Lab3: Machine Language and Assembly Programming

Goal

Learn how

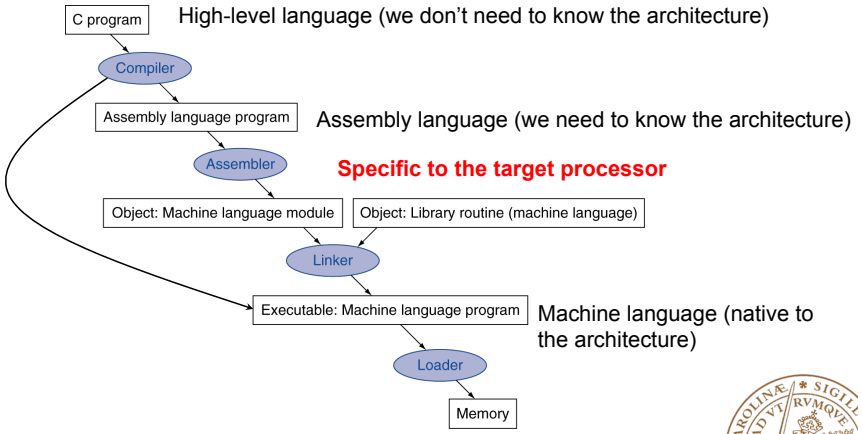
- instructions are executed
- registers are used
- to write subroutines in assembly language
- to pass and return arguments from subroutines
- the stack is used

Programmers vs. computers

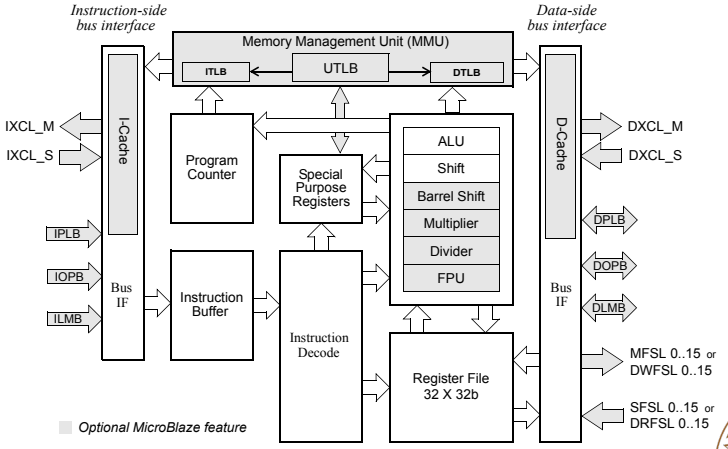
- Programmers can write programs in a high-level programming language, or assembly language

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

Programming



MicroBlaze Architecture



Machine Language

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

00001	101010111101110	011

Opcode Operand (memory) Operand (register)

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

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

Type A	0-5	6-10	11-15	16-20	21-31	Semantics
ADD Rd,Ra,Rb	000000	Rd	Ra	Rb	00000000000	Rd := Rb + Ra

Registers

- 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

Register Usage Conventions

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

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**

General Purpose Registers

Symbol	Description
Ra	R0 - R31, General Purpose Register, source operand a
Rb	R0 - R31, General Purpose Register, source operand b
Rd	R0 - R31, General Purpose Register, destination operand

In total 32 32-bit register are available

Some registers have a special purpose (R14-R17):
Return addresses

- for interrupts
- User vectors
- Breaks
- Hardware exceptions

Register Map

Bits	Name	Description	Reset Value
0:31	R0	Always has a value of zero. Anything written to R0 is discarded	0x00000000
0:31	R1 through R13	32-bit general purpose registers	-
0:31	R14	32-bit register used to store return addresses for interrupts.	-
0:31	R15	32-bit general purpose register. Recommended for storing return addresses for user vectors.	-
0:31	R16	32-bit register used to store return addresses for breaks.	-
0:31	R17	If MicroBlaze is configured to support hardware exceptions, this register is loaded with the address of the instruction following the instruction causing the HW exception, except for exceptions in delay slots that use BTR instead (see "Branch Target Register (BTR)"); if not, it is a general purpose register.	-
0:31	R18 through R31	R18 through R31 are 32-bit general purpose registers.	-

Maskininstruktioner

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

Maskininstruktioner

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

MicroBlaze Instruction Set

- Arithmetic Instructions
- Logic Instructions
- Branch Instructions
- Load/Store 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

Type A: up to **two** source and **one** destination register

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)

Type B: **one** source and **immediate operand**

Imm field: a 16 bit value that is sign extend to 32 bits

Logic instructions – Type A

Type A	
OR Rd, Ra, Rb	Rd=Ra | Rb
AND Rd, Ra, Rb	Rd=Ra & Rb
XOR Rd, Ra, Rb	Rd=Rb ^ Ra
ANDN Rd, Ra, Rb	Rd=Ra & (~Rb)

OR Rd,Ra,Rb	100000	Rd	Ra	Rb	00000000000	Rd := Ra or Rb
AND Rd,Ra,Rb	100001	Rd	Ra	Rb	00000000000	Rd := Ra and Rb
XOR Rd,Ra,Rb	100010	Rd	Ra	Rb	00000000000	Rd := Ra xor Rb
ANDN Rd,Ra,Rb	100011	Rd	Ra	Rb	00000000000	Rd := Ra and $\overline{\text{Rb}}$

Logic instructions – Type B

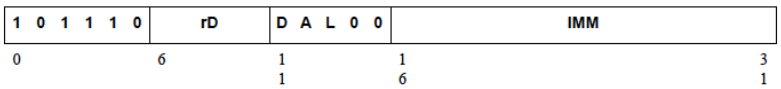
Type B	
ORI Rd, Ra, Imm	Rd=Ra | signExtend32(Imm)
ANDI Rd, Ra, Imm	Rd=Ra & signExtend32(Imm)
XORI Rd, Ra, Imm	Rd=Ra ^ signExtend32(Imm)
ANDNI Rd, Ra, Imm	Rd=Ra & (~signExtend32(Imm))

ORI Rd,Ra,Imm	101000	Rd	Ra	Imm	Rd := Ra or s(Imm)
ANDI Rd,Ra,Imm	101001	Rd	Ra	Imm	Rd := Ra and s(Imm)
XORI Rd,Ra,Imm	101010	Rd	Ra	Imm	Rd := Ra xor s(Imm)
ANDNI Rd,Ra,Imm	101011	Rd	Ra	Imm	Rd := Ra and $\overline{\text{s(Imm)}}$

Unconditional Branch - BRI

Modify the Program Counter (PC) register

Type B	
BRID Imm *branch immediate with delay*	PC=PC+signExtend32(Imm)
BRLID Rd, Imm *branch and link immediate with delay (function call)*	PC=PC+signExtend32(Imm) Rd=PC



If the D bit is set, it means that there is a delay slot and the instruction following the branch (that is, in the branch delay slot) is allowed to complete execution before executing the target instruction

Unconditional Branch - BRI

Modify the Program Counter (PC) register

Unconditional Branch Immediate

bri	IMM	Branch Immediate
brai	IMM	Branch Absolute Immediate
brid	IMM	Branch Immediate with Delay
braid	IMM	Branch Absolute Immediate with Delay
brlid	rD, IMM	Branch and Link Immediate with Delay
bralid	rD, IMM	Branch Absolute and Link Immediate with Delay

Branch to the instruction located at address determined by IMM, sign-extended to 32 bits.

Unconditional Branch - Return from subroutine

Modifies the Program Counter (PC) register

Type B	
RTSD Ra, Imm *return from subroutine*	PC=Ra+signExtend32(Imm)
RTID Ra, Imm *return from interrupt*	PC=Ra+signExtend32(Imm) set interrupt enable in MSR

RTSD Ra,Imm	101101	10000	Ra	Imm	PC := Ra + s(Imm)
RTID Ra,Imm	101101	10001	Ra	Imm	PC := Ra + s(Imm) MSR[IE] := 1

Return from subroutine will branch to the location specified by the contents of rA plus the IMM field, sign-extended to 32 bits.

Conditional Branch - (1)

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

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

BEQI

Branch if rA is equal to 0, to the instruction located in the offset value of IMM. The target of the branch will be the instruction at address PC + IMM.

BNEI

Branch if rA not equal to 0, to the instruction located in the offset value of IMM. The target of the branch will be the instruction at address PC + IMM.

Branch Instructions- Conditional (2)

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

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

BLTI

Branch if rA is less than 0, to the instruction located in the offset value of IMM. The target of the branch will be the instruction at address PC + IMM.

BLEI

Branch if rA is less or equal to 0, to the instruction located in the offset value of IMM. The target of the branch will be the instruction at address PC + IMM.

Branch Instructions- Conditional (2)

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

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

BGTI

Branch if rA is greater than 0, to the instruction located in the offset value of IMM. The target of the branch will be the instruction at address PC + IMM.

BGEI

Branch if rA is greater or equal to 0, to the instruction located in the offset value of IMM. The target of the branch will be the instruction at address PC + IMM.

Load/Store Instructions-Typ A

Type A	
LW Rd, Ra, Rb *load word*	Address=Ra+Rb Rd=*Address
SW Rd, Ra, Rb *store word*	Address=Ra+Rb *Address=Rd

LW

Loads a word (32 bits) from the word aligned memory location that results from adding the contents of registers rA and rB. The data is placed in register rD.

SW

Stores the contents of register rD, into the word aligned memory location that results from adding the contents of registers rA and rB.

Load/Store Instructions-Typ B

Type B	
LWI Rd, Ra, Imm *Load word immediate*	Address=Ra+signExtend32(Imm) Rd=*Address
SWI Rd, Ra, Imm *store word immediate*	Address=Ra+signExtend32(Imm) *Address=Rd

LWI

Loads a word (32 bits) from the word aligned memory location that results from adding the contents of register rA and the value IMM, sign-extended to 32 bits. The data is placed in register rD. A data

SWI

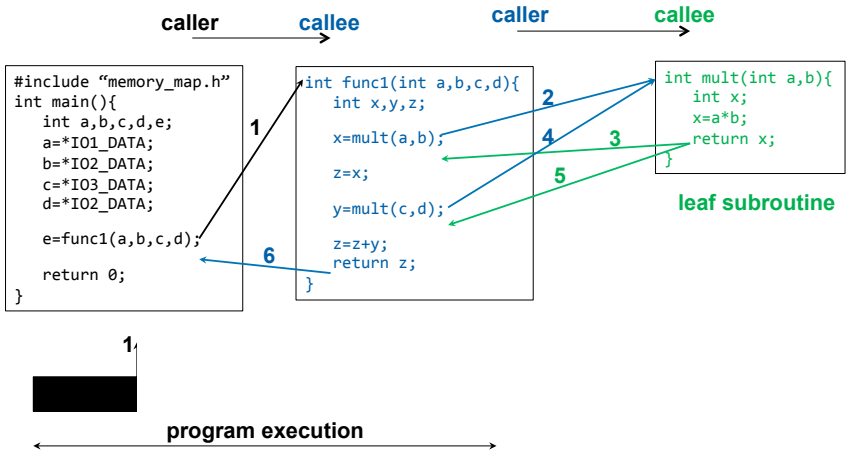
Stores the contents of register rD, into the word aligned memory location that results from adding the contents of registers rA and the value IMM, sign-extended to 32 bits.

Other Instructions

IMM Imm *immediate*	Extend the Imm of a preceding Type B instruction to 32 bits
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*	

The value of a special purpose registers can be transferred to or from a general purpose register by using **mts** and **mfs** instructions, respectively.

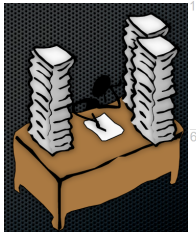
Functions (subroutines)

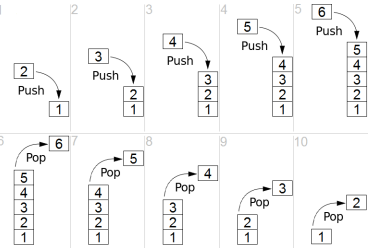


Functions (subroutines) - problems

- 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**
- How to pass arguments to functions?
- How to return values from functions?
  - **FOLLOW A REGISTER USAGE CONVENTION**

Stack





- Memory segment
- Grows towards lower memory address
- 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 →

Stack frame

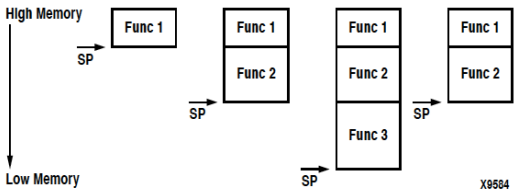
- Temporal storage for the function for its own bookkeeping
  - Return address
  - Local variables used by the function
  - Save registers that the function may modify, but the caller function does not want changed
  - Input arguments to callee functions

Stack Frame Convention

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

Stack pointer points to the top of the latest Stack Frame

Calling a function/ Returning from a function call



After a call from F 1 to F 2, the value of the stack pointer (SP) is decremented. This value of SP is again decremented to accommodate the stack frame for F 3. On return from F 3 the value of the stack pointer is increased to its original value in the function, F 2.

- Calling a function
  - update the stack pointer (PUSH)
  - load the stack frame
- Returning from a function
  - restore the registers that have been previously saved
  - update the stack pointer (POP)

Assembly program

Assembly directives (e.g. other source files, allocate memory,...
Assembly instructions
Symbols (labels)

```
.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
```

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	nop
0x6CC	andi r4,r5,1
0x6D0	add r3,r3,r4
0x6D4	sra r5,r5
0x6D8	brid while
0x6DC	nop
0x6E0	rtsd r15, 8
0x6E4	nop

```
.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
        rsubi r6,1,r6
        bri again
done:   nop
```

LUNDS UNIVERSITET

Example

- Inspect if the binary representation of a number is a palindrome, i.e. reading it left to right is the same as reading it right to left. If so return 1, otherwise return 0.
- Example: “0x000FF000”→ returns 1
  “0xFF000000”→ returns 0

Example

```
int palindrome(int x){
int temp, count, inverted, copied, result;
   count=32;
   inverted=0;
   copied=x;
   while (count!=0){
        inverted=inverted<<1+x&1;
        copied>>=1;
        count--;
   }
   result= inverted^x;
   if (result!=0)
        result=-1;
   return result+1;
}
```

Solution

```
.global palindrome
.text
.ent palindrome
palindrome:  add  r3,r0,r0
             addi r7,r0,32
again:       beqi r7, done
             add r3,r3,r3
             andi r4,r6,1
             add r3,r3,r4
             addi r7,r7,-1
             bri again
done:        xor r4,r5,r3
             beqi r4, result
             addi r4,r0,-1
result:      addi r3,r4,1
             rtsd r15, 8
.end palindrome
```