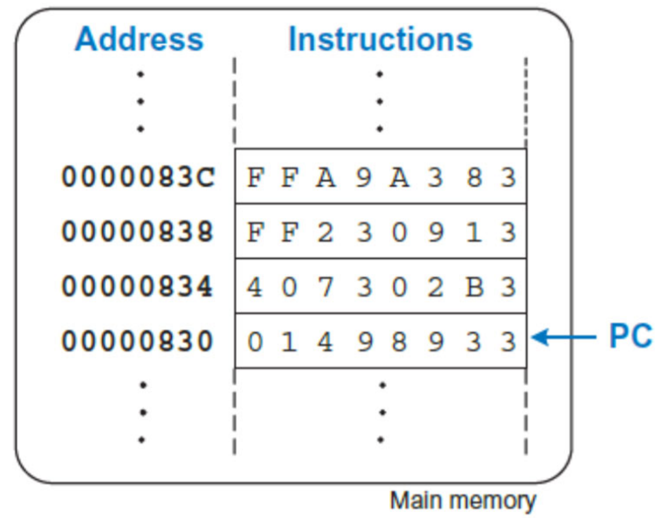
RISC-V ISA (continue) and Assembly Programming

Acknowledgment: Slides are adapted from Harris and Harris textbook instructor's material

The Power of the Stored Program

- 32-bit instructions & data stored in memory
- Sequence of instructions: Only difference between two applications
- To run a new program:
  - No rewiring required
  - Simply store new program in memory
- Program Execution:
  - Processor *fetches* (reads) instructions from memory in sequence
  - Processor performs the specified operation

Assembly code	Machine code
`add s2, s3, s4`	0x01498933
`sub t0, t1, t2`	0x407302B3
`addi s2, t1, -14`	0xFF230913
`lw t2, -6(s3)`	0xFFA9A383



Program Counter (PC): keeps track of current instruction

Generating Constants

- 12-bit constants using `addi`:

C Code

```
// int is a 32-bit signed word
int a = 0xf3c;
```

RISC-V assembly code

```
# s0 = a
addi s0, zero, 0xf3c
```

- 32-bit constants using load upper immediate (`lui`) and `addi`:

C Code

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

RISC-V assembly code

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

Corner case when top bit is set to 1 in lower 12 bits due to sign extension (to be discussed later)

Pseudo Instructions

Pseudoinstruction	RISC-V Instructions	Description	Operation
`j label`	`jal zero, label`	jump	`PC = label`
`jr ra`	`jalr zero, ra, 0`	jump register	`PC = ra`
`mv t5, s3`	`addi t5, s3, 0`	move	`t5 = t3`
`not s7, t2`	`xori s7, t2, -1`	one's complement	`s7 = ~t2`
`nop`	`addi zero, zero, 0`	no operation	
`li s8, 0x7EF`	`addi s8, zero, 0x7EF`	load 12-bit immediate	`s8 = 0x7EF`
`li s8, 0x56789DEF`	`lui s8, 0x5678A` `addi s8, s8, 0xDEF`	load 32-bit immediate	`s8 = 0x56789DEF`
`bgt s1, t3, L3`	`blt t3, s1, L3`	branch if >	`if (s1 > t3), PC = L3`
`bgez t2, L7`	`bge t2, zero, L7`	branch if ≥ 0	`if (t2 ≥ 0), PC = L7`
`call L1`	`jal L1`	call nearby function	`PC = L1, ra = PC + 4`
`call L5`	`auipc ra,` $imm_{31:12}$ `jalr ra, ra,` $imm_{11:0}$	call far away function	`PC = L5, ra = PC + 4`
`ret`	`jalr zero, ra, 0`	return from function	`PC = ra`

High-Level Code Constructs

- `if` statements
- `if/else` statements
- `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
done:
```

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 for the opposite case (`pow == 128`) of the C code (`pow != 128`).

For Loops

C Code

```
// 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, zero
       add  s0, zero, zero
       addi t0, zero, 10
for:   bgeu s0, t0, done
       add  s1, s1, s0
       addi s0, s0, 1
       j    for
done:
```

Note that this code use bge instead of blt to reduce the dynamic instruction count. Can we modify the original C code to reduce dynamic instruction count further?

<u>Dynamic instruction count</u> is the number of executed assembly instructions (proxy of a time to run the code)
3 + 4x9 + 1 = 40 instructions for the code above
<u>Static instruction count</u> is the number of assembly instructions in the code (proxy of a space needed to store the code in memory)
7 instructions for the code above (pseudo instruction j is converted to jal)

Lesser Dynamic Instruction Count Code

Original C Code

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

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

Similar Functionality C Code

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

Do {
  sum = sum + i;
  i= i + 1;
}while (i<10)
```

RISC-V assembly code

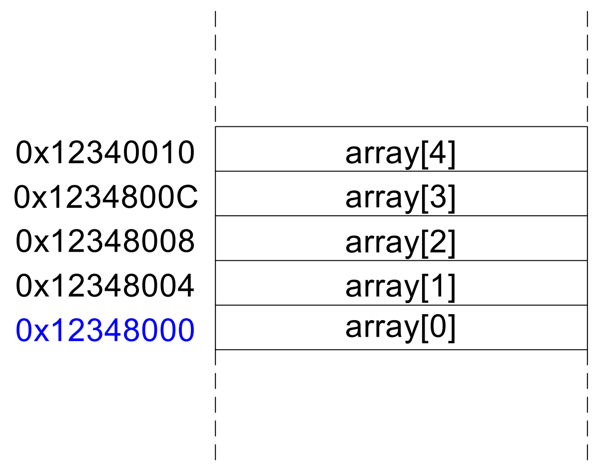
```
# s0 = i, s1 = sum
        addi s1, zero, zero
        add  s0, zero, zero
        addi t0, zero, 10
loop:   add  s1, s1, s0
        addi s0, s0, 1
        bltu s0, t0, loop
done:
```

3 + 3x9 = 30 instructions for the new code

May need to modify a prolog code to check the condition before entering the body when converting from “for” to “do-while” loop

Arrays

- 5-element array
- **Base address** = 0x12348000 (address of first element, `array[0]`)
- First step in accessing an array: load base address into a register



Accessing Arrays

```
// C Code
  int array[5];
  array[0] = array[0] * 2;
  array[1] = array[1] * 2;

# RISC-V assembly code
# s0 = array base address
  lui  s0, 0x12348              # 0x12348 in upper half of s0

  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
```

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, 0x23B80          # s0 = 0x23B80000 (given)
  addi s1, zero, 0           # i = 0
  addi t2, zero, 1000        # t2 = 1000

loop:
  bltu s1, t2, skip
  j done
skip:
  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:
```

Dynamic instruction count = 3 + 9x1000 + 2 = 9005. Can we do better?

Function Calls

- **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);
}
```

Function Conventions

- **Caller:**
  - passes **arguments** to callee
  - jumps to callee
- **Callee:**
  - **performs** the function
  - **returns** result to caller
  - **returns** to point of call

  - **must not interfere** with the behavior of the caller, i.e., it must know where to return to after it completes and it must not trample on any registers or memory needed by the caller

RISC-V Function Conventions

- **Call Function:** jump and link (`jal`)
- **Return** from function: jump register (`jalr`, same as `jr` pseudo)
- **Arguments**: up to 8 registers can be passed in `a0 – a7`
- **Return value**: `a0`

Function Calls

C Code

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

void simple() {
  return;
}
```

RISC-V assembly code

```
0x00400200 main:   jal  ra, simple
0x00400204         add  s0, s1, s2
...

0x00401020 simple: jr ra
```

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

`jal ra`: jumps to `simple`

`ra` stores address of the next instruction after jal

`jr ra`: jumps to address stores in `ra`

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

```
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  ra, diffofsums  # call Function
  add  a0, zero, zero  # y = returned value
  ...

# s0 = result
diffofsums:
  add s0, a0, a1     # s0 = f + g
  add s1, a2, a3     # s1 = h + i
  sub a0, s0, s1     # result = (f + g) - (h + i)
  jr  ra             # return to caller
```

What if caller needs s0, s1 which are modified by the callee in this example?
What if there are more registers to pass to or return from function?

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

The Stack

- Grows down (from higher to lower memory addresses)
- Stack pointer: `sp` points to top of the stack

Address	Data	
7FFFFFFC	12345678	← sp
7FFFFFF8		
7FFFFFF4		
7FFFFFF0		
⋮	⋮	

Address	Data	
7FFFFFFC	12345678	
7FFFFFF8	AABBCCDD	
7FFFFFF4	11223344	← sp
7FFFFFF0		
⋮	⋮	

Storing Register Values on the Stack

```
diffofsums:
  addi sp, sp, -8      # make space on stack
                       # to store 2 registers
  sw   s0, 4(sp)       # save t0 on stack
  sw   s1, 0(sp)       # save t1 on stack
  add  s0,  a0, a1     # s0 = f + g
  add  s1,  a2, a3     # s1 = h + i
  sub  a0,  s0, s1     # result = (f + g) - (h + i)
  lw   s1, 0(sp)       # restore t1 from stack
  lw   s0, 4(sp)       # restore t0 from stack
  addi sp, sp, 8       # deallocate stack space
  jr   ra              # return to caller
```

- Extra instructions for saving registers to and restoring registers from stack increase dynamic count
- A more efficient way is to agree on what registers can be scrambled by the callee and which ones should be saved

Rules on Preserving Memory and Registers

Preserved (*callee*-saved)	**Nonpreserved (*caller*-saved)**
Saved registers: `s0-s11`	Temporary registers: `t0-t6`
Return address: `ra`	Argument registers: `a0-a7`
Stack pointer: `sp` (as well as gp, tp)	
Stack above the stack pointer	Stack below the stack pointer

Revisiting Diffofsumm Example

```
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
```

The callee can scramble t0 and t1 according the agreement with the caller so no need to save them

Nested Function Calls

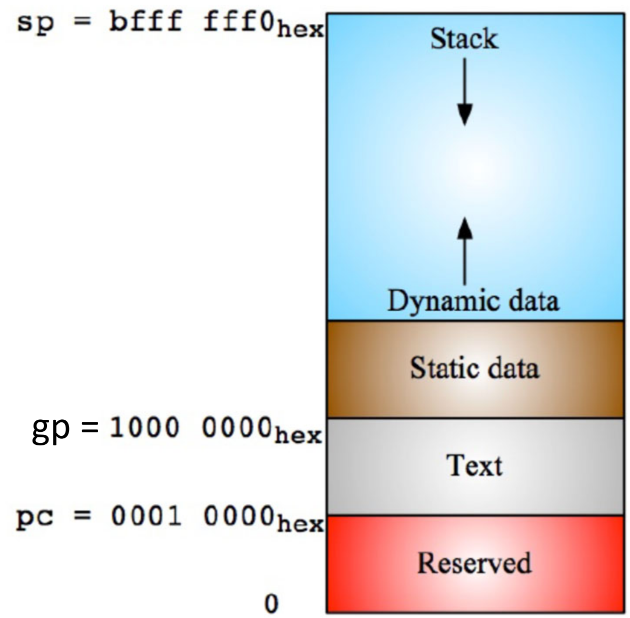
```
proc1:
  addi sp, sp, -4      # make space on stack
  sw   ra, 0(sp)       # save ra on stack
  jal  ra, proc2
  ...
  lw   ra,  0(sp)     # restore ra from stack
  addi sp, sp, 4      # deallocate stack space
  jr  ra              # return to caller
```

Need to save return address (typically to register ra); otherwise it will be lost (replaced with new when performing jal)

Example of Memory Layout
(specific to the studied baseline RV32)

Three important memory areas allocated when running a C program:

- Static: Variables declared once per program, cease to exist only after execution completes (e.g., C globals)
- Stack: Space to the used by procedure during execution; this is very local C variables are stored and where registered can be "spilled". Starts are high memory and grows down.
- Heap: Variables declared dynamically via malloc. Grows up.



RISC-V Register Set Revisited

Name	Register Number	Use
`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`	Temporary registers
`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`	Temporary registers

- Registers used for specific purposes:
  - `x0` 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
  - `a0–7` registers are used to pass to or return values from a function
  - the stack pointer, `sp`, is a pointer to the top (used value) of the stack
  - `ra` holds the return address from the function
  - the frame pointer, `fp` points to the base of the stack frame. It does not change during function execution and is used by a function to access stack when `sp` is changing dynamically
  - `gp` is a pointer to the base of "static" data segment

Register use rules (other than x0) only matters if a program needs to communicate to other programs (O/S etc.)

C Structures (background material)

- A `struct` is a data structure composed from simpler data types.
  - Like a class in Java/C++ but without methods or inheritance.

```
struct point {   /* type definition */
    int x;
    int y;
};
```

- The C arrow operator (`->`) dereferences and extracts a structure field with a single operator.

```
struct point *p;
 /* code to assign to pointer */
printf("x is %d\n", (*p).x);
printf("x is %d\n", p->x);
```

How big are structs? (background material)

- Recall C operator `sizeof()` which gives size in bytes (of type or variable)
- How big is `sizeof(p)`?

```
struct p {
    double x;
    int y;
};
```

- Compiler may word align integer `y`, when having data types < 4 bytes

Linked List (background material)

- A specific data structure of struct elements (nodes) with the specified order of nodes
- In the simplest singly-linked list, a node consists of data and link (pointer) to the next node of a list
- Need to know pointer to the head of the list to handle singly-linked list
- Doubly-linked list node keeps an additional pointer to the previous node of the list

Array vs. Linked list

- Fixed or slowly changing size & order
- Contiguous space in memory (could be allocated dynamically or statically)
- Fast traversal / no memory overhead but fixed structure

- Dynamically changing size, order
- Could be contiguous (when all elements are statically allocated) but most often not when allocated dynamically
- Typically slower traversal / additional memory for storing pointers but flexible structure

Deleting From Doubly Linked List Example

Consider the following C structure:

```
struct mylist {
        double value;
        struct mylist *next;
        struct mylist *prev;
}
```

(a) *40 points* Assuming that **cur** is a pointer to some node in a circular doubly linked list (with more than 1 node) convert the following C procedure to the ~~MIPS~~ RISC-V assembly one. Assume that the value of **cur** is passed in register **$a0**.

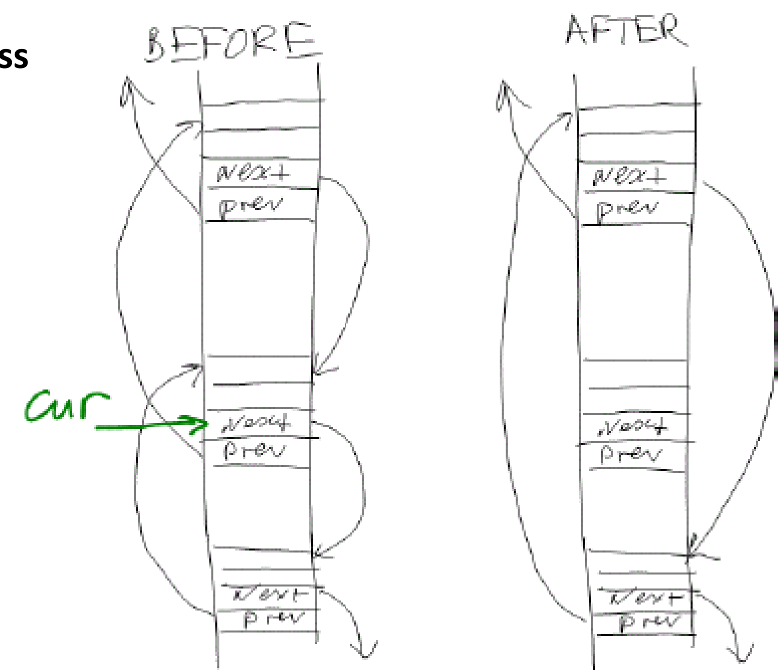
```
void  myfunc (struct mylist *cur) {
        cur → prev → next  =  cur → next;
        cur → next → prev  =  cur → prev;
        return;
}
```

(b) *10 points* Briefly describe in English what the code does

The function deletes element **cur** from the doubly linked list by redirecting **next** and **prev** pointers from the previous element and the next element, respectively.

Note that the memory addresses are growing down in this figure

Low address



High address

Deleting From Doubly Linked List Example

```
struct mylist {
        double value;
        struct mylist *next;
        struct mylist *prev;
}
```

(a) *40 points* Assuming that **cur** is a pointer to some node in a circular doubly linked list (with more than 1 node) convert the following C procedure to the MIPS assembly one. Assume that the value of **cur** is passed in register **$a0**.

```
void  myfunc (struct mylist *cur)  {
        cur → prev → next  =  cur → next;
        cur → next → prev  =  cur → prev;
        return;
}
```

```
myfunct:
    lw    $t0, 8($a0)        # load to $t0 pointer cur->next
    lw    $t1, 12($a0)       # load to $t1 pointer cur->prev
    sw    $t0, 8($t1)        # cur → prev → next  =  cur → next   (i.e. $t0);
    sw    $t1, 12($t0)       # cur → next → prev  =  cur → prev   (i.e. $t1);
    jr    $ra                # jump out of the procedure myfunct
```

How to Compile & Run a Program