Compiling Code, Procedures and Stacks

RISC-V Recap

- Computational Instructions executed by ALU
  - Register-Register: op dest, src1, src2
  - Register-Immediate: op dest, src1, const

RISC-V Recap

- Computational Instructions executed by ALU
  - Register-Register: op dest, src1, src2
  - Register-Immediate: op dest, src1, const
- Control flow instructions
  - Unconditional: jal label and jalr register
  - Conditional: br_comp src1, src2, label

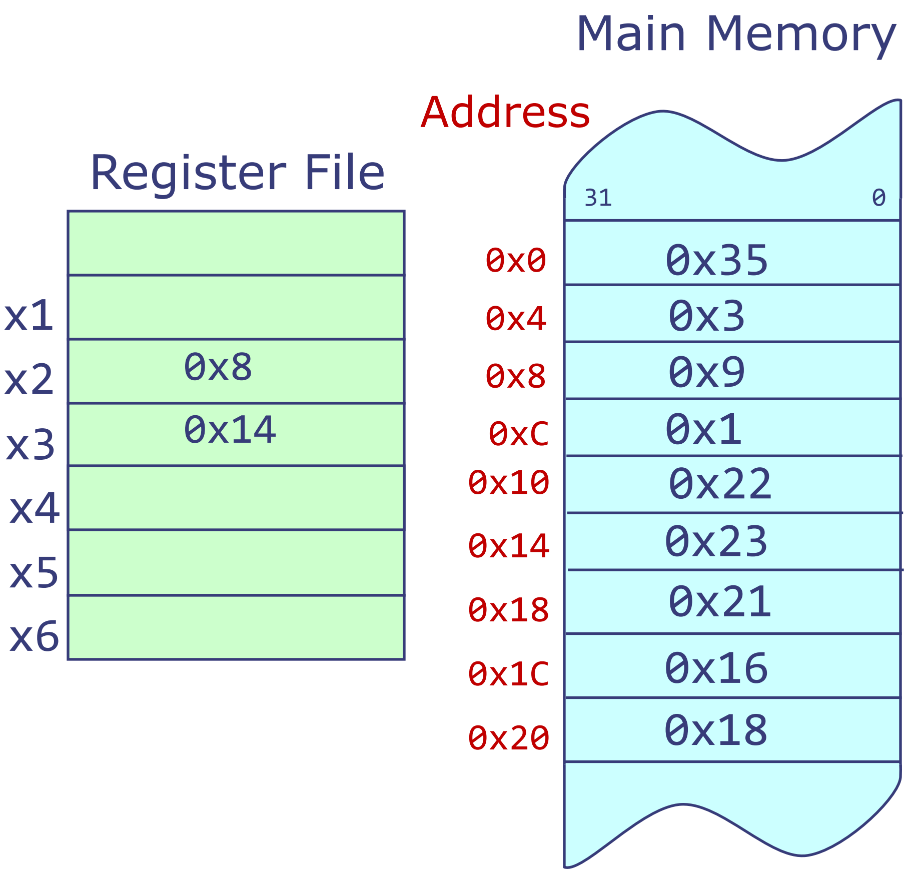
RISC-V Recap

- Computational Instructions executed by ALU
  - Register-Register: op dest, src1, src2
  - Register-Immediate: op dest, src1, const
- Control flow instructions
  - Unconditional: jal label and jalr register
  - Conditional: br_comp src1, src2, label
- Loads and Stores
  - lw dest, offset(base)
  - sw src, offset(base)
  - Base is a register, offset is a small constant

RISC-V Recap

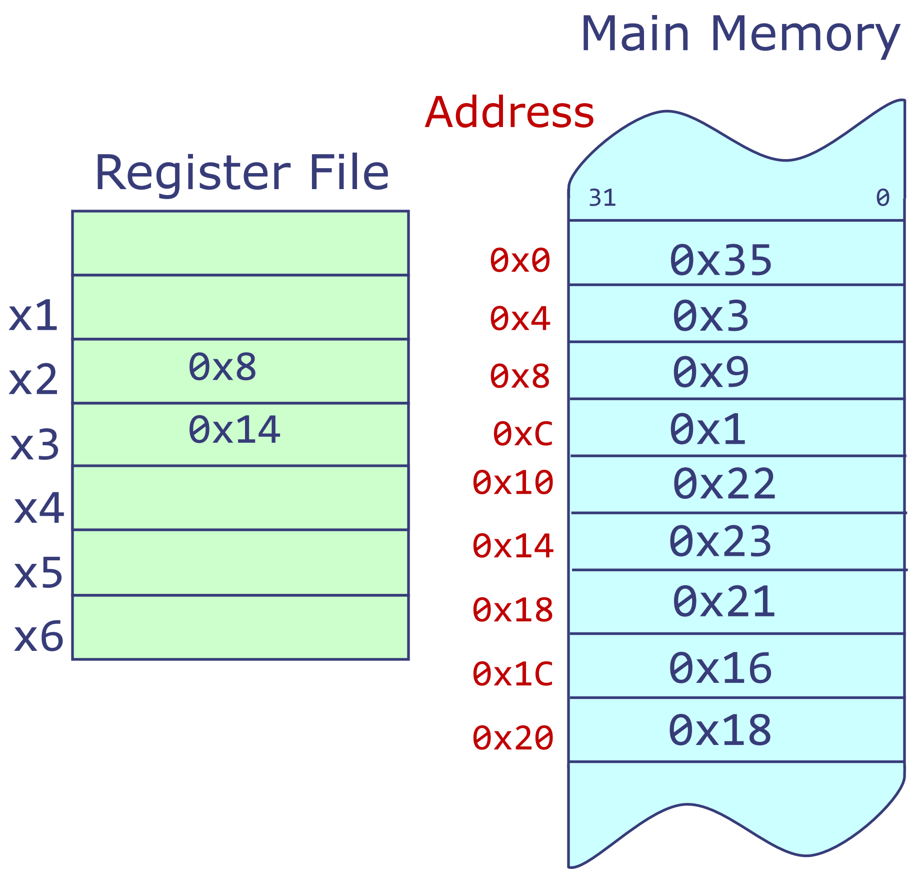
- Computational Instructions executed by ALU
  - Register-Register: op dest, src1, src2
  - Register-Immediate: op dest, src1, const
- Control flow instructions
  - Unconditional: jal label and jalr register
  - Conditional: br_comp src1, src2, label
- Loads and Stores
  - lw dest, offset(base)
  - sw src, offset(base)
  - Base is a register, offset is a small constant
- Pseudoinstructions
  - Shorthand for other instructions

Registers vs Memory



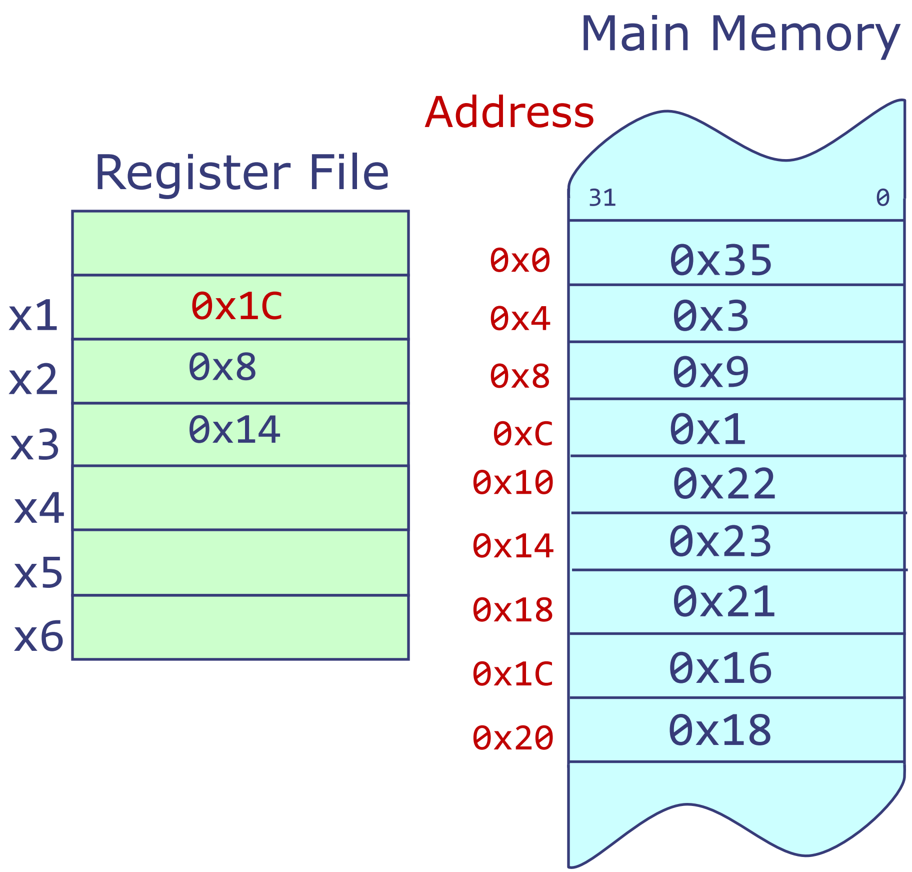
Registers vs Memory

```
add x1, x2, x3
    x1 =
```



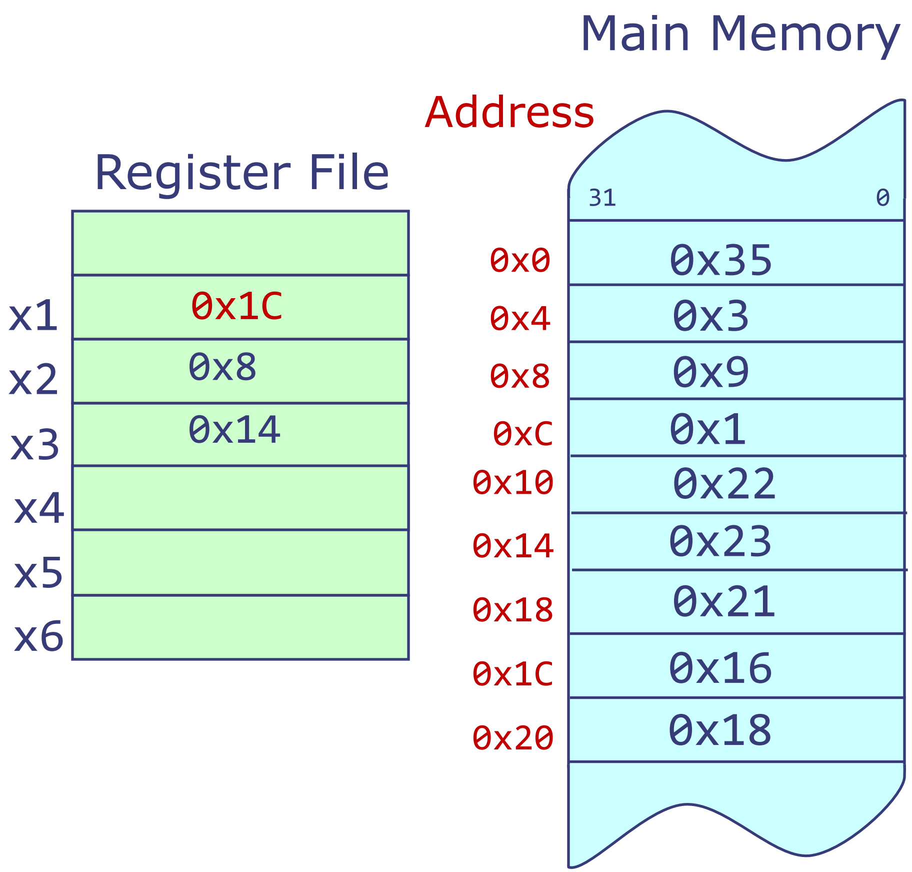
Registers vs Memory

```
add x1, x2, x3
    x1 = 0x1C
```



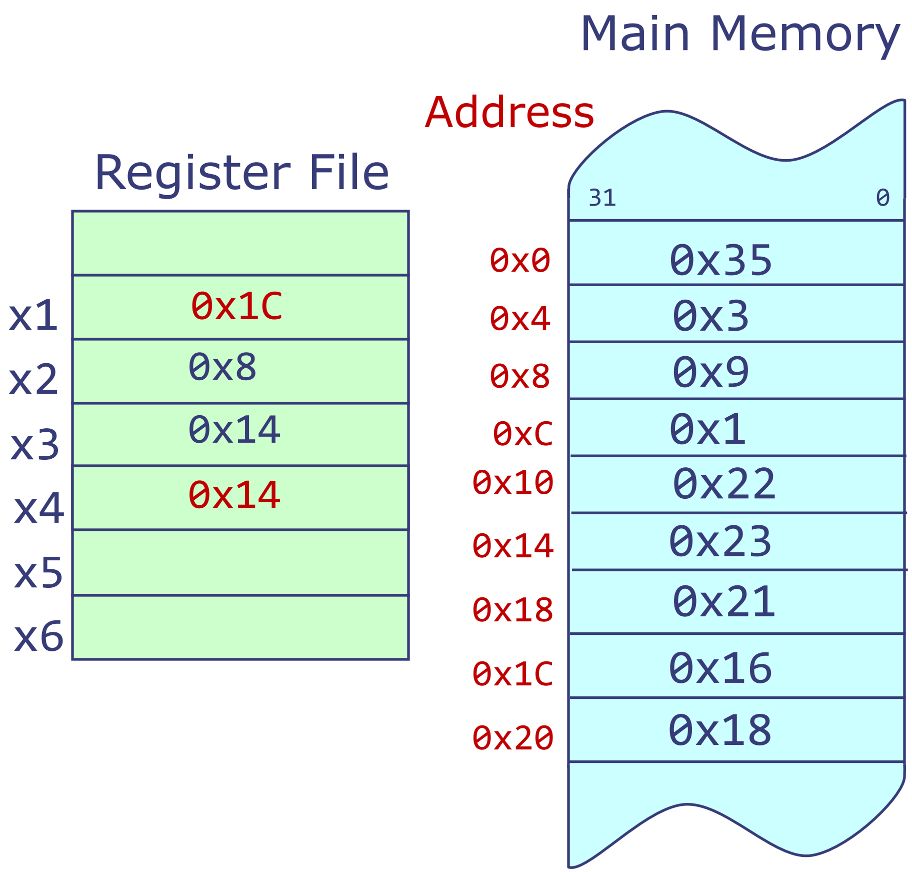
Registers vs Memory

```
add x1, x2, x3
    x1 = 0x1C

mv x4, x3
    x4 =
```



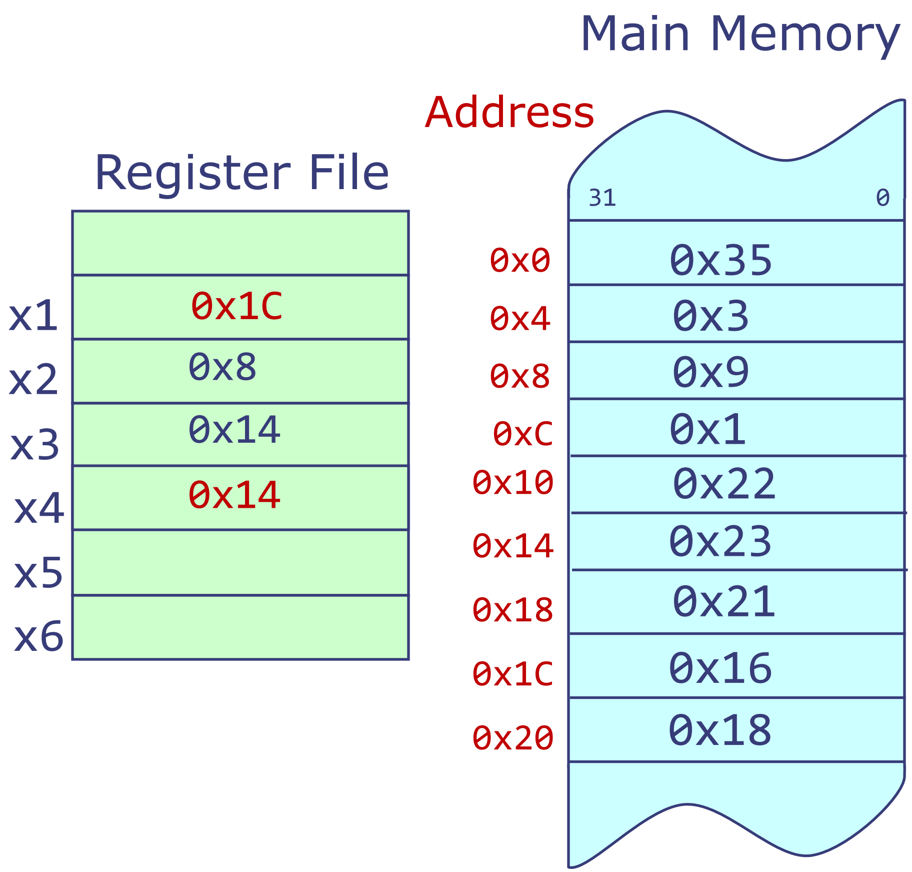
Registers vs Memory

```
add x1, x2, x3
    x1 = 0x1C

mv x4, x3
    x4 = 0x14
```



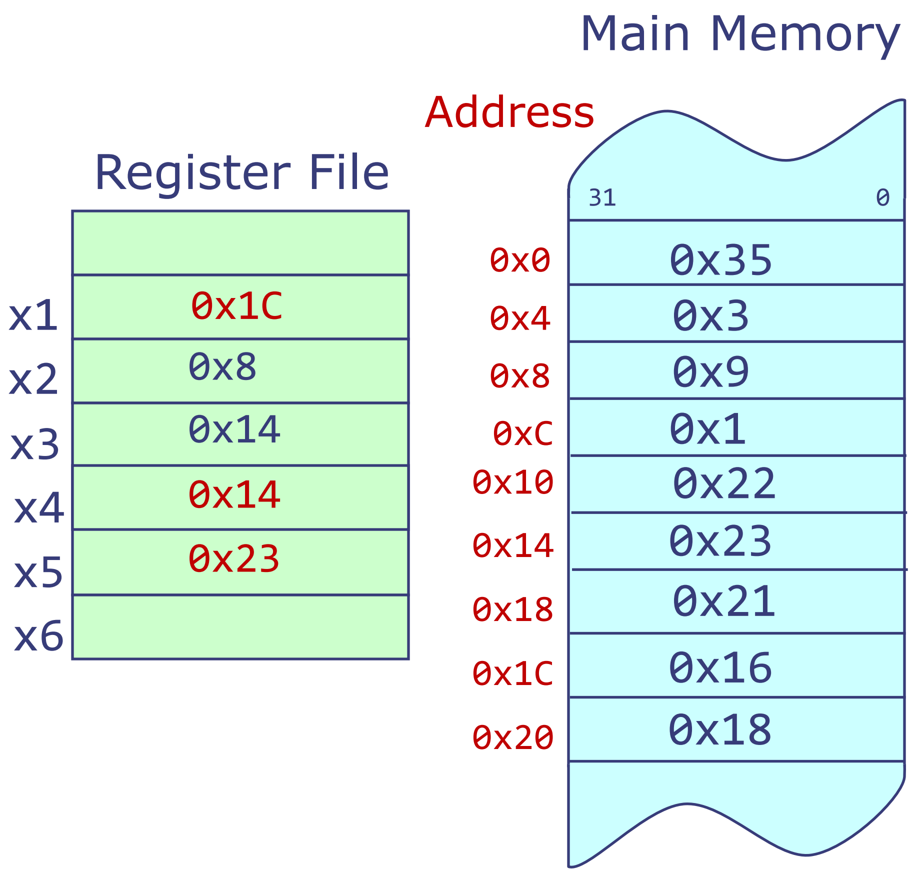
Registers vs Memory

```
add x1, x2, x3
    x1 = 0x1C

mv x4, x3
    x4 = 0x14

lw x5, 0(x3)
    x5 =
```



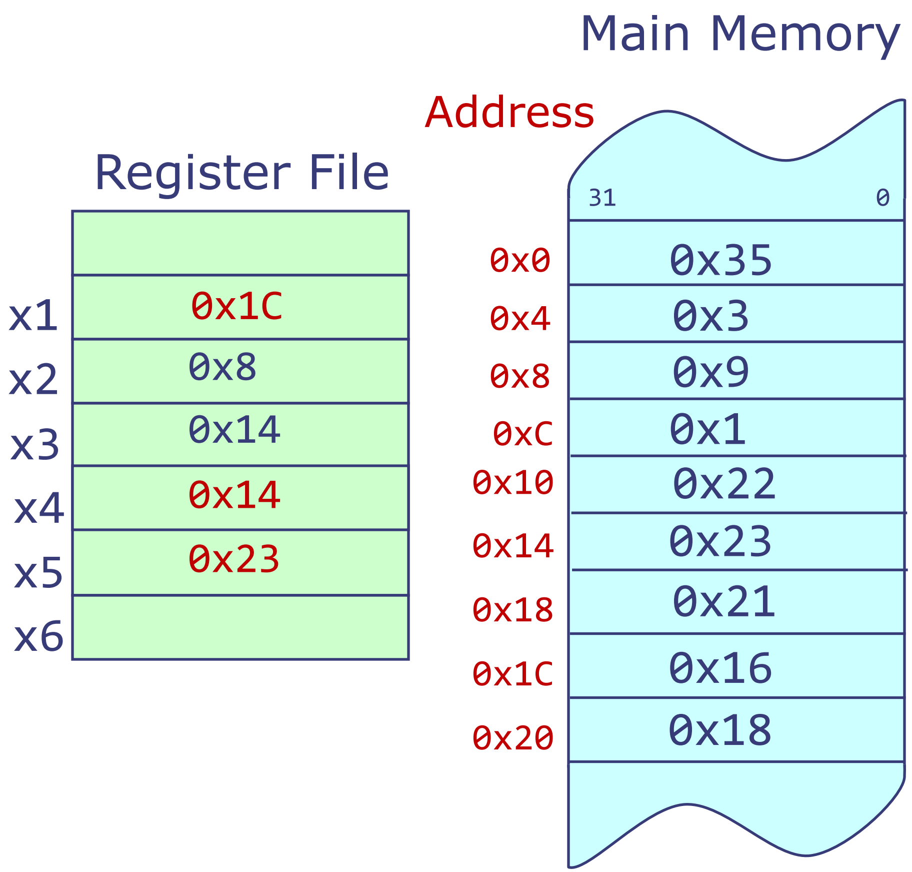
Registers vs Memory

```
add x1, x2, x3
    x1 = 0x1C

mv x4, x3
    x4 = 0x14

lw x5, 0(x3)
    x5 = 0x23
```



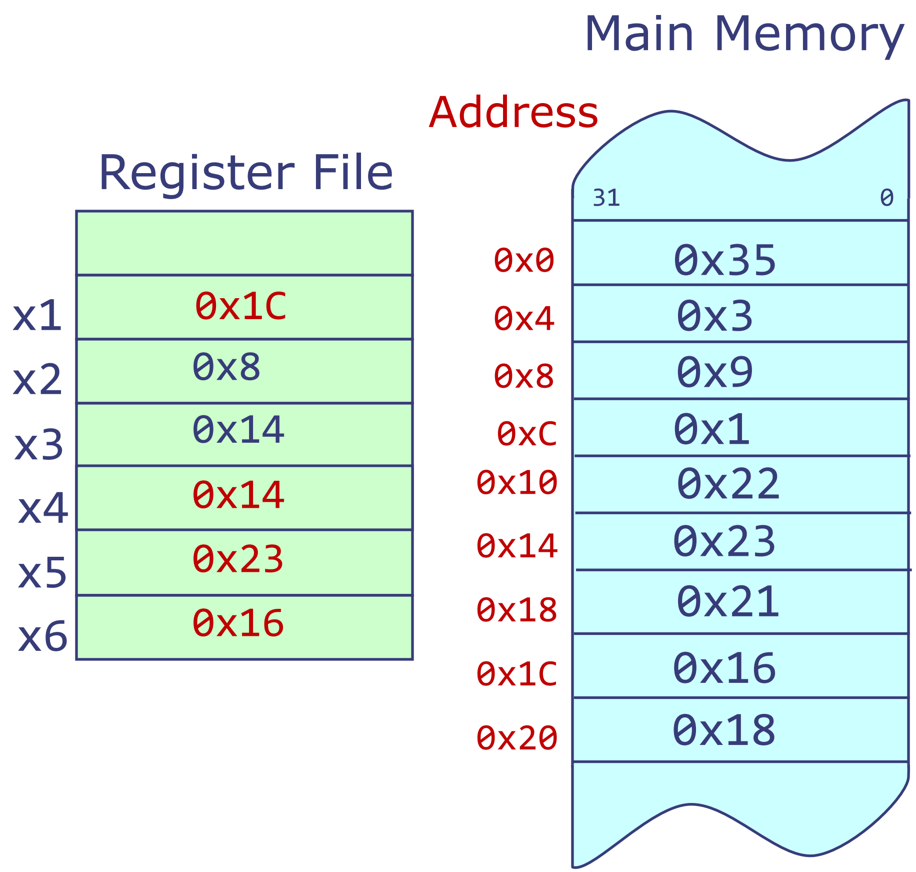
Registers vs Memory

```
add x1, x2, x3
    x1 = 0x1C

mv x4, x3
    x4 = 0x14

lw x5, 0(x3)
    x5 = 0x23

lw x6, 8(x3)
    x6 =
```



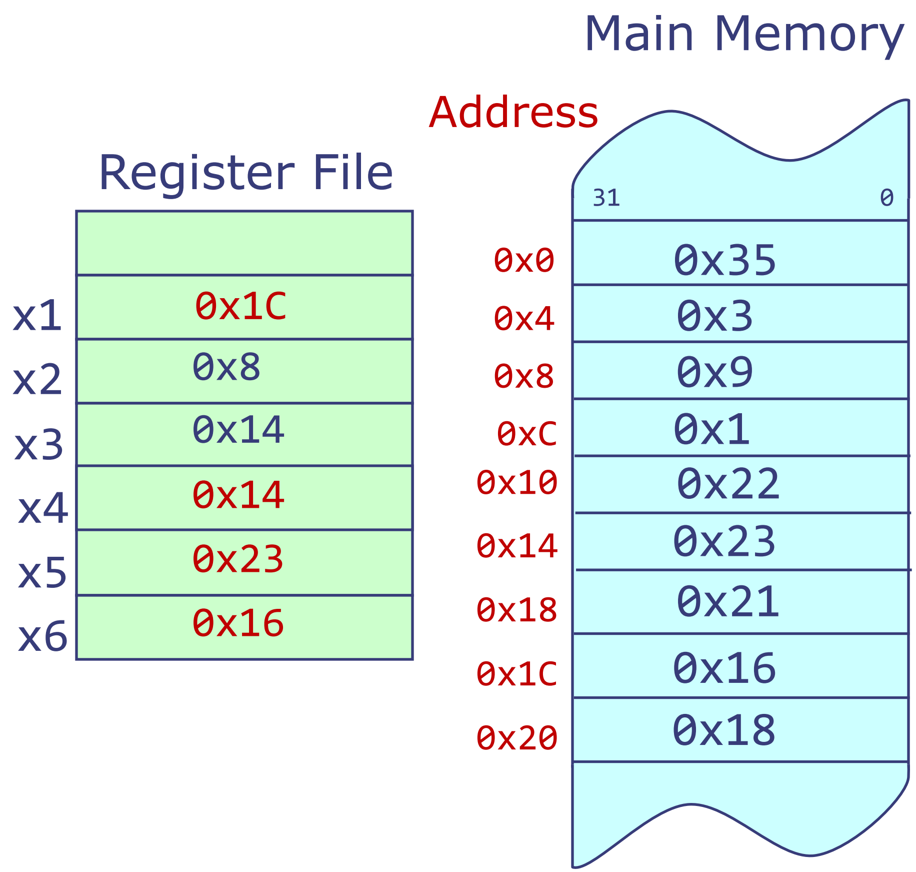
Registers vs Memory

```
add x1, x2, x3
    x1 = 0x1C

mv x4, x3
    x4 = 0x14

lw x5, 0(x3)
    x5 = 0x23

lw x6, 8(x3)
    x6 = 0x16
```



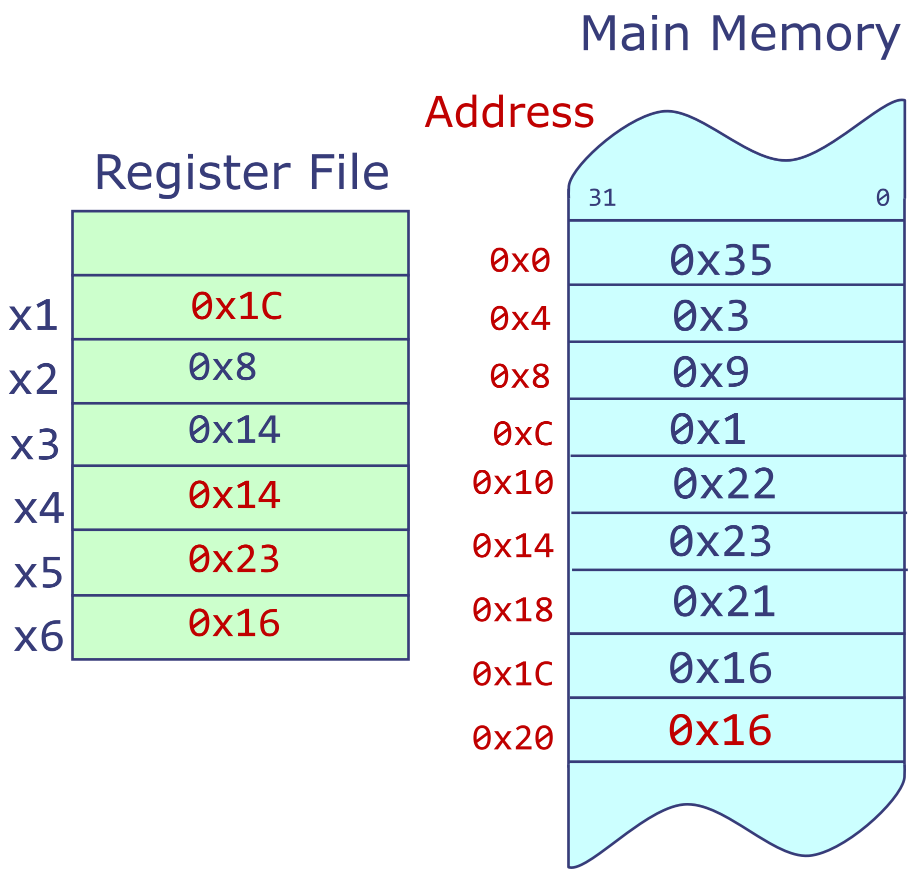
Registers vs Memory

```
add x1, x2, x3
    x1 = 0x1C

mv x4, x3
    x4 = 0x14

lw x5, 0(x3)
    x5 = 0x23

lw x6, 8(x3)
    x6 = 0x16

sw x6, 0xC(x3)
```



Registers vs Memory

```
add x1, x2, x3
    x1 = 0x1C

mv x4, x3
    x4 = 0x14

lw x5, 0(x3)
    x5 = 0x23

lw x6, 8(x3)
    x6 = 0x16

sw x6, 0xC(x3)
value of x6 (0x16)
is written to M[0x14+0xC]
```



Dealing with Constants

- Execute `a = b+3`

Assume a is in register x1 and b is in x2.

Dealing with Constants

Assume a is in register x1 and b is in x2.

- Execute `a = b+3`
  - Small constants (12-bit) can be handled via Register-Immediate ALU operations

Dealing with Constants

Assume a is in register x1 and b is in x2.

- Execute `a = b+3`
  - Small constants (12-bit) can be handled via Register-Immediate ALU operations

```
addi x1, x2, 3
```

Dealing with Constants

Assume a is in register x1 and b is in x2.

- Execute `a = b+3`
  - Small constants (12-bit) can be handled via Register-Immediate ALU operations

```
addi x1, x2, 3
```

- Execute `a = b+0x123456`

Dealing with Constants

Assume a is in register x1 and b is in x2.

- Execute `a = b+3`
  - Small constants (12-bit) can be handled via Register-Immediate ALU operations

```
addi x1, x2, 3
```

- Execute `a = b+0x123456`
  - Largest 12 bit 2's complement constant is 2^{11}-1 = 2047 (0x7FF)
  - Use `li` pseudoinstruction to set register to large constant

Dealing with Constants

Assume a is in register x1 and b is in x2.

- Execute `a = b+3`
  - Small constants (12-bit) can be handled via Register-Immediate ALU operations

```
addi x1, x2, 3
```

- Execute `a = b+0x123456`
  - Largest 12 bit 2's complement constant is $2^{11}-1$ = 2047 (0x7FF)
  - Use `li` pseudoinstruction to set register to large constant

```
li x4, 0x123456
```

Dealing with Constants

Assume a is in register x1 and b is in x2.

- Execute `a = b+3`
  - Small constants (12-bit) can be handled via Register-Immediate ALU operations

```
addi x1, x2, 3
```

- Execute `a = b+0x123456`
  - Largest 12 bit 2's complement constant is 2^{11}-1 = 2047 (0x7FF)
  - Use `li` pseudoinstruction to set register to large constant

```
li x4, 0x123456
```

```
lui x4, 0x123
addi x4, x4, 0x456
```

Dealing with Constants

Assume a is in register x1 and b is in x2.

- Execute `a = b+3`
  - Small constants (12-bit) can be handled via Register-Immediate ALU operations

```
addi x1, x2, 3
```

- Execute `a = b+0x123456`
  - Largest 12 bit 2's complement constant is $2^{11}-1$ = 2047 (0x7FF)
  - Use `li` pseudoinstruction to set register to large constant

```
li x4, 0x123456
```

```
lui x4, 0x123
addi x4, x4, 0x456
```

`x4 = 0x123000`

Dealing with Constants

Assume a is in register x1 and b is in x2.

- Execute `a = b+3`
  - Small constants (12-bit) can be handled via Register-Immediate ALU operations

```
addi x1, x2, 3
```

- Execute `a = b+0x123456`
  - Largest 12 bit 2's complement constant is 2^{11}-1 = 2047 (0x7FF)
  - Use `li` pseudoinstruction to set register to large constant

```
li x4, 0x123456
```

```
lui x4, 0x123
addi x4, x4, 0x456
```

x4 = 0x123000

  - Can also use `li` pseudoinstruction for small constants

Dealing with Constants

Assume a is in register x1 and b is in x2.

- Execute `a = b+3`
  - Small constants (12-bit) can be handled via Register-Immediate ALU operations

```
addi x1, x2, 3
```

- Execute `a = b+0x123456`
  - Largest 12 bit 2's complement constant is 2^{11}-1 = 2047 (0x7FF)
  - Use `li` pseudoinstruction to set register to large constant

```
li x4, 0x123456
```

```
lui x4, 0x123
addi x4, x4, 0x456
```

`x4 = 0x123000`

  - Can also use `li` pseudoinstruction for small constants

```
li x4, 0x12
```

```
addi x4, x0, 0x12
```

Compiling Simple Expressions

- Assign variables to registers
- Translate operators into computational instructions
- Use register-immediate instructions to handle operations with small constants
- Use the `li` pseudoinstruction for large constants

Example C code

```
int x, y, z;
…
y = (x + 3) | (y + 123456);
z = (x * 4) ^ y;
```

Compiling Simple Expressions

- Assign variables to registers
- Translate operators into computational instructions
- Use register-immediate instructions to handle operations with small constants
- Use the `li` pseudoinstruction for large constants

Example C code

```
int x, y, z;
…
y = (x + 3) | (y + 123456);
z = (x * 4) ^ y;
```

RISC-V Assembly

```
// x: x10, y: x11, z: x12
// x13, x14 used for temporaries
```

Compiling Simple Expressions

- Assign variables to registers
- Translate operators into computational instructions
- Use register-immediate instructions to handle operations with small constants
- Use the `li` pseudoinstruction for large constants

Example C code

```
int x, y, z;
…
y = (x + 3) | (y + 123456);
z = (x * 4) ^ y;
```

RISC-V Assembly

```
// x: x10, y: x11, z: x12
// x13, x14 used for temporaries
```

Compiling Simple Expressions

- Assign variables to registers
- Translate operators into computational instructions
- Use register-immediate instructions to handle operations with small constants
- Use the `li` pseudoinstruction for large constants

Example C code

```c
int x, y, z;
…
y = (x + 3) | (y + 123456);
z = (x * 4) ^ y;
```

RISC-V Assembly

```
// x: x10, y: x11, z: x12
// x13, x14 used for temporaries
addi x13, x10, 3
```

Compiling Simple Expressions

- Assign variables to registers
- Translate operators into computational instructions
- Use register-immediate instructions to handle operations with small constants
- Use the `li` pseudoinstruction for large constants

Example C code

```
int x, y, z;
…
y = (x + 3) | (y + 123456);
z = (x * 4) ^ y;
```

RISC-V Assembly

```
// x: x10, y: x11, z: x12
// x13, x14 used for temporaries
addi x13, x10, 3
```

Compiling Simple Expressions

- Assign variables to registers
- Translate operators into computational instructions
- Use register-immediate instructions to handle operations with small constants
- Use the `li` pseudoinstruction for large constants

Example C code

```
int x, y, z;
…
y = (x + 3) | (y + 123456);
z = (x * 4) ^ y;
```

RISC-V Assembly

```
// x: x10, y: x11, z: x12
// x13, x14 used for temporaries
addi x13, x10, 3
li x14, 123456
add x14, x11, x14
```

Compiling Simple Expressions

- Assign variables to registers
- Translate operators into computational instructions
- Use register-immediate instructions to handle operations with small constants
- Use the `li` pseudoinstruction for large constants

Example C code

```
int x, y, z;
…
y = (x + 3) | (y + 123456);
z = (x * 4) ^ y;
```

RISC-V Assembly

```
// x: x10, y: x11, z: x12
// x13, x14 used for temporaries
addi x13, x10, 3
li x14, 123456
add x14, x11, x14
```

Compiling Simple Expressions

- Assign variables to registers
- Translate operators into computational instructions
- Use register-immediate instructions to handle operations with small constants
- Use the `li` pseudoinstruction for large constants

Example C code

```
int x, y, z;
…
y = (x + 3) | (y + 123456);
z = (x * 4) ^ y;
```

RISC-V Assembly

```
// x: x10, y: x11, z: x12
// x13, x14 used for temporaries
addi x13, x10, 3
li x14, 123456
add x14, x11, x14
or x11, x13, x14
```

Compiling Simple Expressions

- Assign variables to registers
- Translate operators into computational instructions
- Use register-immediate instructions to handle operations with small constants
- Use the `li` pseudoinstruction for large constants

Example C code

```c
int x, y, z;
…
y = (x + 3) | (y + 123456);
z = (x * 4) ^ y;
```

RISC-V Assembly

```
// x: x10, y: x11, z: x12
// x13, x14 used for temporaries
addi x13, x10, 3
li x14, 123456
add x14, x11, x14
or x11, x13, x14
```

Compiling Simple Expressions

- Assign variables to registers
- Translate operators into computational instructions
- Use register-immediate instructions to handle operations with small constants
- Use the `li` pseudoinstruction for large constants

Example C code

```
int x, y, z;
…
y = (x + 3) | (y + 123456);
z = (x * 4) ^ y;
```

RISC-V Assembly

```
// x: x10, y: x11, z: x12
// x13, x14 used for temporaries
addi x13, x10, 3
li x14, 123456
add x14, x11, x14
or x11, x13, x14
slli x13, x10, 2
```

Compiling Simple Expressions

- Assign variables to registers
- Translate operators into computational instructions
- Use register-immediate instructions to handle operations with small constants
- Use the `li` pseudoinstruction for large constants

Example C code

```
int x, y, z;
…
y = (x + 3) | (y + 123456);
z = (x * 4) ^ y;
```

RISC-V Assembly

```
// x: x10, y: x11, z: x12
// x13, x14 used for temporaries
addi x13, x10, 3
li x14, 123456
add x14, x11, x14
or x11, x13, x14
slli x13, x10, 2
```

Compiling Simple Expressions

- Assign variables to registers
- Translate operators into computational instructions
- Use register-immediate instructions to handle operations with small constants
- Use the `li` pseudoinstruction for large constants

Example C code

```c
int x, y, z;
…
y = (x + 3) | (y + 123456);
z = (x * 4) ^ y;
```

RISC-V Assembly

```
// x: x10, y: x11, z: x12
// x13, x14 used for temporaries
addi x13, x10, 3
li x14, 123456
add x14, x11, x14
or x11, x13, x14
slli x13, x10, 2
xor x12, x13, x11
```

Compiling Conditionals

- *if* statements can be compiled using branches:

C code	RISC-V Assembly

```
if (expr) {                (compile expr into xN)
  if-body                  beqz xN, endif
}                          (compile if-body)
                         endif:
```

Compiling Conditionals

- *if* statements can be compiled using branches:

C code

```
if (expr) {
  if-body
}
```

RISC-V Assembly

```
  (compile expr into xN)
  beqz xN, endif
  (compile if-body)
endif:
```

- *Example: Compile the following C code*

```
int x, y;
…
if (x < y) {
  y = y - x;
}
```

Compiling Conditionals

- *if* statements can be compiled using branches:

C code

```
if (expr) {
  if-body
}
```

RISC-V Assembly

```
  (compile expr into xN)
  beqz xN, endif
  (compile if-body)
endif:
```

- *Example: Compile the following C code*

```
int x, y;          // x: x10, y: x11
…
if (x < y) {
  y = y - x;
}
```

Compiling Conditionals

- *if* statements can be compiled using branches:

C code

```
if (expr) {
  if-body
}
```

RISC-V Assembly

```
  (compile expr into xN)
  beqz xN, endif
  (compile if-body)
endif:
```

- *Example: Compile the following C code*

```
int x, y;
…
if (x < y) {
  y = y - x;
}
```

```
// x: x10, y: x11
slt x12, x10, x11
```

Compiling Conditionals

- *if* statements can be compiled using branches:

C code

```
if (expr) {
  if-body
}
```

RISC-V Assembly

```
  (compile expr into xN)
  beqz xN, endif
  (compile if-body)
endif:
```

- *Example: Compile the following C code*

```
int x, y;
…
if (x < y) {
  y = y – x;
}
```

```
  // x: x10, y: x11
  slt x12, x10, x11
  beqz x12, endif

endif:
```

Compiling Conditionals

- *if* statements can be compiled using branches:

C code

```
if (expr) {
  if-body
}
```

RISC-V Assembly

```
  (compile expr into xN)
  beqz xN, endif
  (compile if-body)
endif:
```

- *Example: Compile the following C code*

```
int x, y;
…
if (x < y) {
  y = y - x;
}
```

```
  // x: x10, y: x11
  slt x12, x10, x11
  beqz x12, endif
  sub x11, x11, x10
endif:
```

Compiling Conditionals

- *if* statements can be compiled using branches:

C code

```
if (expr) {
  if-body
}
```

RISC-V Assembly

```
  (compile expr into xN)
  beqz xN, endif
  (compile if-body)
endif:
```

- *Example: Compile the following C code*

```
int x, y;
…
if (x < y) {
  y = y - x;
}
```

```
  // x: x10, y: x11
  slt x12, x10, x11
  beqz x12, endif
  sub x11, x11, x10
endif:
```

We can sometimes combine *expr* and the branch

```
  bge x10, x11, endif
  sub x11, x11, x10
endif:
```

Compiling Conditionals

- *if-else* statements are similar:

C code

```
if (expr) {
  if-body
} else {
  else-body
}
```

RISC-V Assembly

```
  (compile expr into xN)
  beqz xN, else
  (compile if-body)
  j endif
else:
  (compile else-body)
endif:
```

Compiling Loops

- Loops can be compiled using *backward* branches:

C code

```
while (expr) {
  while-body
}
```

RISC-V Assembly

```
while:
  (compile expr into xN)
  beqz xN, endwhile
  (compile while-body)
  j while
endwhile:
```

Compiling Loops

- Loops can be compiled using *backward* branches:

C code

```
while (expr) {
  while-body
}
```

RISC-V Assembly

```
while:
  (compile expr into xN)
  beqz xN, endwhile
  (compile while-body)
  j while
endwhile:
```

Compiling Loops

- Loops can be compiled using *backward* branches:

C code

```
while (expr) {
  while-body
}
```

RISC-V Assembly

```
while:
  (compile expr into xN)
  beqz xN, endwhile
  (compile while-body)
  j while
endwhile:
```

- *Can you write a version that executes fewer instructions?*

Compiling Loops

- Loops can be compiled using *backward* branches:

C code

```
while (expr) {
  while-body
}
```

RISC-V Assembly

```
while:
  (compile expr into xN)
  beqz xN, endwhile
  (compile while-body)
  j while
endwhile:
```

- *Can you write a version that executes fewer instructions?*

```
  // Version with one branch
  // or jump per iteration
  j compare
loop:
  (compile while-body)
compare:
  (compile expr into xN)
  bnez xN, loop
```

Putting it all together

C code

```
while (x != y) {
    if (x > y) {
        x = x - y;
    } else {
        y = y - x;
    }
}
```

RISC-V Assembly

Putting it all together

C code

```
while (x != y) {
    if (x > y) {
        x = x - y;
    } else {
        y = y - x;
    }
}
```

RISC-V Assembly

```
// x: x10, y: x11
```

Putting it all together

C code	RISC-V Assembly

C code

```
while (x != y) {
   if (x > y) {
      x = x - y;
   } else {
      y = y - x;
   }
}
```

RISC-V Assembly

```
// x: x10, y: x11
j compare
loop:
  (compile while-body)
compare:
      bne x10, x11, loop
```

Putting it all together

C code

```c
while (x != y) {
   if (x > y) {
      x = x - y;
    } else {
      y = y - x;
    }
}
```

RISC-V Assembly

```
// x: x10, y: x11
j compare
loop:
  ble x10, x11, else
  sub x10, x10, x11
  j endif
else:
  sub x11, x11, x10
endif:
compare:
     bne x10, x11, loop
```

Procedures

C code

```c
int gcd(int a, int b)
{
  int x = a;
  int y = b;
  while (x != y) {
   if (x > y) {
      x = x - y;
    } else {
      y = y - x;
    }
  }
  return x;
}
```

RISC-V Assembly

```
// x: x10, y: x11
j compare
loop:
  ble x10, x11 else
  sub x10, x10, x11
  j endif
else:
  sub x11, x11, x10
endif:
compare:
      bne x10, x11, loop
```

Procedures

- Procedure (a.k.a. function or subroutine): Reusable code fragment that performs a specific task

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

Procedures

- Procedure (a.k.a. function or subroutine): Reusable code fragment that performs a specific task
  - Single named entry point

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

Procedures

- Procedure (a.k.a. function or subroutine): Reusable code fragment that performs a specific task
  - Single named entry point
  - Zero or more formal arguments

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

Procedures

- Procedure (a.k.a. function or subroutine): Reusable code fragment that performs a specific task
  - Single named entry point
  - Zero or more formal arguments
  - Local storage

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

Procedures

- Procedure (a.k.a. function or subroutine): Reusable code fragment that performs a specific task
  - Single named entry point
  - Zero or more formal arguments
  - Local storage
  - Returns to the caller when finished

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

Procedures

- Procedure (a.k.a. function or subroutine): Reusable code fragment that performs a specific task
  - Single named entry point
  - Zero or more formal arguments
  - Local storage
  - Returns to the caller when finished

- Using procedures enables abstraction and reuse
  - Compose large programs from collections of simple procedures

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

bool coprimes(int a, int b) {
  return gcd(a, b) == 1;
}

coprimes(5, 10); // false
coprimes(9, 10); // true
```

Managing a procedure's register space

- A caller uses the same register set as the called procedure

Managing a procedure’s register space

- 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

Managing a procedure’s register space

- 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

Managing a procedure's register space

- 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

Implementing procedures

- A caller needs to pass arguments to the called procedure, as well as get results back from the called procedure
  - both are done through registers

Implementing procedures

- A caller needs to pass arguments to the called procedure, as well as get results back from the called procedure
  - both are done through registers
- A procedure can be called from many different places

```
        …
[0x100] j sum
        …
[0x678] j sum
        …
```

```
sum:
        …
        j ?
```

Implementing procedures

- A caller needs to pass arguments to the called procedure, as well as get results back from the called procedure
  - both are done through registers
- A procedure can be called from many different places
  - The caller can get to the called procedure code simply by executing a unconditional jump instruction

```
        …
[0x100] j sum
        …
[0x678] j sum
        …
```

```
sum:
        …
        j ?
```

Implementing procedures

- A caller needs to pass arguments to the called procedure, as well as get results back from the called procedure
  - both are done through registers
- A procedure can be called from many different places
  - The caller can get to the called procedure code simply by executing a unconditional jump instruction
  - However, to return to the correct place in the calling procedure, the called procedure has to know which of the possible return addresses it should use

```
        …
[0x100] j sum
        …
[0x678] j sum
        …
```

```
sum:
        …
        j ?
```

0x104?
0x67C?

Implementing procedures

- A caller needs to pass arguments to the called procedure, as well as get results back from the called procedure
  - both are done through registers
- A procedure can be called from many different places
  - The caller can get to the called procedure code simply by executing a unconditional jump instruction
  - However, to return to the correct place in the calling procedure, the called procedure has to know which of the possible return addresses it should use

```
        …
[0x100] j sum
        …
[0x678] j sum
        …
```

```
sum:
        …
        j ?
```

```
0x104?
0x67C?
```

Return address must be saved and passed to the called procedure!

Procedure Linking

- How to transfer control to callee and back to caller?

  `proc_call: jal ra, label`

  1. Stores address of proc_call + 4 in register ra (return address register)

```
        …
[0x100] jal ra, sum
        …
[0x678] jal ra, sum
        …
```

```
sum:
      …
      jr ra
```

Procedure Linking

- How to transfer control to callee and back to caller?

  `proc_call: jal ra, label`

  1. Stores address of proc_call + 4 in register ra (return address register)
  2. Jumps to instruction at address label where label is the name of the procedure

```
        …
[0x100] jal ra, sum
        …
[0x678] jal ra, sum
        …
```

```
sum:
      …
      jr ra
```

Procedure Linking

- How to transfer control to callee and back to caller?

  `proc_call: jal ra, label`

  1. Stores address of proc_call + 4 in register ra (return address register)
  2. Jumps to instruction at address label where label is the name of the procedure
  3. After executing procedure, `jr ra` to return to caller and continue execution

```
        …
[0x100] jal ra, sum
        …
[0x678] jal ra, sum
        …
```

```
sum:
    …
    jr ra
```

Procedure Linking

- How to transfer control to callee and back to caller?

  `proc_call: jal ra, label`

  1. Stores address of proc_call + 4 in register ra (return address register)
  2. Jumps to instruction at address label where label is the name of the procedure
  3. After executing procedure, `jr ra` to return to caller and continue execution

```
        …
[0x100] jal ra, sum
        …
[0x678] jal ra, sum
        …
```

ra = 0x104

```
sum:
        …
        jr ra
```

Procedure Linking

- How to transfer control to callee and back to caller?

  `proc_call: jal ra, label`

  1. Stores address of proc_call + 4 in register ra (return address register)
  2. Jumps to instruction at address label where label is the name of the procedure
  3. After executing procedure, `jr ra` to return to caller and continue execution

```
        …
[0x100] jal ra, sum
        …
[0x678] jal ra, sum
        …
```

ra = 0x104

```
sum:
        …
        jr ra
```

1st time: jump to 0x104

Procedure Linking

- How to transfer control to callee and back to caller?

  `proc_call: jal ra, label`

  1. Stores address of proc_call + 4 in register ra (return address register)
  2. Jumps to instruction at address label where label is the name of the procedure
  3. After executing procedure, `jr ra` to return to caller and continue execution

```
        …
[0x100] jal ra, sum
        …
[0x678] jal ra, sum
        …
```

ra = 0x104

```
sum:
        …
        jr ra
```

1st time: jump to 0x104

Procedure Linking

- How to transfer control to callee and back to caller?

  `proc_call: jal ra, label`

  1. Stores address of proc_call + 4 in register ra (return address register)
  2. Jumps to instruction at address label where label is the name of the procedure
  3. After executing procedure, `jr ra` to return to caller and continue execution

```
        …
[0x100] jal ra, sum
        …
[0x678] jal ra, sum
        …
```

ra = 0x104

ra = 0x67C

```
sum:
      …
      jr ra
```

1st time: jump to 0x104

Procedure Linking

- How to transfer control to callee and back to caller?

  `proc_call: jal ra, label`

  1. Stores address of proc_call + 4 in register ra (return address register)
  2. Jumps to instruction at address label where label is the name of the procedure
  3. After executing procedure, `jr ra` to return to caller and continue execution

```
        …
[0x100] jal ra, sum
        …
[0x678] jal ra, sum
        …
```

ra = 0x104

ra = 0x67C

```
sum:
      …
      jr ra
```

1st time: jump to 0x104

2nd time: jump to 0x67C

Procedure Linking

- How to transfer control to callee and back to caller?

  `proc_call: jal ra, label`

  1. Stores address of proc_call + 4 in register ra (return address register)
  2. Jumps to instruction at address label where label is the name of the procedure
  3. After executing procedure, `jr ra` to return to caller and continue execution

```
        …
[0x100] jal ra, sum
        …
[0x678] jal ra, sum
        …
```

ra = 0x104

ra = 0x67C

```
sum:
      …
      jr ra
```

1st time: jump to 0x104

2nd time: jump to 0x67C

Procedure calls: Complications

- Suppose proc A calls proc B calls proc C
  - a single return address register won't work; the return address for proc B would wipe out the return address for proc A!
  - a similar complication arises in the memory space where the registers of proc A are saved – this space has to be different from the place where the registers of proc B are saved

Procedure Storage Needs

- Basic requirements for procedure calls:
  - Input arguments
  - Return address
  - Results

Procedure Storage Needs

- Basic requirements for procedure calls:
  - Input arguments
  - Return address
  - Results

- Local storage:
  - Variables that compiler can’t fit in registers
  - Space to save caller’s register values for registers that we overwrite

Procedure Storage Needs

- Basic requirements for procedure calls:
  - Input arguments
  - Return address
  - Results
- Local storage:
  - Variables that compiler can’t fit in registers
  - Space to save caller’s register values for registers that we overwrite

Each procedure call has its own instance of all this data known as the procedure’s *activation record.*

Insight (ca. 1960): We Need a Stack!

- Need data structure to hold activation records

- Activation records are allocated and deallocated in last-in-first-out (LIFO) order

Insight (ca. 1960): We Need a Stack!

- Need data structure to hold activation records

- Activation records are allocated and deallocated in last-in-first-out (LIFO) order

- Stack: push, pop, access to top element

Insight (ca. 1960): We Need a Stack!

- Need data structure to hold activation records

- Activation records are allocated and deallocated in last-in-first-out (LIFO) order

- Stack: push, pop, access to top element

- We only need to access to the activation record of the currently executing procedure

Insight (ca. 1960): We Need a Stack!

- Need data structure to hold activation records
- Activation records are allocated and deallocated in last-in-first-out (LIFO) order
- Stack: push, pop, access to top element

proc A

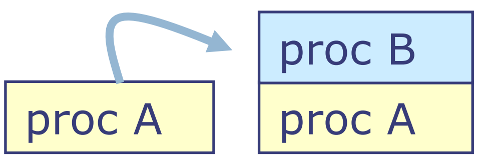
- We only need to access to the activation record of the currently executing procedure

Insight (ca. 1960): We Need a Stack!

- Need data structure to hold activation records

- Activation records are allocated and deallocated in last-in-first-out (LIFO) order

- Stack: push, pop, access to top element



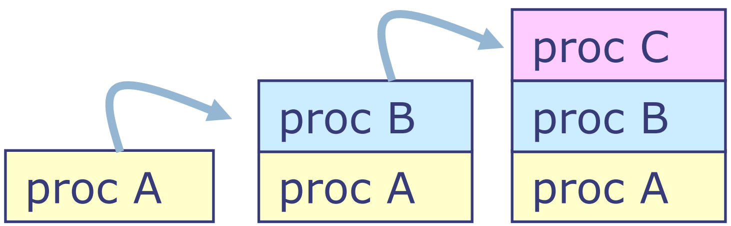
- We only need to access to the activation record of the currently executing procedure

Insight (ca. 1960): We Need a Stack!

- Need data structure to hold activation records

- Activation records are allocated and deallocated in last-in-first-out (LIFO) order

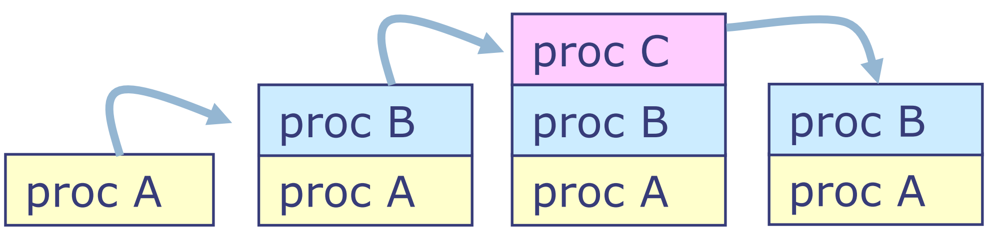
- Stack: push, pop, access to top element



- We only need to access to the activation record of the currently executing procedure

Insight (ca. 1960): We Need a Stack!

- Need data structure to hold activation records
- Activation records are allocated and deallocated in last-in-first-out (LIFO) order
- Stack: push, pop, access to top element



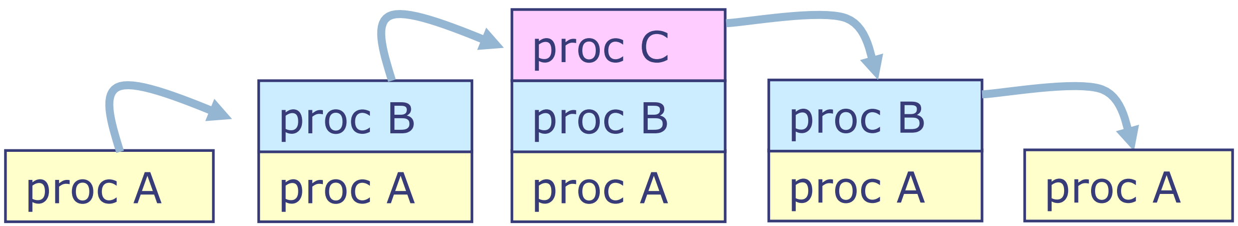
- We only need to access to the activation record of the currently executing procedure

Insight (ca. 1960): We Need a Stack!

- Need data structure to hold activation records

- Activation records are allocated and deallocated in last-in-first-out (LIFO) order

- Stack: push, pop, access to top element

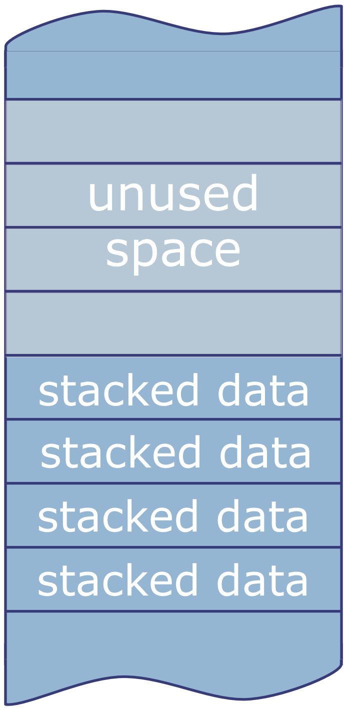


- We only need to access to the activation record of the currently executing procedure

RISC-V Stack

- Stack is in memory → need a register to point to it
  - In RISC-V, stack pointer `sp` is `x2`

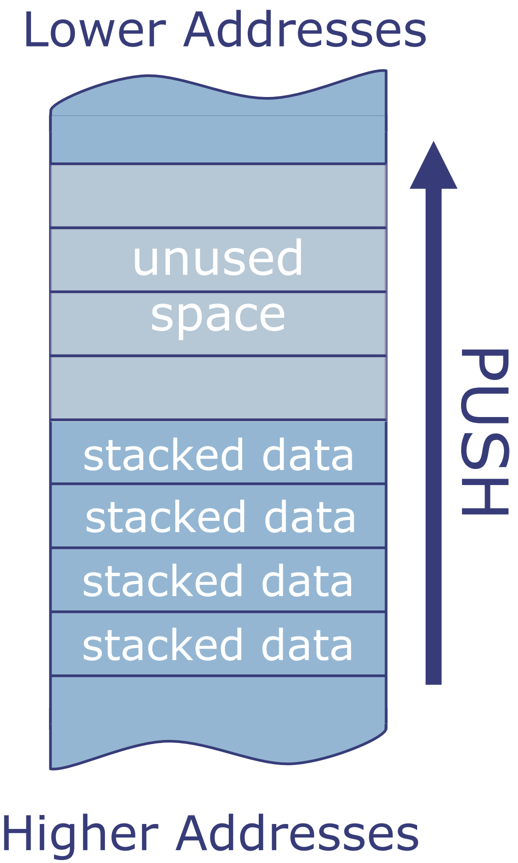
Lower Addresses



Higher Addresses

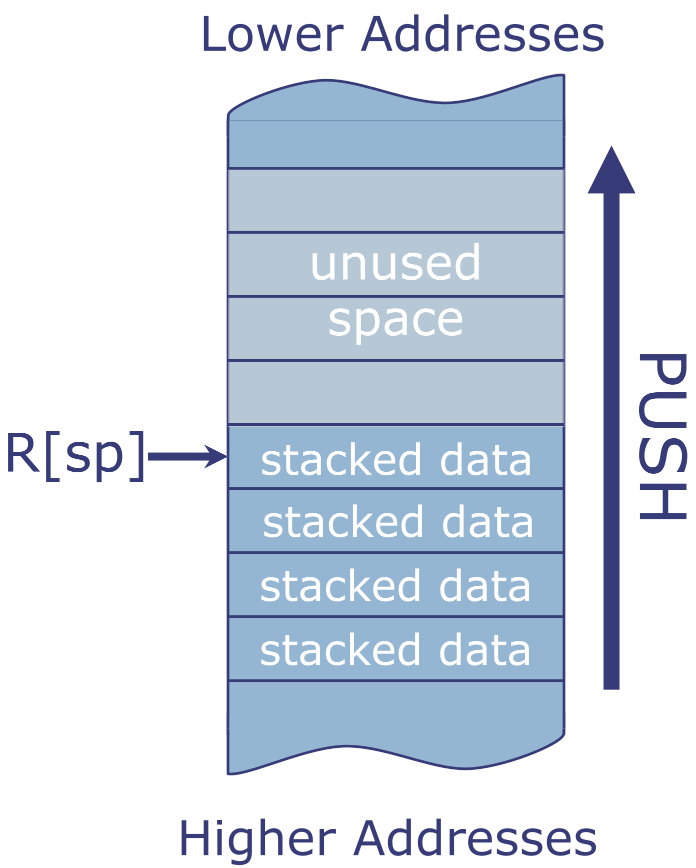
RISC-V Stack

- Stack is in memory → need a register to point to it
  - In RISC-V, stack pointer `sp` is `x2`
- Stack grows down from higher to lower addresses
  - Push decreases `sp`
  - Pop increases `sp`



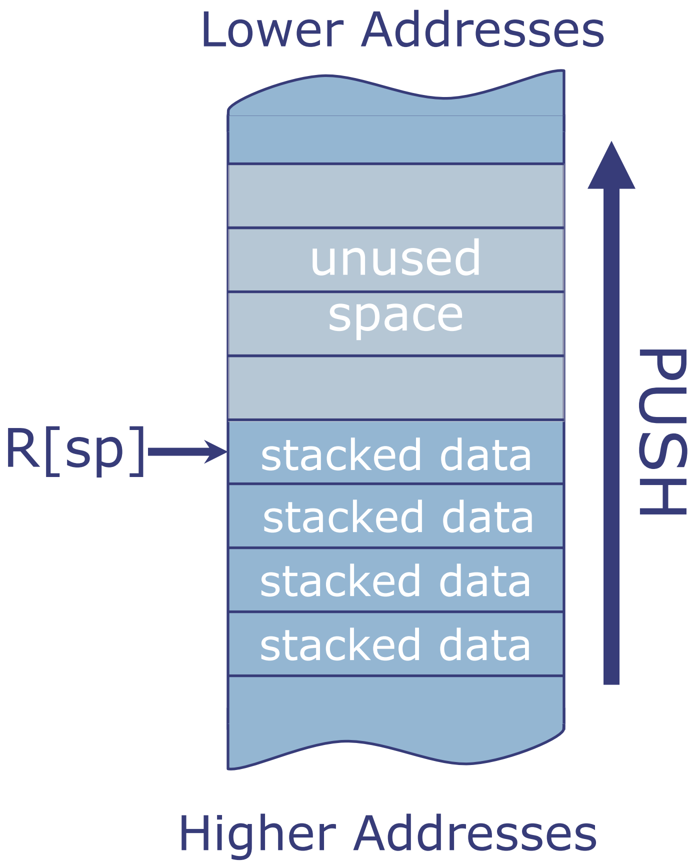
RISC-V Stack

- Stack is in memory → need a register to point to it
  - In RISC-V, stack pointer sp is x2
- Stack grows down from higher to lower addresses
  - Push decreases sp
  - Pop increases sp
- sp points to top of stack (last pushed element)



RISC-V Stack

- Stack is in memory → need a register to point to it
  - In RISC-V, stack pointer `sp` is `x2`
- Stack grows down from higher to lower addresses
  - Push decreases `sp`
  - Pop increases `sp`
- `sp` points to top of stack (last pushed element)
- Discipline: Can use stack *at any time*, but leave it as you found it!



Using the stack

Using the stack

Sample entry sequence

```
addi sp, sp, -8
sw ra, 0(sp)
sw a0, 4(sp)
```

Using the stack

Sample entry sequence

```
addi sp, sp, -8
sw ra, 0(sp)
sw a0, 4(sp)
```

Corresponding Exit sequence

```
lw ra, 0(sp)
lw a0, 4(sp)
addi sp, sp, 8
```

Calling Convention

- The calling convention specifies rules for register usage across procedures

Calling Convention

- The calling convention specifies rules for register usage across procedures
- RISC-V calling convention gives symbolic names to registers x0-x31 to denote their role:

Calling Convention

- The calling convention specifies rules for register usage across procedures
- RISC-V calling convention gives symbolic names to registers x0-x31 to denote their role:

Symbolic name	Registers	Description	Saver
a0 to a7	x10 to x17	Function arguments	Caller
a0 and a1	x10 and x11	Function return values	Caller
ra	x1	Return address	Caller

Calling Convention

- The calling convention specifies rules for register usage across procedures
- RISC-V calling convention gives symbolic names to registers x0-x31 to denote their role:

Symbolic name	Registers	Description	Saver
a0 to a7	x10 to x17	Function arguments	Caller
a0 and a1	x10 and x11	Function return values	Caller
ra	x1	Return address	Caller
t0 to t6	x5-7, x28-31	Temporaries	Caller
s0 to s11	x8-9, x18-27	Saved registers	Callee

Calling Convention

- The calling convention specifies rules for register usage across procedures
- RISC-V calling convention gives symbolic names to registers x0-x31 to denote their role:

Symbolic name	Registers	Description	Saver
a0 to a7	x10 to x17	Function arguments	Caller
a0 and a1	x10 and x11	Function return values	Caller
ra	x1	Return address	Caller
t0 to t6	x5-7, x28-31	Temporaries	Caller
s0 to s11	x8-9, x18-27	Saved registers	Callee
sp	x2	Stack pointer	Callee
gp	x3	Global pointer	---
tp	x4	Thread pointer	---

Calling Convention

- The calling convention specifies rules for register usage across procedures
- RISC-V calling convention gives symbolic names to registers x0-x31 to denote their role:

Symbolic name	Registers	Description	Saver
a0 to a7	x10 to x17	Function arguments	Caller
a0 and a1	x10 and x11	Function return values	Caller
ra	x1	Return address	Caller
t0 to t6	x5-7, x28-31	Temporaries	Caller
s0 to s11	x8-9, x18-27	Saved registers	Callee
sp	x2	Stack pointer	Callee
gp	x3	Global pointer	---
tp	x4	Thread pointer	---
zero	x0	Hardwired zero	---

Caller-Saved vs Callee-Saved Registers

- A caller-saved register is not preserved across function calls (callee can overwrite it)
  - If caller wants to preserve its value, it must save it on the stack before transferring control to the callee

Caller-Saved vs Callee-Saved Registers

- A caller-saved register is not preserved across function calls (callee can overwrite it)
  - If caller wants to preserve its value, it must save it on the stack before transferring control to the callee
  - argument registers (aN), return address (ra), and temporary registers (tN)

Caller-Saved vs Callee-Saved Registers

- A caller-saved register is not preserved across function calls (callee can overwrite it)
  - If caller wants to preserve its value, it must save it on the stack before transferring control to the callee
  - argument registers (aN), return address (ra), and temporary registers (tN)

- A callee-saved register is preserved across function calls
  - If callee wants to use it, it must save its value on stack and restore it before returning control to the caller

Caller-Saved vs Callee-Saved Registers

- A caller-saved register is not preserved across function calls (callee can overwrite it)
  - If caller wants to preserve its value, it must save it on the stack before transferring control to the callee
  - argument registers (aN), return address (ra), and temporary registers (tN)

- A callee-saved register is preserved across function calls
  - If callee wants to use it, it must save its value on stack and restore it before returning control to the caller
  - Saved registers (sN), stack pointer (sp)

Example: Using callee-saved registers

- Implement `f` using `s0` and `s1` to store temporary values

```
int f(int x, int y) {
    return (x + 3) | (y + 123456);
}
```

Example: Using callee-saved registers

- Implement `f` using `s0` and `s1` to store temporary values

```
int f(int x, int y) {
    return (x + 3) | (y + 123456);
}
```

```
f:




    ret
```

Example: Using callee-saved registers

- Implement `f` using `s0` and `s1` to store temporary values

```
int f(int x, int y) {
    return (x + 3) | (y + 123456);
}
```

```
f:



    addi s0, a0, 3
    li s1, 123456
    add s1, a1, s1
    or a0, s0, s1



    ret
```

Example: Using callee-saved registers

- Implement `f` using `s0` and `s1` to store temporary values

```
int f(int x, int y) {
     return (x + 3) | (y + 123456);
}
```

```
f:



    addi s0, a0, 3
    li s1, 123456
    add s1, a1, s1
    or a0, s0, s1



    ret
```

Example: Using callee-saved registers

- Implement `f` using `s0` and `s1` to store temporary values

```
int f(int x, int y) {
    return (x + 3) | (y + 123456);
}
```

```
f:
  addi sp, sp, -8   // allocate 2 words (8 bytes) on stack
  sw s0, 4(sp)      // save s0
  sw s1, 0(sp)      // save s1
  addi s0, a0, 3
  li s1, 123456
  add s1, a1, s1
  or a0, s0, s1



  ret
```

Example: Using callee-saved registers

- Implement f using s0 and s1 to store temporary values

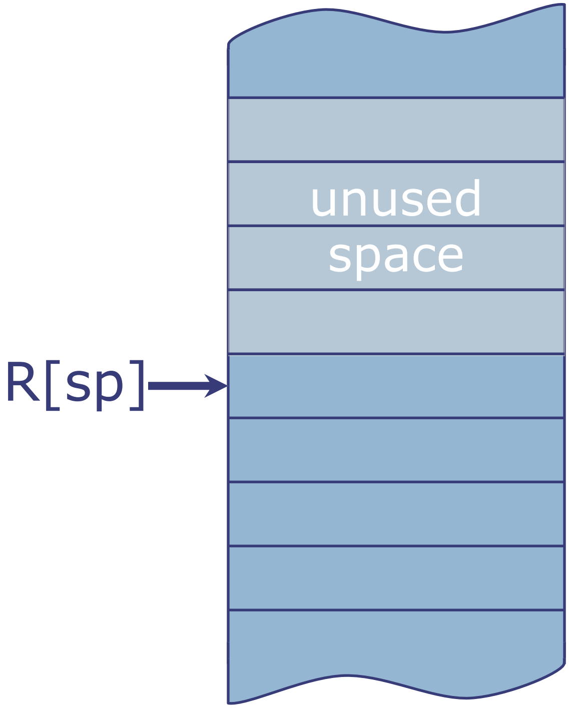
```
int f(int x, int y) {
     return (x + 3) | (y + 123456);
}
```

```
f:
  addi sp, sp, -8   // allocate 2 words (8 bytes) on stack
  sw s0, 4(sp)      // save s0
  sw s1, 0(sp)      // save s1
  addi s0, a0, 3
  li s1, 123456
  add s1, a1, s1
  or a0, s0, s1
  lw s1, 0(sp)      // restore s1
  lw s0, 4(sp)      // restore s0
  addi sp, sp, 8    // deallocate 2 words from stack
                    // (restore sp)
  ret
```

Example: Using callee-saved registers

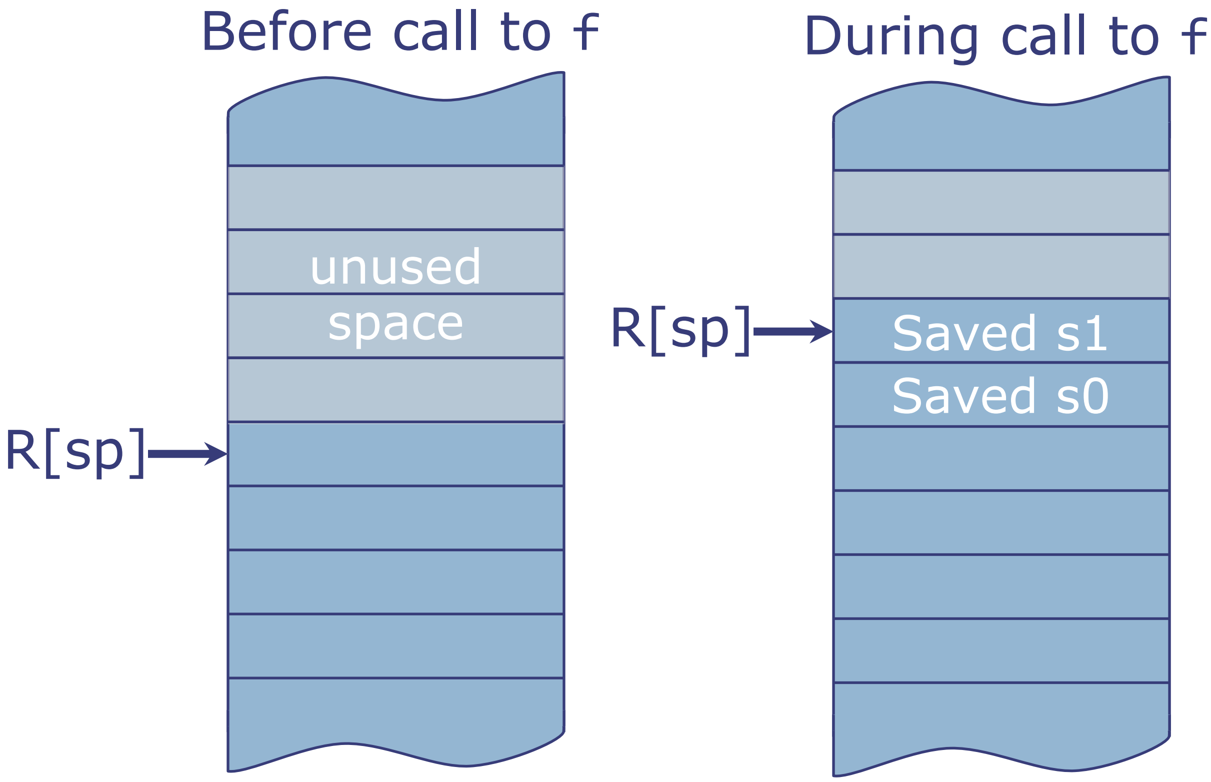
- Stack contents:

Before call to `f`



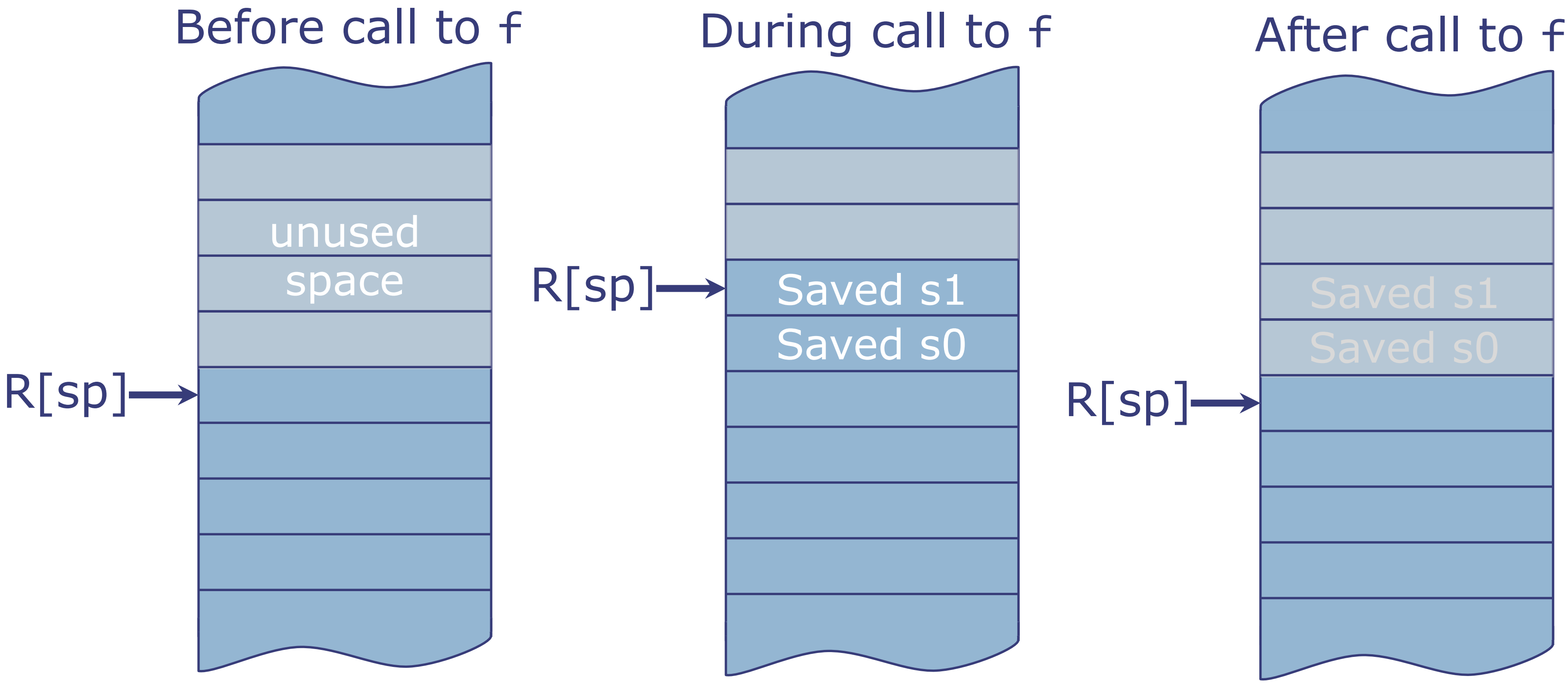
Example: Using callee-saved registers

- Stack contents:



Example: Using callee-saved registers

- Stack contents:



Example: Using caller-saved registers

Caller

```
int x = 1;
int y = 2;
int z = sum(x, y);
int w = sum(z, y);
```

Callee

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

Example: Using caller-saved registers

Caller

```
int x = 1;
int y = 2;
int z = sum(x, y);
int w = sum(z, y);

li a0, 1
li a1, 2




jal ra, sum
// a0 = sum(x, y)
```

Callee

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

Example: Using caller-saved registers

Caller

```
int x = 1;
int y = 2;
int z = sum(x, y);
int w = sum(z, y);
```

```
li a0, 1
li a1, 2



jal ra, sum
// a0 = sum(x, y)
```

Callee

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

```
sum:
  add a0, a0, a1
  ret
```

Example: Using caller-saved registers

Caller

```
int x = 1;
int y = 2;
int z = sum(x, y);
int w = sum(z, y);

li a0, 1
li a1, 2
addi sp, sp, -8
sw ra, 0(sp)
sw a1, 4(sp) // save y
jal ra, sum
// a0 = sum(x, y)
```

Callee

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

sum:
  add a0, a0, a1
  ret
```

Example: Using caller-saved registers

Caller

```
int x = 1;
int y = 2;
int z = sum(x, y);
int w = sum(z, y);
```

```
li a0, 1
li a1, 2
addi sp, sp, -8
sw ra, 0(sp)
sw a1, 4(sp) // save y
jal ra, sum
// a0 = sum(x, y)
```

Callee

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

```
sum:
  add a0, a0, a1
  ret
```

Why did we save a1?

Example: Using caller-saved registers

Caller

```
int x = 1;
int y = 2;
int z = sum(x, y);
int w = sum(z, y);
```

```
li a0, 1
li a1, 2
addi sp, sp, -8
sw ra, 0(sp)
sw a1, 4(sp) // save y
jal ra, sum
// a0 = sum(x, y)
```

Callee

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

```
sum:
  add a0, a0, a1
  ret
```

Why did we save a1?

Callee may have modified a1 (caller doesn't see implementation of sum!)

Example: Using caller-saved registers

Caller

```
int x = 1;
int y = 2;
int z = sum(x, y);
int w = sum(z, y);
```

```
li a0, 1
li a1, 2
addi sp, sp, -8
sw ra, 0(sp)
sw a1, 4(sp) // save y
jal ra, sum
// a0 = sum(x, y)
lw a1, 4(sp) // restore y
```

Callee

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

```
sum:
  add a0, a0, a1
  ret
```

Why did we save a1?

Callee may have modified a1 (caller doesn't see implementation of sum!)

Example: Using caller-saved registers

Caller

```c
int x = 1;
int y = 2;
int z = sum(x, y);
int w = sum(z, y);
```

```
li a0, 1
li a1, 2
addi sp, sp, -8
sw ra, 0(sp)
sw a1, 4(sp) // save y
jal ra, sum
// a0 = sum(x, y)
lw a1, 4(sp) // restore y
jal ra, sum
// a0 = sum(z, y)
```

Callee

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

```
sum:
  add a0, a0, a1
  ret
```

Why did we save a1?

Callee may have modified a1 (caller doesn't see implementation of sum!)

Example: Using caller-saved registers

Caller

```
int x = 1;
int y = 2;
int z = sum(x, y);
int w = sum(z, y);
```

```
li a0, 1
li a1, 2
addi sp, sp, -8
sw ra, 0(sp)
sw a1, 4(sp) // save y
jal ra, sum
// a0 = sum(x, y)
lw a1, 4(sp) // restore y
jal ra, sum
// a0 = sum(z, y)
lw ra, 0(sp)
addi sp, sp, 8
```

Callee

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

```
sum:
  add a0, a0, a1
  ret
```

Why did we save a1?

Callee may have modified a1 (caller doesn't see implementation of sum!)

Thank you!

Next lecture:
More Procedures and MMIO