EECS 388:
Embedded Systems, Spring 2023
Lecture 7
Instructions: Language of the Computer

Lecture 4

Instructor: Tamzidul Hoque, Assistant Professor, Dept. of EECS, University of Kansas (hoque@ku.edu), office: Eaton 2038

Credits: Some slides are adopted from Henk Corporaal's Computer Architecture and Organization course

Procedures

- A function commonly used in high level programs like C or Java
  - Makes code readable
  - Allows code reuse

Caller function

```
int main () {
    int x, y;
    int a = 100;
    int b = 250;
    x=gcd(a, b);
    x=y+1;
    Y= gcd(x, a);
    return 0; }
```

Callee function

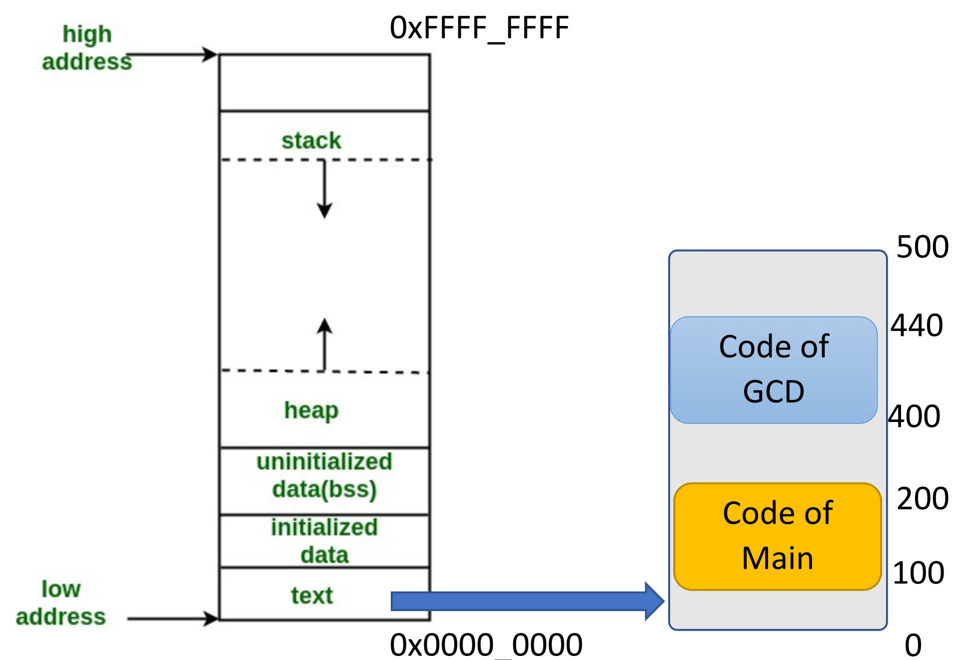
```
int gcd(int m, int n) {
      int x = m;
      int y = n;
      while(x != y) {
            if(x > y) {
                  x = x - y;
            } else{
                  y = y - x;
            }
      }
      return x;
}
```

Memory Map of a Running Program

A typical memory representation of a C program consists of:

1. Static
   - -Text/code segment
   - -Initialized data segment
   - -Uninitialized data segment

2. Stack
3. Heap



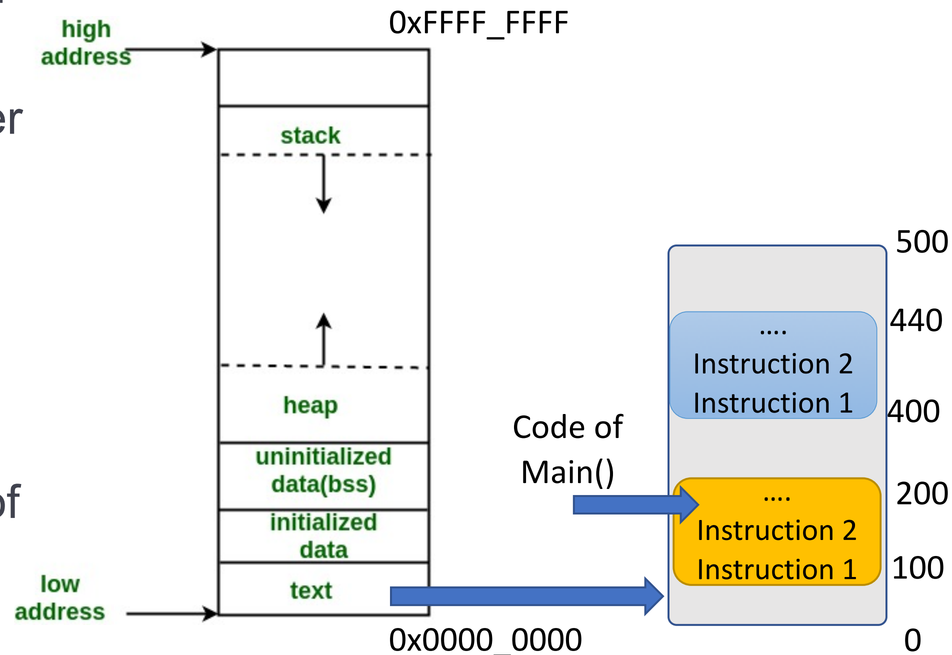
Memory Map of a Running Program

Program Counter (PC):

- A register that holds the address of the instruction to be executed
- Usually, the PC is incremented after fetching an instruction

Example:

- Main() starts from address 100
- PC=100
- When instruction 1 is fetched, PC=PC+4=104, which is address of instruction 2



Caller and Callee

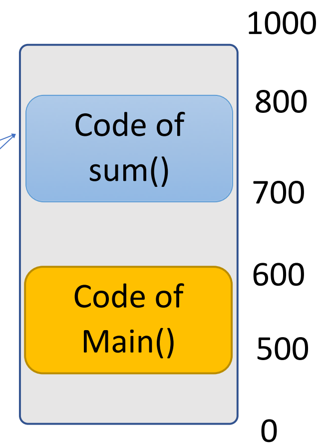
- Lets assume that main() is a caller program and sum() is a callee
- A caller needs to pass arguments to the called procedure, as well as get results back from the called procedure
- A caller uses the same register set as the called procedure

```
void main(){
        m=1;
        n=2;
        o=sum(m,n);
        p=m+1;
}

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

Calling a procedure: Return address

- When a callee is executed, the flow of execution jumps to a different segment of the memory
  - Performs jump instruction to a memory address
- Main() resides in address 500
- Sum() resides in address 700
- **Issues:**
  - If we don't return to main() after sum() ends, the line p=m+1 will not execute
  - How can we return execution to main()?

```
void main(){
        m=1;
        n=2;
        o=sum(m,n);
        p=m+1;
        q=sum(p,n);
        r=n+1;
}

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



Jump and link

- Use jump and link instruction (jal):
  - Written as: jal Procedure_Address
- Link means that the link (address) to go back to the caller is preserved
  - In return address register $ra
  - Now we can go back to main() after execution of sum() completes
- What exactly is being stored in $ra ?
  - The address of next instruction, from where the procedure was called
  - $ra=500+4
  - PC becomes the address of sum()

```
void main(){
        m=1;
        n=2;
        o=sum(m,n); ---Address 500
        p=m+1;
}

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

How to return from the procedure?

- We have the return address
  - Use jump register instruction: jr $ra
  - Makes PC=$ra and starts executing from where we left

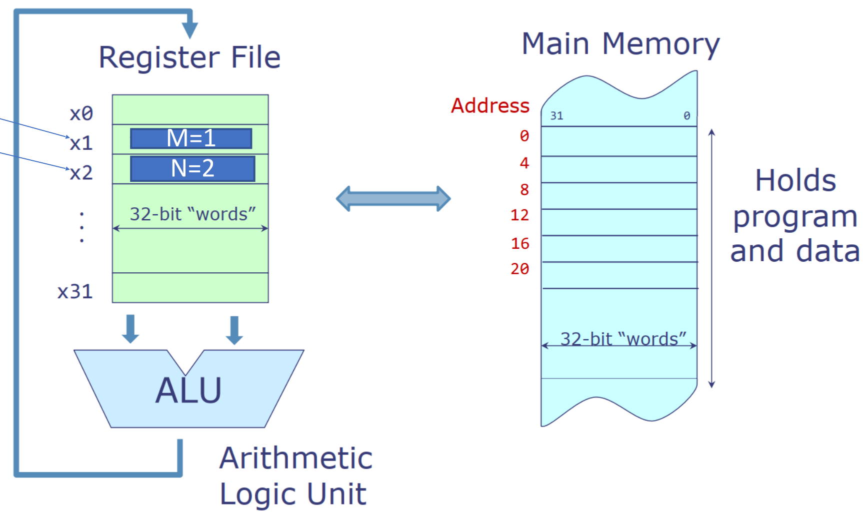
Question:
  - What is the value of PC after sum() execution completes?
  - What instruction is in that address?

```
void main(){
       m=1;
       n=2;
       o=sum(m,n); ---Address 500
       p=m+1;
}

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

Register Management during Procedure

Procedures could overwrite registers that are currently in use by the caller program.

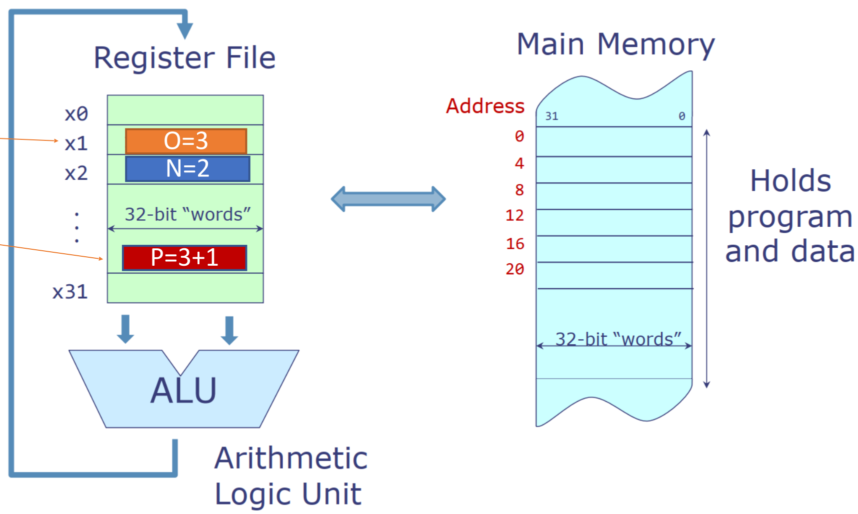
```
void main(){
        m=1;
        n=2;
        o=sum(m,n);
        p=m+1;
}

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



Register Management during Procedure

- Procedures could overwrite registers that are currently in use by the caller program.
- Only 32 registers are not enough for the compiler

```
void main(){
        m=1;
        n=2;
        o=sum(m,n);
        p=m+1;
}

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



Register Management during Procedure

- A caller uses the same register set as the called procedure
  - A caller should not rely on how the called procedure manages its register space
  - Ideally, procedure implementation should be able to use all registers
- Either the caller or the callee saves the caller’s registers in memory and restores them when the procedure call has completed execution
- We use stack data structures for saving the registers

MIPS Registers

Some registers are preserved/stored onto the stack memory (by caller itself) before the procedure/callee uses them.

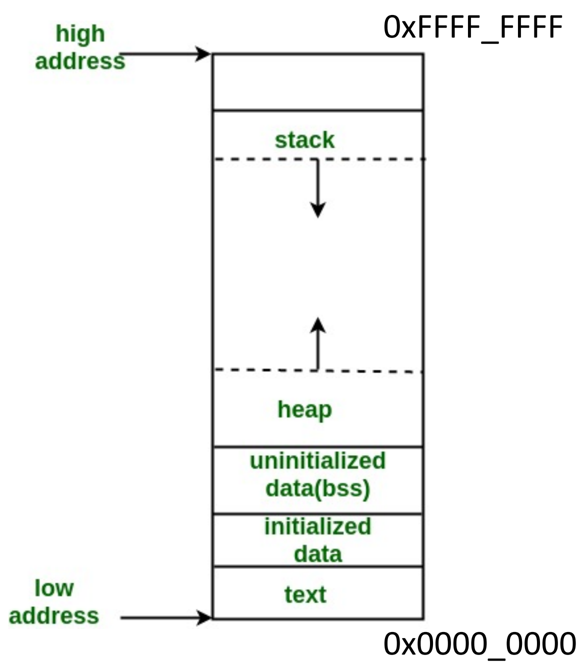
Name	Register number	Usage	Preserved on call?
$zero	0	The constant value 0	n.a.
$v0–$v1	2–3	Values for results and expression evaluation	no
$a0–$a3	4–7	Arguments	no
$t0–$t7	8–15	Temporaries	no
$s0–$s7	16–23	Saved	yes
$t8–$t9	24–25	More temporaries	no
$gp	28	Global pointer	yes
$sp	29	Stack pointer	yes
$fp	30	Frame pointer	yes
$ra	31	Return address	yes

Stack

- Stack is in memory →need a register to point to it
  - MIPS uses stack pointer register ($sp)

- Stack grows down from higher to lower addresses
  - Push decreases sp
  - Pop increases sp
  - $sp points to top of stack (last pushed element)

- Rule of using stack:
  - Can use stack at any time, but leave it as you found it!



Example (page 81)

Let's turn the example on page 51 into a C procedure:

```
int leaf_example (int g, int h, int i, int j)
{
    int f;

    f = (g + h) - (i + j);
    return f;
}
```

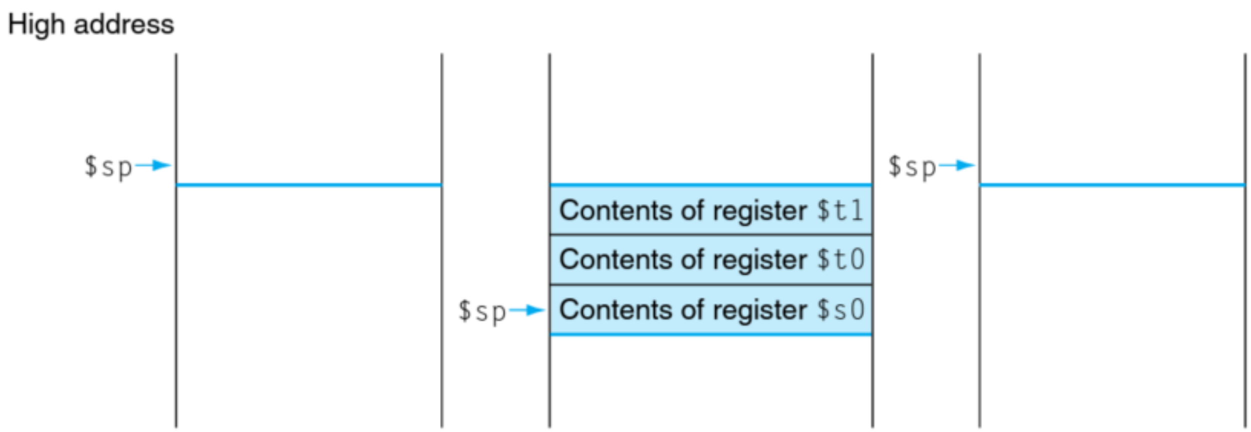
What is the compiled MIPS assembly code?

Assumptions: leaf_example is called by main().
Main() was using $t1, $t0,
Main() has stored input argument g, h, I and j to register $a0 to $a4, the return value f is in $s0

Example (page 81)

Step 1: Push the old values on the stack

```
addi $sp,$sp,-12 # adjust stack to make room for 3 items
sw   $t1, 8($sp)  # save register $t1 for use afterwards
sw   $t0, 4($sp)  # save register $t0 for use afterwards
sw   $s0, 0($sp)  # save register $s0 for use afterwards
```



Example (page 81)

- Step 2: Compute

```
add $t0,$a0,$a1 # register $t0 contains g + h
add $t1,$a2,$a3 # register $t1 contains i + j
sub $s0,$t0,$t1 # f = $t0 - $t1, which is (g + h)-(i + j)
```

- Step 3: Store return value

```
add $v0,$s0,$zero # returns f ($v0 = $s0 + 0)
```

- Step 4: Clear Stack

```
lw   $s0, 0($sp)  # restore register $s0 for caller
lw   $t0, 4($sp)  # restore register $t0 for caller
lw   $t1, 8($sp)  # restore register $t1 for caller
addi $sp,$sp,12  # adjust stack to delete 3 items
```

- Step 4: Go back to main()

```
jr   $ra     # jump back to calling routine
```