Lab3: Machine Language and Assembly Programming

Goal

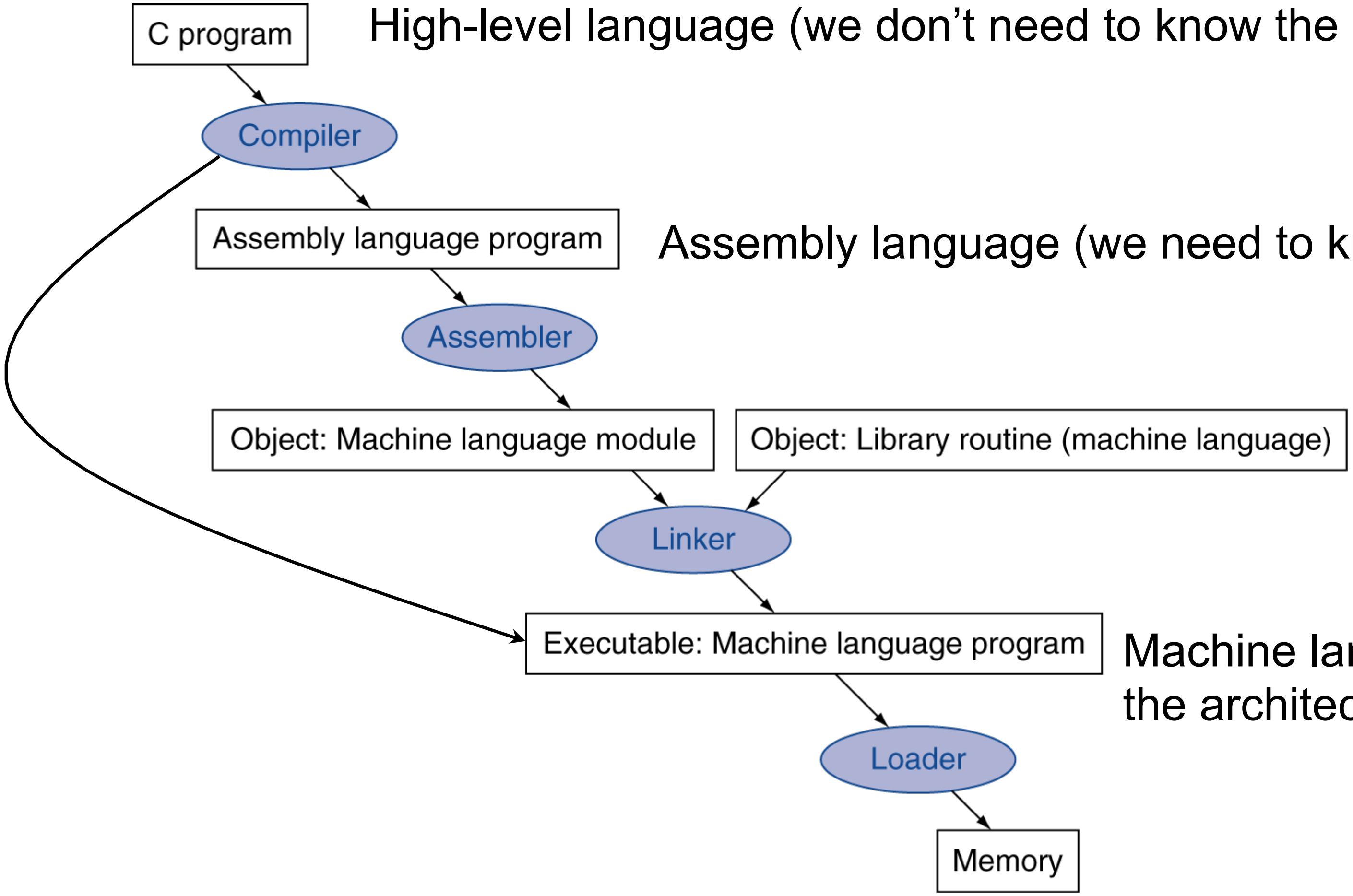
- Learn how instructions are executed
- Learn how registers are used
- Write subroutines in assembly language
- Learn how to pass and return arguments from subroutines
- Learn how the stack is used

Programmers vs. computers

- Programmers can write programs in a high-level programming language, or assembly language

- Computers can only execute programs written in their own native language (machine code)

Programming



High-level language (we don't need to know the architecture)

Assembly language (we need to know the architecture)

Machine language (native to the architecture)

Example

- `a=b+c;`
  1. Load variable `b` from memory into `register1`
  2. Load variable `c` from memory into `register2`
  3. Perform the addition `register1+register2` and store the result in `register3`
  4. Store `register3` to the memory address of variable `a`
- Each step translates into one machine instruction

Machine Language

- Processor can only execute machine instructions
- The instructions reside in memory along with data
- Machine instruction is a sequence of bits

00001101010111101110

Opcode Op1 Op2 Op3

- There is a set of machine instructions that are supported by a given computer architecture (Instruction Set)

Maskininstruktioner

Lectures

- Definitioner:
  - Vad ska göras (operationskod)?
  - Vem är inblandad (source operander)?
  - Vart ska resultatet (destination operand)?
  - Hur fortsätta efter instruktionen?

Maskininstruktioner

Lectures

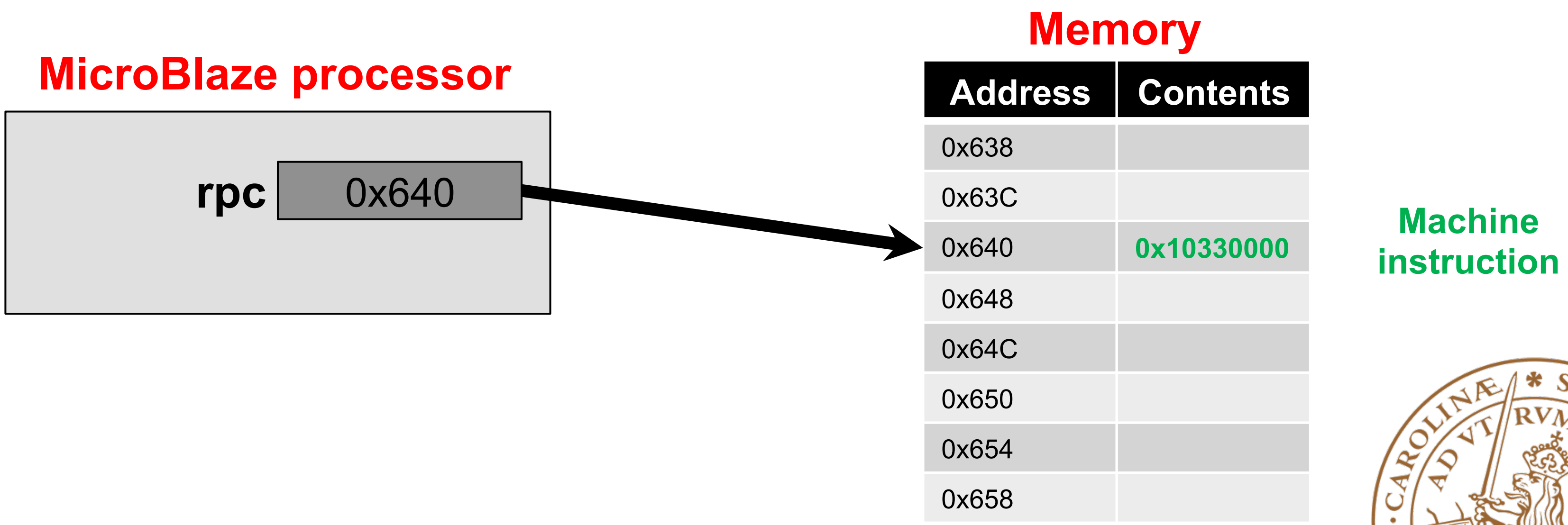
- Att bestämma:
  - Typ av operander och operationer
  - Antal adresser och adresseringsformat
  - Registeraccess
  - Instruktionsformat
    - Fixed eller flexibelt

Inside the MicroBlaze processor

- Thirty two 32-bit general purpose registers, `r0-r31`
- `r0` is a read-only register containing the value `0`
- A set of special purpose registers
  - **rpc,** Program Counter
    - keeps the address of the instruction being executed
    - special purpose register `0`
    - can be read with an **`MFS`** instruction
  - **rmsr,** Machine Status Register
    - contains control and status bits for the processor
    - special purpose register `1`
    - can be accessed with both **`MFS`** and **`MTS`** instructions

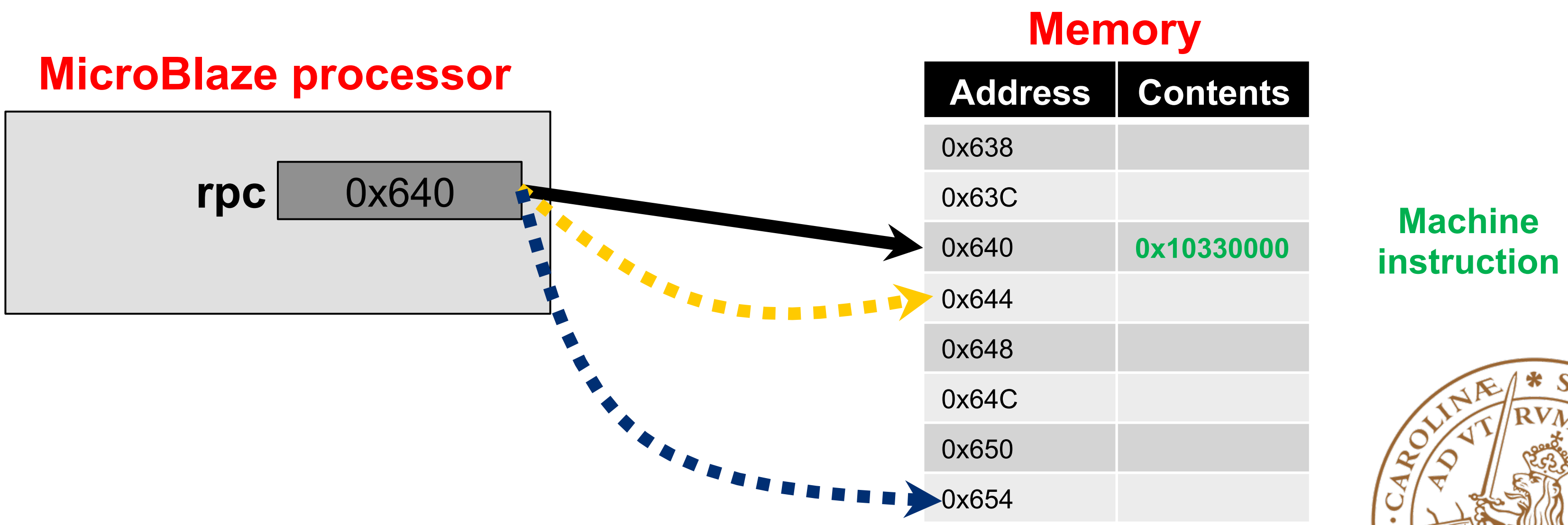
Program Counter (rpc)

- Contains the memory address of the instruction that is to be fetched and executed by the processor
- After the execution of the current instruction, this register is updated to point to the memory address of the next instruction that should be fetched and executed



Program Counter (rpc)

- If the current instruction does not explicitly modify **rpc,** after execution, the **rpc** is updated to point to the successive memory address (the yellow arrow)
- If the current instruction explicitly modifies **rpc,** after execution, the **rpc** points to the new value assigned to it (blue arrow)



MicroBlaze machine instructions

- Fixed size (all instructions have the same size)
- Operands are provided through general purpose registers or immediate values that are encoded in the instruction itself
- Two instruction formats

Type A	**Opcode**	**Rd**	**Ra**	**Rb**	**func**
Type B	**Opcode**	**Rd**	**Ra**	**Immediate**	

- Opcode- operation code (encoded with 6 bits)
- Rd- destination register (encoded with 5 bits)
- Ra, Rb- source registers (each encoded with 5 bits)
- Immediate- 16bit value (signed extended to 32 bits unless an IMM instruction is used before)

Type A instruction- Example

- Logical AND
  - Syntax:

    ```
    AND  Rd, Ra, Rb
    ```

  - Description:

    ```
    Rd = Ra & Rb
    ```

  - Machine code

100001	R_{d4}-R_{d0}	R_{a4}-R_{a0}	R_{b4}-R_{b0}	00000000000

  - Machine code example:

    100001 01010 01010 01110 00000000000

    ```
    R10 = R10 & R14
    ```

Type B Instruction - Example

- Logical AND
  - Syntax:

```
ANDI  Rd, Ra, Imm
```

  - Description:

```
Rd = Ra & signExtend32(Imm)
```

  - Machine code

101001	R_{d4}-R_{d0}	R_{a4}-R_{a0}	Imm_{15}-Imm_{0}

  - Machine code example:

101001 01010 01010 1111000000000000

```
R10 = R10 & 0B11111111111111111111000000000000
```

Can be overwritten by a preceding IMM instruction

MicroBlaze Instruction Set

- Arithmetic Instructions
- Logic Instructions
- Branch Instructions
- Memory Access Instructions
- Other

Arithmetic instructions – Type A

Type A	
ADD Rd, Ra, Rb *add*	Rd=Ra+Rb, Carry flag affected
ADDK Rd, Ra, Rb *add and keep carry*	Rd=Ra+Rb, Carry flag not affected
RSUB Rd, Ra, Rb *reverse subtract*	**Rd=Rb-Ra**, Carry flag affected

Arithmetic instructions – Type B

Type B	
ADDI Rd, Ra, Imm *add immediate*	Rd=Ra+signExtend32(Imm)**
ADDIK Rd, Ra, Imm *add immediate and keep carry*	Rd=Ra+signExtend32(Imm)**
RSUBIK Rd, Ra, Imm *reverse subtract with immediate*	**Rd=signExtend32(Imm)**-Ra**
SRA Rd, Ra *arithmetic shift right*	Rd=(Ra>>1)

- `Imm` field is a 16 bit value that is sign extend to 32 bits
- To use 32 bit immediate value a Type B instruction can be preceded by an **`IMM`** instruction which overwrites the sign extension

Logic instructions – Type A

Type A	
OR Rd, Ra, Rb *logical or*	Rd=Ra | Rb
AND Rd, Ra, Rb *logical add*	Rd=Ra & Rb
XOR Rd, Ra, Rb *logical xor*	Rd=Rb ^ Ra
ANDN Rd, Ra, Rb *logical and not*	Rd=Ra & (~Rb)

Logic instructions – Type B

Type B	
ORI Rd, Ra, Imm *logical OR with immediate*	Rd=Ra | signExtend32(Imm)
ANDI Rd, Ra, Imm *logical AND with immediate*	Rd=Ra & signExtend32(Imm)
XORI Rd, Ra, Imm *logical XOR with immediate*	Rd=Ra ^ signExtend32(Imm)
ANDNI Rd, Ra, Imm *logical AND NOT with immediate*	Rd=Ra & (~signExtend32(Imm))

- `Imm` field is a 16 bit value that is sign extend to 32 bits
- To use 32 bit immediate value a Type B instruction can be preceded by an **`IMM`** instruction which overwrites the sign extension

Branch Instructions- Unconditional

Modify the Program Counter (PC) register

Type B	
BRID Imm *branch immediate with delay*	PC=PC+signExtend32(Imm) allow delay slot execution
BRLID Rd, Imm *branch and link immediate with delay (function call)*	Rd=PC PC=PC+signExtend32(Imm) allow delay slot execution
RTSD Ra, Imm *return from subroutine*	PC=Ra+signExtend32(Imm) allow delay slot execution
RTID Ra, Imm *return from interrupt*	PC=Ra+signExtend32(Imm) allow delay slot execution set interrupt enable in MSR

- `Imm` field is a 16 bit value that is sign extend to 32 bits
- To use 32 bit immediate value a Type B instruction can be preceded by an **`IMM`** instruction which overwrites the sign extension

Branch Instructions- Unconditional

Modify the Program Counter (PC) register

Type B	
`BRID` `Imm` *`branch immediate with delay`*	`PC=PC+signExtend32(Imm)` `allow delay slot execution`
`BRLID` `Rd, Imm` *`branch and link immediate`* *`with delay (function call)`*	`Rd=PC` `PC=PC+signExtend32(Imm)` `allow delay slot execution`
`RTSD` `Ra, Imm` *`return from subroutine`*	`PC= a+signExtend32(Imm)` `allc delay slot execution`
`RTID` `Ra, Imm` *`return from interrupt`*	`PC R nExtend32(Imm)` `a ution` `s in MSR`

Call a function

- `Imm` field is a 16 bit value that is sign
- To use 32 bit immediate value a Type B instruction can be preceded by an **`IMM`** instruction which overwrites the sign extension

Branch Instructions- Unconditional

Modify the Program Counter (PC) register

Type B	
BRID Imm *branch immediate with delay*	PC=PC+signExtend32(Imm) allow delay slot execution
BRLID Rd, Imm *branch and link immediate with delay (function call)*	Rd=PC PC=PC+signExtend32(Imm) allow delay slot execution
RTSD Ra, Imm *return from subroutine*	PC=Ra+signExtend32(Imm) allow delay slot execution
RTID Ra, Imm *return from interrupt*	PC=Ra+signExtend32(Imm) …cution s… in MSR

Return from a function

- `Imm` field is a 16 bit value that is sign
- To use 32 bit immediate value a Type B instruction can be preceded by an **`IMM`** instruction which overwrites the sign extension

Branch Instructions- Conditional (1)

Modify the Program Counter (PC) register if a condition is satisfied

Type B	
BEQI Ra, Imm *branch if equal*	if Ra==0 PC=PC+signExtend32(Imm)
BNEI Ra, Imm *branch if not equal*	if Ra!=0 PC=PC+signExtend32(Imm)

- `Imm` field is a 16 bit value that is sign extend to 32 bits
- To use 32 bit immediate value a Type B instruction can be preceded by an **`IMM`** instruction
- Branch instructions that allow execution of the instruction in the branch delay slot are available, **`BEQID`** and **`BNEID`**

Branch Instructions- Conditional (2)

Modify the Program Counter (PC) register if a condition is satisfied

Type B	
BLTI Ra, Imm *branch if lower than*	`if Ra<0` `PC=PC+signExtend32(Imm)`
BLEI Ra, Imm *branch if lower equal than*	`if Ra<=0` `PC=PC+signExtend32(Imm)`
BGTI Ra, Imm *branch if greater than*	`if Ra>0` `PC=PC+signExtend32(Imm)`
BGEI Ra, Imm *branch if greater equal than*	`if Ra>=0` `PC=PC+signExtend32(Imm)`

- `Imm` field is a 16 bit value that is sign extend to 32 bits
- To use 32 bit immediate value a Type B instruction can be preceded by an **`IMM`** instruction
- Branch instructions that allow execution of the instruction in the branch delay slot are available (**`D`** is appended to the mnemonic)

Allow/Disallow Branch Delay Slot Execution

Disallow

```
      addi r1,r0,10
      bgti r1,done
      addi r1,r1,-1
      add  r2,r2,r0
done: add  r2,r1,r0
```

r1=r0+10=10

r2=r0+r1=r1

Allow

```
      addi  r1,r0,10
      bgtid r1,done
      addi  r1,r1,-1
      add   r2,r2,r0
done: add   r2,r1,r0
```

r2=10

r2=9

Allowing branch delay slot exeuction is usefull in pipeling
(see Lecture on pipelining)

Memory Access Instructions

Processor reads from a given memory address

Type A	
`LW` `Rd, Ra, Rb` *load word*	`Address=Ra+Rb` `Rd=*Address`
Type B	
`LWI` `Rd, Ra, Imm` *load word immediate*	`Address=Ra+signExtend32(Imm)` `Rd=*Address`

- `Imm` field is a 16 bit value that is sign extend to 32 bits
- To use 32 bit immediate value a Type B instruction can be preceded by an **`IMM`** instruction

Memory Access Instructions

Processor writes to a given memory address

Type A	
SW Rd, Ra, Rb *store word*	Address=Ra+Rb *Address=Rd
Type B	
SWI Rd, Ra, Imm *store word immediate*	Address=Ra+signExtend32(Imm) *Address=Rd

- `Imm` field is a 16 bit value that is sign extend to 32 bits
- To use 32 bit immediate value a Type B instruction can be preceded by an **`IMM`** instruction

Other Instructions

IMM Imm *immediate*	Overwrites the sign extension to 32 bits of the following Type B instruction. The Imm field provides the 16 upper bits of the 32 bit immediate value later used by the Type B instruction
MFS Rd,Sa *move from special purpose register*	Rd=Sa Sa- special purpose register, source operand
MTS Sd,Ra *move to special purpose register*	Sd=Ra Sd- special purpose register, destination operand
NOP *No operation*	

IMM instruction- Example

```
IMM   0x5002
ANDI r10,r10,0xF000
```
(Type B instruction preceded by IMM)

```
ANDI r10,r10,OXF000
```
(Type B instruction not preceded by IMM)

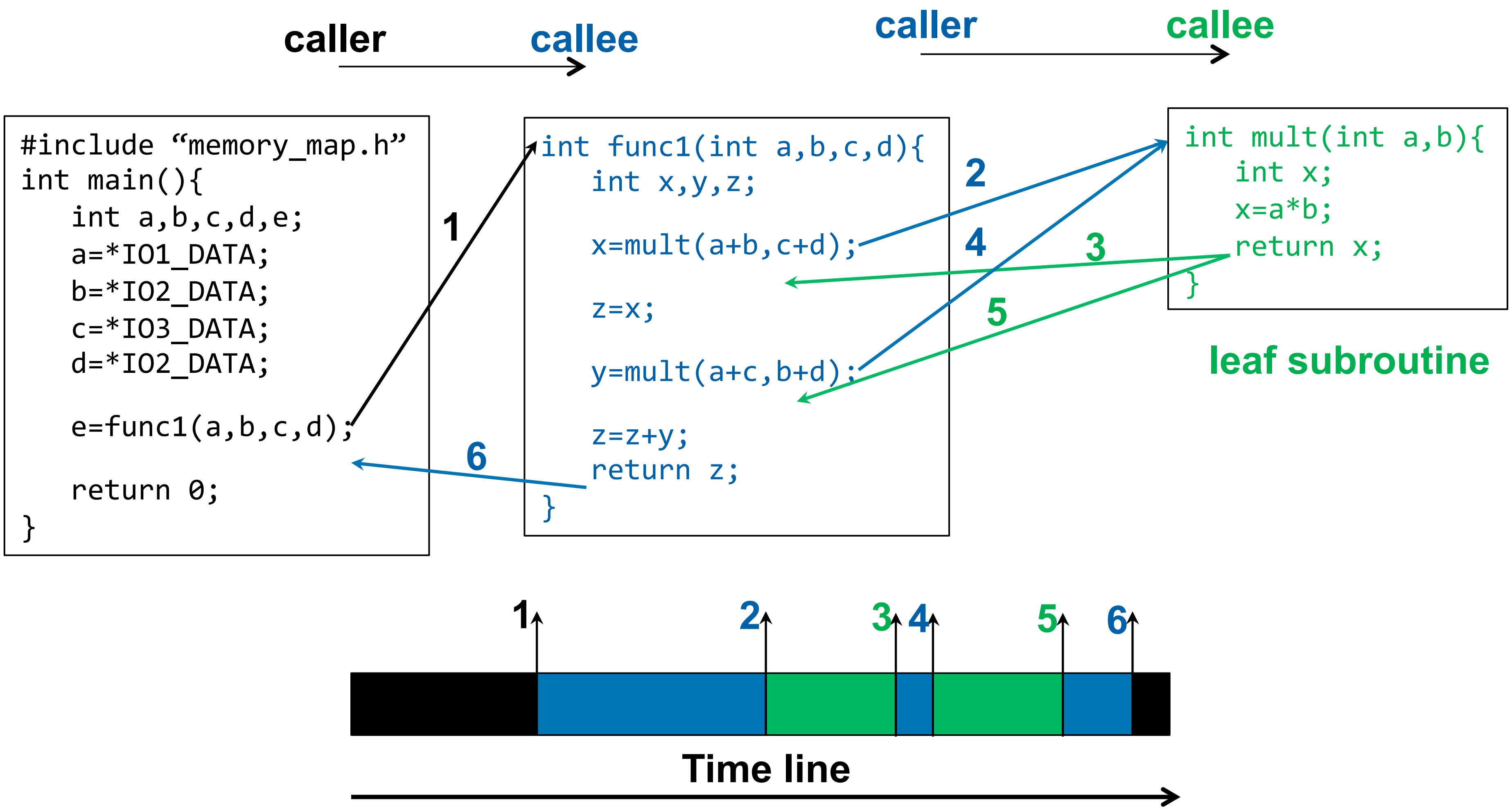
```
r10=r10 & 0x5002F000
```

- `5002` → `Imm` **field from the** `IMM` **instruction**
- `F000` → `Imm` **field from the** `ANDI` **instruction**

```
r10=r10 & 0xFFFFF000
```

- `FFFF` → **Sign extension to 32 bits**
- `F000` → `Imm` **field from the** `ANDI` **instruction**

Functions (subroutines)



Functions (subroutines)

- Caller
  - Prepare input arguments and pass them to the callee
  - Provide a return address to the callee
- Callee
  - Provide return values (outputs)
  - Ensure that the caller can seamlessly proceed, once the callee returns to the caller

Functions (subroutines) - problems

- How to pass arguments to functions?
- How to return values from functions?
  - **FOLLOW A REGISTER USAGE CONVENTION**
- How to ensure that registers retain values across function calls?
- Where to return after a function has been executed?
- Where to store temporary local variables of a function?
  - **USE THE STACK**

Register Usage Convention

- Dedicated
  - dedicated usage
- Volatile
  - Do not retain values across function calls
  - Store temporary results
  - Passing parameters/ Return values
- Non-volatile
  - Must be saved across function calls
  - Saved by callee

Register Usage Convention

Dedicated	
`r0`	Keeps value zero
`r1`	Stack pointer
`r14`	Return address for interrupts
`r15`	Return address for subroutines
`r18`	Assembler temporary
Volatile	
`r3-r4`	Return values/ Temporaries
`r5-r10`	Passing parameters/Temporaries
`r11-r12`	Temporaries
Non-volatile	
`r19-r31`	Saved across function calls

Stack

- Memory segment
- Grows towards lower memory addresses
- Access the stack through a stack pointer
- Stack pointer points to the top of the stack
- Two operations
  - PUSH an item on top of the stack
  - POP the top item from the stack

Memory	
0x00	
0x01	
0x02	
0x03	
…	
0xFFC	
0xFFD	
0xFFE	
0xFFF	

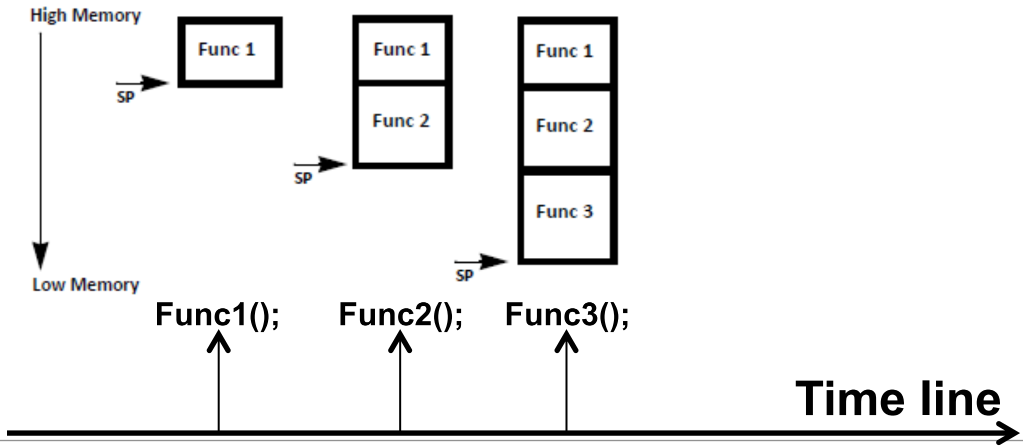
Stack pointer → 0xFFF

Stack frame

- Temporal storage for the function to do its own book-keeping
- Items inside a stack frame include:
  - Return address
  - Local variables used by the function
  - Save registers that the function may modify, but the caller function does not want changed
  - Input arguments to callee functions

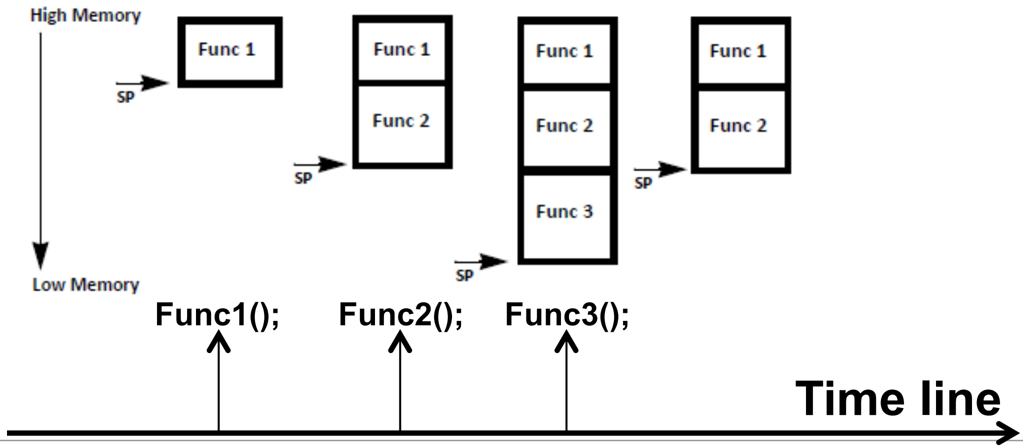
When a function is called…

- Reserve space on the stack for the stack frame
  - Decrement the stack pointer
- Store necessary information in the stack frame
  - Return address
  - Non-volatile registers
- Store input arguments provided through registers, in the caller's stack frame



When a function returns…

- Load necessary information from the stack frame and restore registers
  - Return address
  - Non-volatile registers
- Pop the stack frame from the stack
  - Increment the stack pointer
- Return to the caller



Stack Frame Convention

- How are items in the stack frame organized?

Stack frame top	Return address
	Input arguments to callee function
	Local variables
Stack frame bottom	Saved registers

- Stack pointer (register **`r1`**) points to the top of the latest stack frame

Stack Frame – Return address

- Stack frame always reserves space for a return address
- If the function calls other functions, register **`r15`** is stored in the return address field of the stack frame

Stack Frame – Input arguments

- Present only in stack frames of functions which call other functions that require input arguments
- If present, this field reserves space for at least 6 Words, i.e. for registers **`r5-r10`**
- If the function calls a callee that needs more than 6 arguments, the first six arguments to the callee are provided through registers **`r5-r10`**, while the rest of the arguments will be stored in the input arguments field of the stack frame
- Input arguments field of a stack frame is accessed by the callee function

Stack Frame – Local Variables

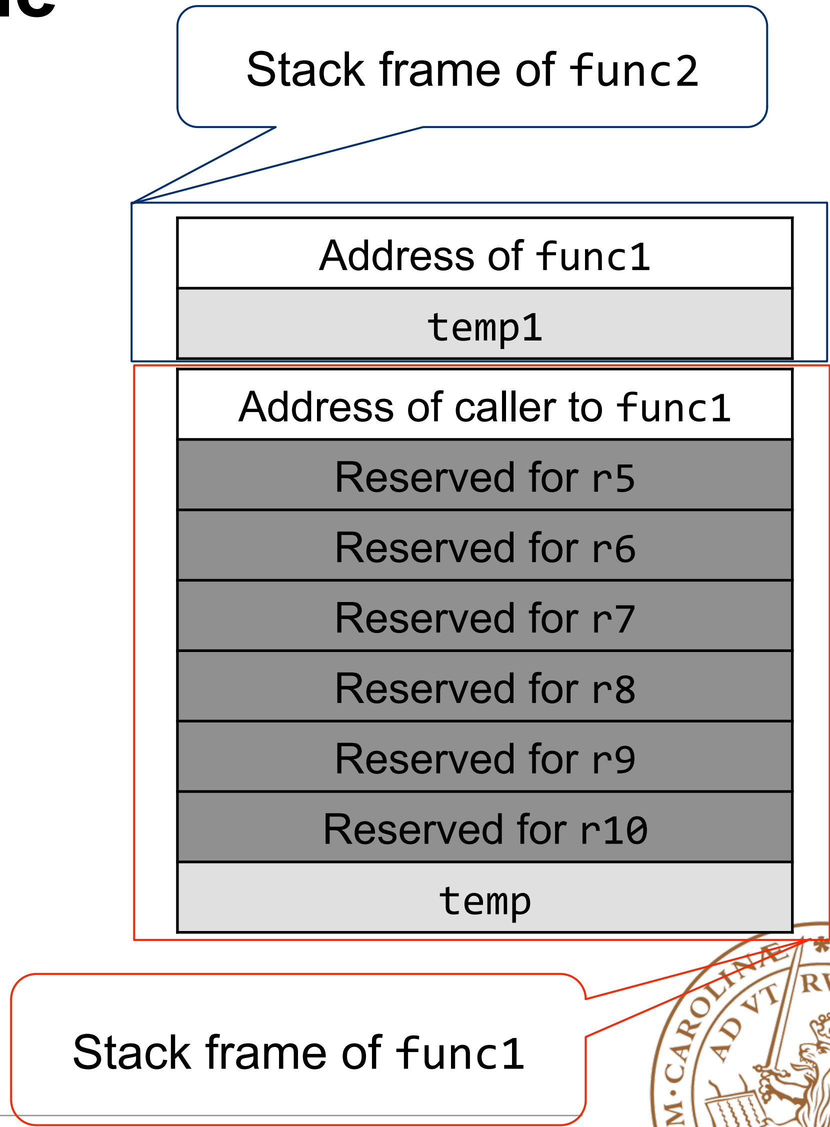
- Present only in stack frames of functions which contain local variables
- The size depends on the number of defined local variables

Stack Frame – Saved registers

- Present only if the function needs to use any of the non-volatile registers `r19-r31`
- Values of the non-volatile registers `r19-r31` are stored in the saved register field of the stack frame
- Before the function returns, values of the non-volatile registers `r19-r31` are restored from the stack frame
- This way, a callee function ensures that the caller can seamlessly proceed with its execution

Stack Frame – Example

```
int func1(){
int temp;
temp=3;
temp=func2(temp,temp+2);
return temp;
}
int func2(int x, int y){
int temp1;
temp1=x*y
return temp1;
}
```



Assembly program

```
.global number_of_ones
.text
.ent number_of_ones
number_of_ones: add r3,r0,r0
while:          beqid r5, result
                nop
                andi r4,r5,1
                add r3,r3,r4
                sra r5,r5
                brid while
                nop
result:         rtsd r15, 8
                nop
.end number_of_ones
```

Assembly directives
Assembly instructions
Symbols (labels)

use labels for branch instructions

Assembly program

```
.global number_of_ones
.text
.ent number_of_ones
number_of_ones:     add r3,r0,r0
while:              beqid r5, result
                    nop
                    andi r4,r5,1
                    add r3,r3,r4
                    sra r5,r5
                    brid while
                    nop
result:             rtsd r15, 8
                    nop
.end number_of_ones
```

```
unsigned int number_of_ones(unsigned int x){
unsigned int temp=0;// temp is stored in r3
  while (x!=0){
         temp=temp+x&1;
         x>>=1;
  }
  return temp;
}
```

Disassembled program

0x6C0	add r3,r0,r0
0x6C4	beqid r5, 28
0x6C8	or r0,r0,r0
0x6CC	andi r4,r5,1
0x6D0	add r3,r3,r4
0x6D4	sra r5,r5
0x6D8	brid -20
0x6DC	or r0,r0,r0
0x6E0	rtsd r15, 8
0x6E4	or r0,r0,r0

```
.global number_of_ones
.text
.ent number_of_ones
number_of_ones:  add r3,r0,r0
while:           beqid r5, result
                 nop
                 andi r4,r5,1
                 add r3,r3,r4
                 sra r5,r5
                 brid while
                 nop
result:          rtsd r15, 8
                 nop
.end number_of_ones
```

Tips and tricks

- Initialize a register with a known value
  - Example load register `r5` with 150

    ```
    addi r5,r0,150
    ```
- Shift to left
  - Example register `r5` to be shifted one position to left

    ```
    add r5,r5,r5 // r5=r5*2==r5<<1
    ```
  - How about shifting multiple positions to the left?

Tips and tricks

- IF statement

Assume x is stored in `r5`

```
if (x>0){
    block_true
    …
}else{
    block_false
    …
}
y=…
```

```
         blei r5, false
         block_true
         …
         bri end_if
false:   block_false
         …
end_if:  y=…
```

Note the condition is inverted

Tips and tricks

- IF statement

Assume x is stored in r5

```
if (x>0){
    block_true
    …
}else{
    block_false
    …
}
y=…
```

```
         bgti r5, true
         block_false
         …
         bri end_if
true:    block_true
         …
end_if:  y=…
```

Note the blocks are swapped

Tips and tricks

- WHILE loop

Assume x is stored in r5

```
while (x>0){
    block
    …
}
y=…
```

```
condition:   blei r5, while_end
             block
             …
             bri condition
while_end:   y=…
```

Note the condition is inverted

Tips and tricks

- Multiplication
  - Example `r3` stores the product `r5*r6`

```
        add r3,r0,r0
again:  beqi r6, done
        add r3,r3,r5
        addi r6,r6-1
        bri again
done:   nop
```

SIGILLVM UNIVERSITATIS GOTHORVM CAROLINÆ
AD VTRVMQVE
1666

LUNDS
UNIVERSITET