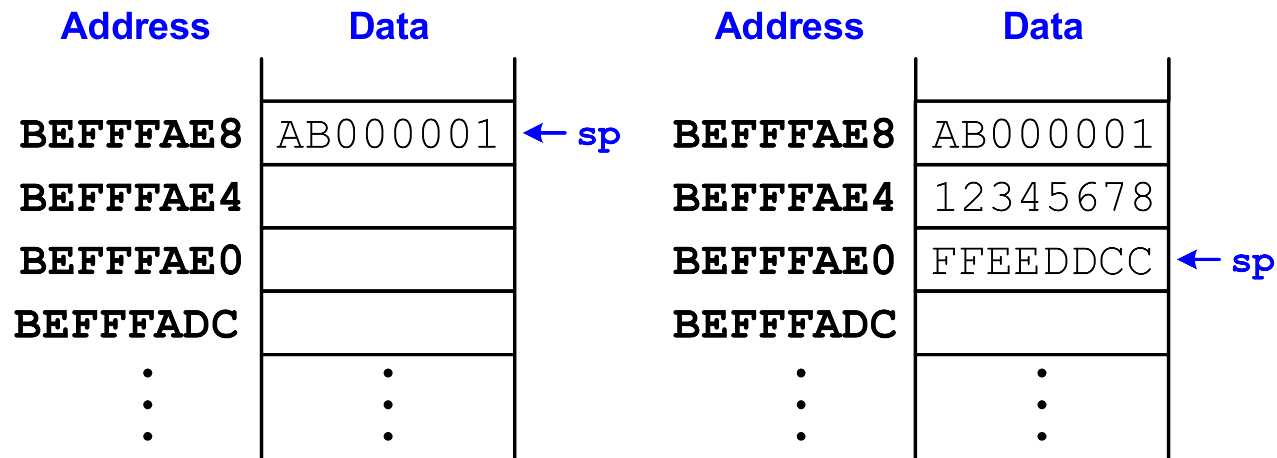
The Stack

DDCA Ch6 - Part 12: The Stack https://www.youtube.com/watch?v=eHerSQTFsMA

The Stack

- Memory used to temporarily save variables
- Like stack of dishes, last-in-first-out (LIFO) queue
- **Expands**: uses more memory when more space needed
- **Contracts**: uses less memory when the space is no longer needed

- Grows down (from higher to lower memory addresses)
- Stack pointer: `sp` points to top of the stack
- The stack is more like a **attic.**
  - You fill it from the attic down to the basement and cellar
- Example:

  Make room on stack for **2 words.**
- 

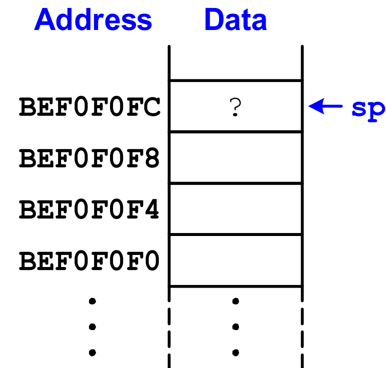
How Functions use the Stack

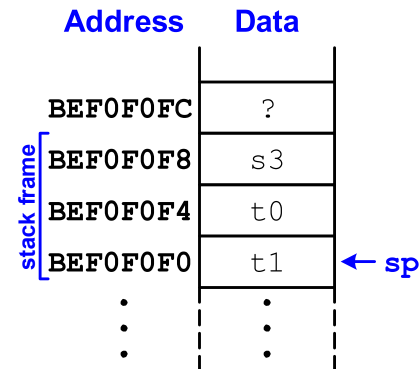
- Called functions must have no unintended side effects
- But `diffofsums` overwrites 3 registers: `t0, t1, s3`

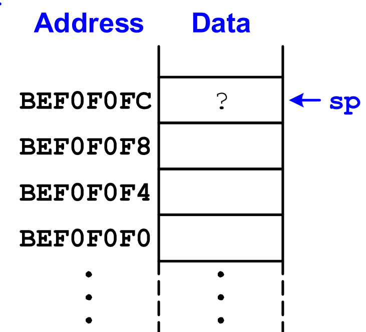
```
# RISC-V assembly
  • # s3 = result
  • diffofsums:
  •   add  t0, a0, a1   # t0 = f + g
  •   add  t1, a2, a3   # t1 = h + i
  •   sub  s3, t0, t1   # result = (f + g) - (h + i)
  •   add  a0, s3, zero # put return value in a0
  •   jr   ra           # return to caller
```

Storing Register Values on the Stack

```
# s3 = result
diffofsums:
  addi sp, sp, -12      # make space on stack to
                        # store three registers
  sw   s3, 8(sp)        # save s3 on stack
  sw   t0, 4(sp)        # save t0 on stack
  sw   t1, 0(sp)        # save t1 on stack
  add  t0, a0, a1       # t0 = f + g
  add  t1, a2, a3       # t1 = h + i
  sub  s3, t0, t1       # result = (f + g) - (h + i)
  add  a0, s3, zero     # put return value in a0
  lw   s3, 8(sp)        # restore s3 from stack
  lw   t0, 4(sp)        # restore t0 from stack
  lw   t1, 0(sp)        # restore t1 from stack
  addi sp, sp, 12       # deallocate stack space
  jr   ra               # return to caller
```







There is a convention about who puts what on the stack.

`This examples does not follow the convention`

Preserved Registers

Preserved ***Callee-Saved***	**Nonpreserved** ***Caller-Saved***
`s0-s11`	`t0-t6`
`sp`	`a0-a7`
`ra`	
stack above `sp`	stack below `sp`

Storing Register Values on the Stack following the convention

```
# s3 = result
diffofsums:
  addi sp, sp, -4        # make space on stack to
                          # store three registers
  sw   s3, 8(sp)          # save s3 on stack
  add  t0, a0, a1         # t0 = f + g
  add  t1, a2, a3         # t1 = h + i
  sub  s3, t0, t1         # result = (f + g) - (h + i)
  add  a0, s3, zero       # put return value in a0
  lw   s3, 8(sp)          # restore s3 from stack
  addi sp, sp, 12         # deallocate stack space
  jr   ra                 # return to caller
```

Optimized `diffofsums`

```
# a0 = result
diffofsums:
  add  t0, a0, a1        # t0 = f + g
  add  t1, a2, a3        # t1 = h + i
  sub  a0, t0, t1        # result = (f + g) - (h + i)
  jr   ra                # return to caller
```

- Stack not needed
  s3 is not used

Non-Leaf Function Calls

- **Non-leaf function:**
  a function that calls another function

Register `ra` must be **preserve** before function call and **restored** after.

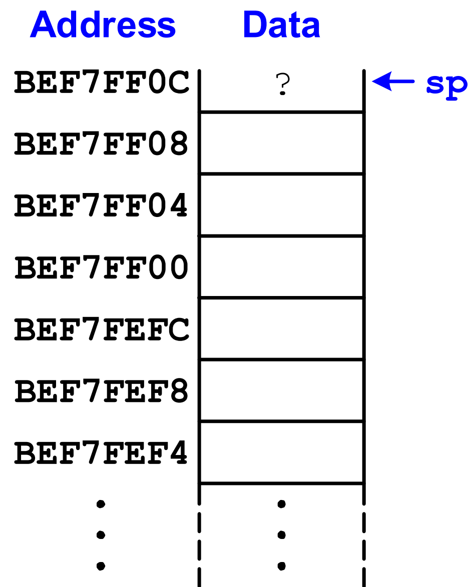
```
func1:
  addi sp, sp, -4    # make space on stack
  sw   ra, 0(sp)     # save ra on stack
  jal  func2
  ...
  lw   ra, 0(sp)     # restore ra from stack
  addi sp, sp, 4     # deallocate stack space
  jr   ra            # return to caller
```

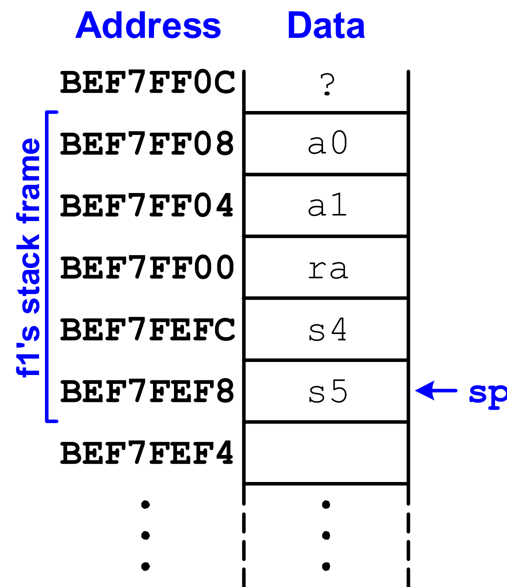
Non-Leaf Function Call Example

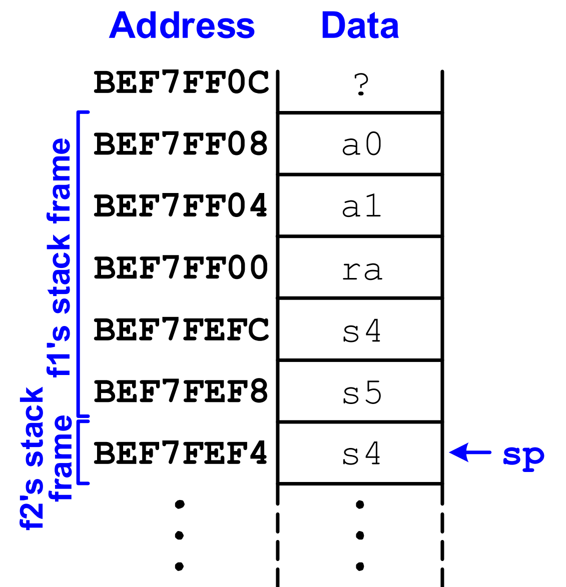
```
# f1 (non-leaf function) uses s4-s5 and needs a0-a1 after call to f2
f1:
  addi sp, sp, -20   # make space on stack for 5 words
  sw   a0, 16(sp)
  sw   a1, 12(sp)
  sw   ra, 8(sp)     # save ra on stack
  sw   s4, 4(sp)
  sw   s5, 0(sp)
  jal  func2
  ...
  lw   ra, 8(sp)     # restore ra (and other regs) from stack
  ...
  addi sp, sp, 20    # deallocate stack space
  jr   ra            # return to caller

# f2 (leaf function) only uses s4 and calls no functions
f2:
  addi sp, sp, -4    # make space on stack for 1 word
  sw   s4, 0(sp)
  ...
  lw   s4, 0(sp)
  addi sp, sp, 4     # deallocate stack space
  jr   ra            # return to caller
```

Stack during Function Calls

```
# f1 (non-leaf function)
f1:
  addi sp, sp, -20
  sw   a0, 16(sp)
  sw   a1, 12(sp)
  sw   ra, 8(sp)
  sw   s4, 4(sp)
  sw   s5, 0(sp)
  jal  func2
  ...
  lw   ra, 8(sp)
  ...
  addi sp, sp, 20
  jr   ra

# f2 (leaf function)
f2:
  addi sp, sp, -4
  sw   s4, 0(sp)
  ...
  lw   s4, 0(sp)
  addi sp, sp, 4
  jr   ra
```







Function Call Summary

- **Caller**
  - Save any needed registers (`ra`, maybe `t0-t6/a0-a7`)
  - Put arguments in `a0-a7`
  - Call function: `jal callee`
  - Look for result in `a0`
  - Restore any saved registers

- **Callee**
  - Save registers that might be disturbed (`s0-s11`)
  - Perform function
  - Put result in `a0`
  - Restore registers
  - Return: `jr ra`

Recursive Functions

Recursive Function Example

- Function that **calls itself**
- When converting to assembly code:
  - In the first pass, treat recursive calls as if it's calling a different function and ignore overwritten registers.
  - Then save/restore registers on stack as needed.

- Example:

  Factorial function:

$$factorial(n) = n! = n * (n-1) * (n-2) * (n-3) \ldots * 1$$

$$factorial(6) = 6! = 6 * 5 * 4 * 3 * 2 * 1 = 720$$

Recursive Function Example

- **High-Level Code**

```
int factorial(int n) {
  if (n <= 1)
    return 1;
  else
    return (n*factorial(n-1));
}
```

- **Example: n = 3**

```
factorial(3): returns 3*factorial(2)
factorial(2): returns 2*factorial(1)
factorial(1): returns 1
```

Thus,

```
factorial(1): returns 1
factorial(2): returns 2*1 = 2
factorial(3): returns 3*2 = 6
```

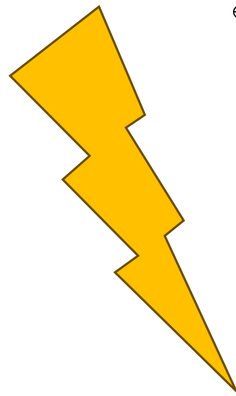
Recursive Function Example: Issue

High-Level Code

```
int factorial(int n) {




  if (n <= 1)
    return 1;


  else
    return (n*factorial(n-1));
}
```

RISC-V Assembly

```
factorial:


  # a0 holds the argument
  addi t0, zero, 1   # temporary = 1
  bgt  a0, t0, else  # if n>1, go to else
  addi a0, zero, 1   # otherwise, return 1

  jr   ra            # return
else:
  addi a0, a0, -1    # n = n - 1
  jal  factorial     # recursive call

  # a0 is now overwritten by the function call
  # Must save it and ra on stack before function call.
  mul  a0, a0, a0    # a0=n*factorial(n-1)
  jr   ra            # return
```



Recursive Function Example

High-Level Code

```
int factorial(int n) {




  if (n <= 1)
    return 1;


  else
    return (n*factorial(n-1));
}
```

Note: `n` is restored from stack into `t1` so it doesn't overwrite return value in `a0`.

RISC-V Assembly

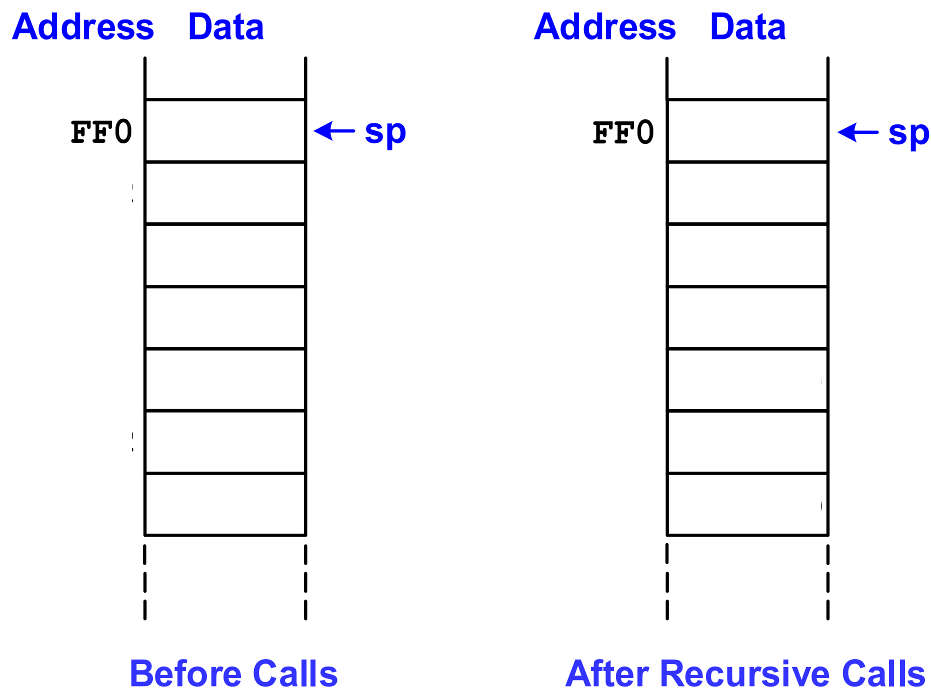
```
factorial:
  addi sp, sp, -8    # save regs
  sw   a0, 4(sp)
  sw   ra, 0(sp)
  addi t0, zero, 1   # temporary = 1
  bgt  a0, t0, else  # if n>1, go to else
  addi a0, zero, 1   # otherwise, return 1
  addi sp, sp, 8     # restore sp
  jr   ra            # return
else:
  addi a0, a0, -1    # n = n - 1
  jal  factorial     # recursive call
  lw   t1, 4(sp)     # restore n into t1
  lw   ra, 0(sp)     # restore ra
  addi sp, sp, 8     # restore sp
  mul  a0, t1, a0    # a0=n*factorial(n-1)
  jr   ra            # return
```

Recursive Functions

```
0x8500 factorial: addi sp, sp, -8     # save registers
0x8504            sw   a0, 4(sp)
0x8508            sw   ra, 0(sp)
0x850C            addi t0, zero, 1    # temporary = 1
0x8510            bgt  a0, t0, else   # if n > 1, go to else
0x8514            addi a0, zero, 1    # otherwise, return 1
0x8518            addi sp, sp, 8      # restore sp
0x851C            jr   ra             # return
0x8520 else:      addi a0, a0, -1     # n = n - 1
0x8524            jal  factorial      # recursive call
0x8528            lw   t1, 4(sp)      # restore n into t1
0x852C            lw   ra, 0(sp)      # restore ra
0x8530            addi sp, sp, 8      # restore sp
0x8534            mul  a0, t1, a0     # a0 = n*factorial(n-1)
0x8538            jr   ra             # return
```

PC+4 = 0x8528 when factorial is called recursively.

When **factorial(3)** is called:




```
0x8500 factorial: addi sp, sp, -8
0x8504            sw   a0, 4(sp)
0x8508            sw   ra, 0(sp)
0x850C            addi t0, zero, 1
0x8510            bgt  a0, t0, else
0x8514            addi a0, zero, 1
0x8518            addi sp, sp, 8
0x851C            jr   ra
0x8520 else:      addi a0, a0, -1
0x8524            jal  factorial
0x8528            lw   t1, 4(sp)
0x852C            lw   ra, 0(sp)
0x8530            addi sp, sp, 8
0x8534            mul  a0, t1, a0
0x8538            jr   ra
```

More on Jumps & Pseudoinstructions

DDCA Ch6 - Part 14: More Jumps & PseudoInstructions https://www.youtube.com/watch?v=lFpJPwTfWII

Jump

- RISC-V has two types of unconditional jumps
  - Jump and link (`jal rd,` $imm_{20:0}$)
    - `rd = PC+4`
    - `PC = PC + imm`
  - jump and link register (`jalr rd, rs,` $imm_{11:0}$)
    - `rd = PC+4`
    - `PC = [rs] + SignExt(imm)`

Pseudoinstructions

- **Pseudoinstructions** are not actual RISC-V instructions but they are often more convenient for the programmer.
- Assembler converts them to real RISC-V instructions.
- Like:

Jump Pseudoinstructions

- RISC-V has four jump psuedoinstructions
  - Jump
    `j    imm` → `jal  x0, imm`
  - Jump and link
    `jal imm` → `jal  ra, imm`
  - Jump register
    `jr  rs` → `jalr x0, rs, 0`
  - Return
    `ret` → `jr   ra` → `jalr x0, ra, 0`

Labels

- Label indicates where to jump
- Represented in jump as immediate offset
  - `imm` = # bytes past jump instruction
  - In example

```
jal simple = jal ra, 0x21C



?? 0x21C ??

imm = (0x51C-0x300) = 0x21C
```

RISC-V assembly code

```
0x00000300 main:    jal   simple         # call
0x00000304          add   s0, s1, s1
...                 ...


0x0000051c simple: jr    ra             #
return
```

Long Jumps

- The immediate is limited in size
  - 20 bits for `jal`, 12 bits for `jalr`
  - Limits how far a program can jump
- Special instruction to help jumping further
  - `auipc rd, imm` # add upper immediate to PC

    `rd = PC + {imm`$_{31:12}$`, 12'b0}`
- Pseudoinstruction: call `imm`$_{31:0}$
  - Behaves like `jal imm`, but allows 32-bit immediate offset

```
auipc ra, imm31:12
jalr ra, ra, imm11:0
```

More RISC-V Pseudoinstructions

Pseudoinstruction	RISC-V Instructions
`j label`	`jal zero, label`
`jr ra`	`jalr zero, ra, 0`
`mv t5, s3`	`addi t5, s3, 0`
`not s7, t2`	`xori s7, t2, -1`
`nop`	`addi zero, zero, 0`
`li s8, 0x56789DEF`	`lui s8, 0x5678A` `addi s8, s8, 0xDEF`
`bgt s1, t3, L3`	`blt t3, s1, L3`
`bgez t2, L7`	`bge t2, zero, L7`
`call L1`	`auipc ra,` $imm_{31:12}$ `jalr ra, ra,` $imm_{11:0}$
`ret`	`jalr zero, ra, 0`

See Appendix B (Harris & Harris) for more pseudoinstructions.

Machine Language

DDCA Ch6 - Part 15: Machine Language https://www.youtube.com/watch?v=oUvjjLeEB2Y