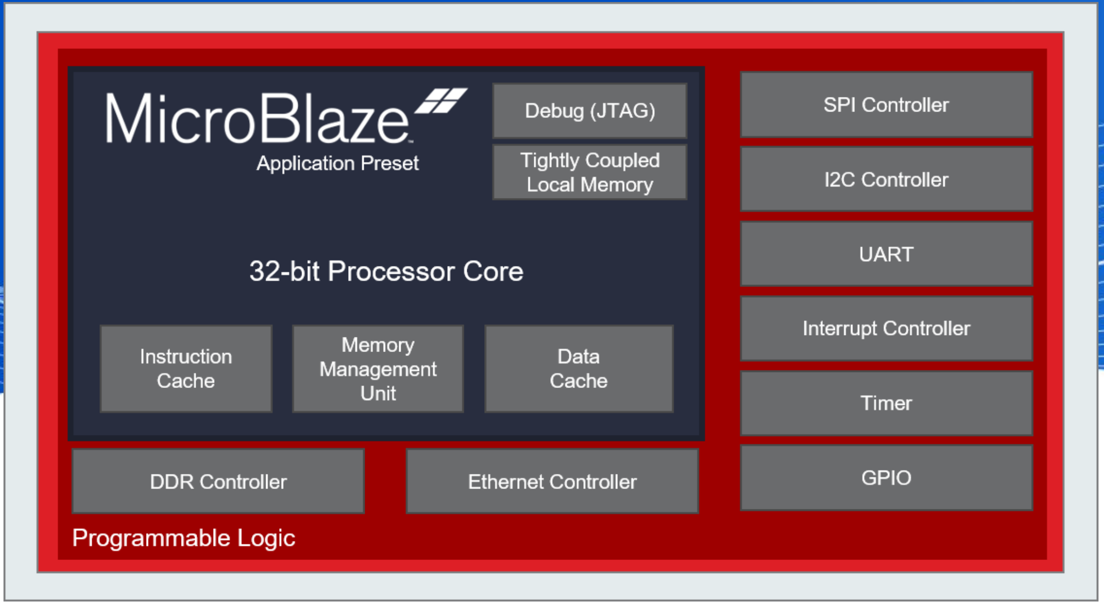
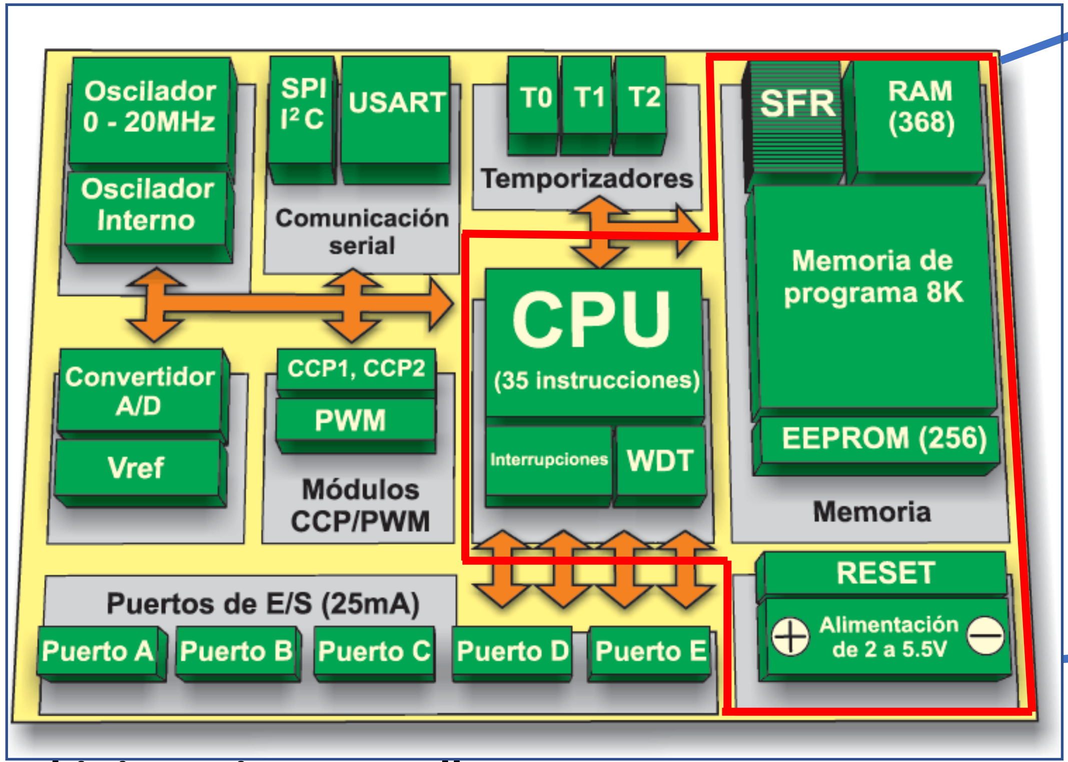
# Lecture 06
# Microblaze



Electrical Engineering Department
Pontificia Universidad Católica de Chile
peclab.ing.uc.cl

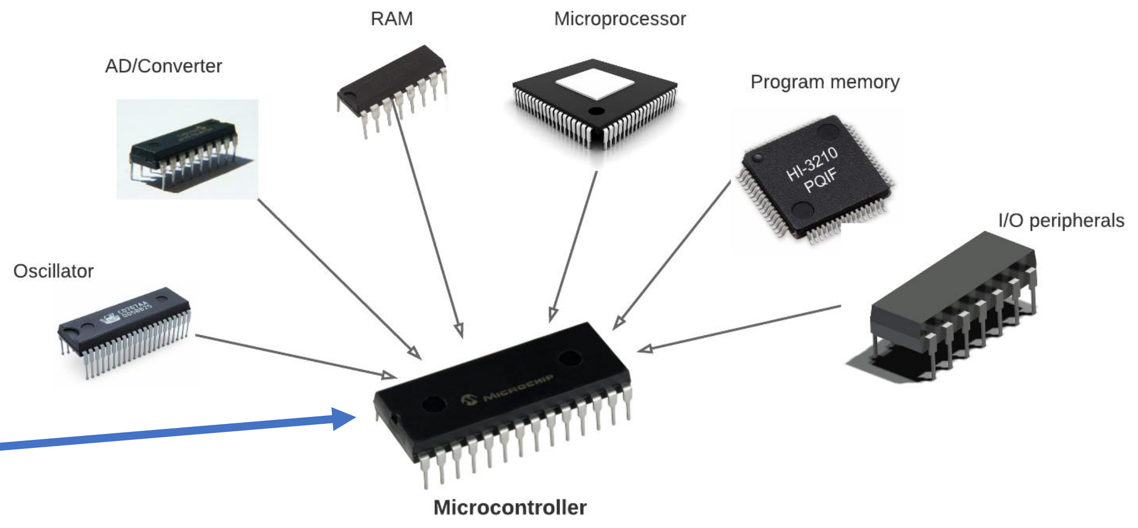
## Microcontroller vs Microprocessor

## With or without peripheral



This is the microprocessor within the microcontroller

This is a Microcontroller:
A microcontroller contain a Microprocessor and also other peripherals!

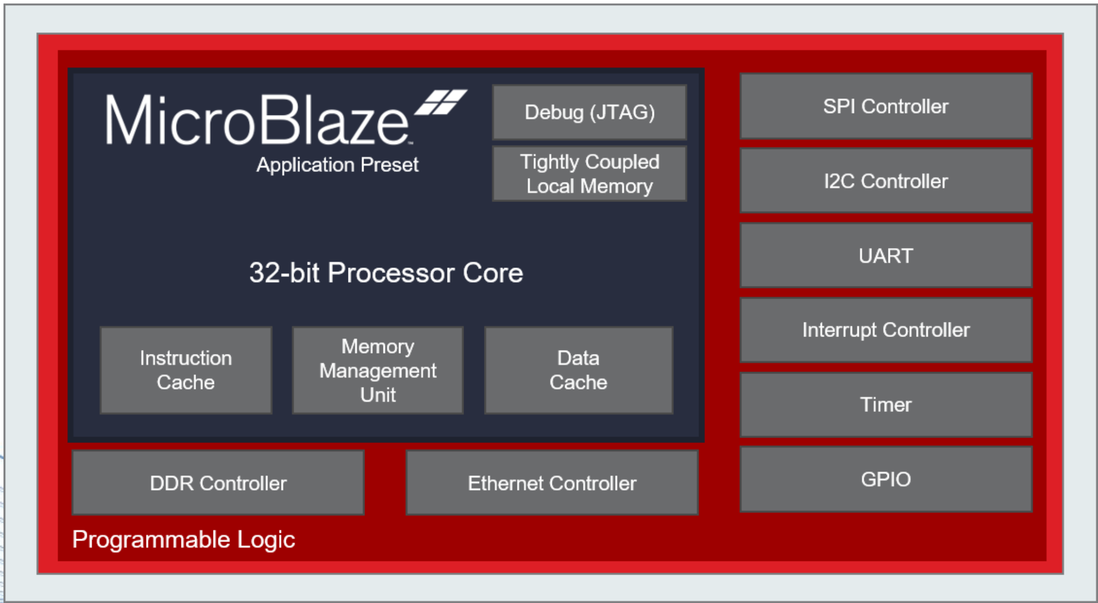


Typical components of a microcontroller

# Microblaze

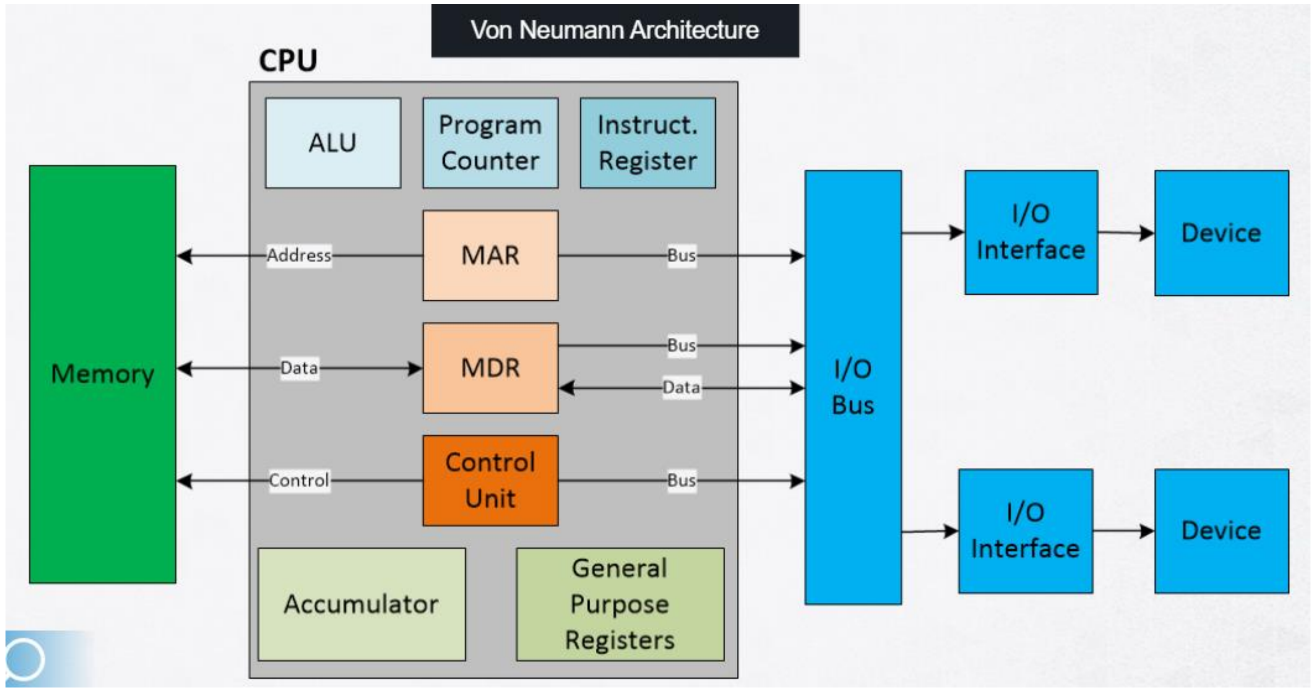
## Overview

- It is usually a softcore processor.
- It can be implemented in FPGA or PL part of a SoC
- It can be also found as hardcore processor (dedicated silicon) in MPSoC and RFSoc.
- It can be configured as 32bit or 64bit.
- 32-bit instruction word with three operands and two addressing modes
- Default 32-bit address bus, extensible to 64 bits
- It is very configurable on its internal capabilities and external connections.



## Microblaze

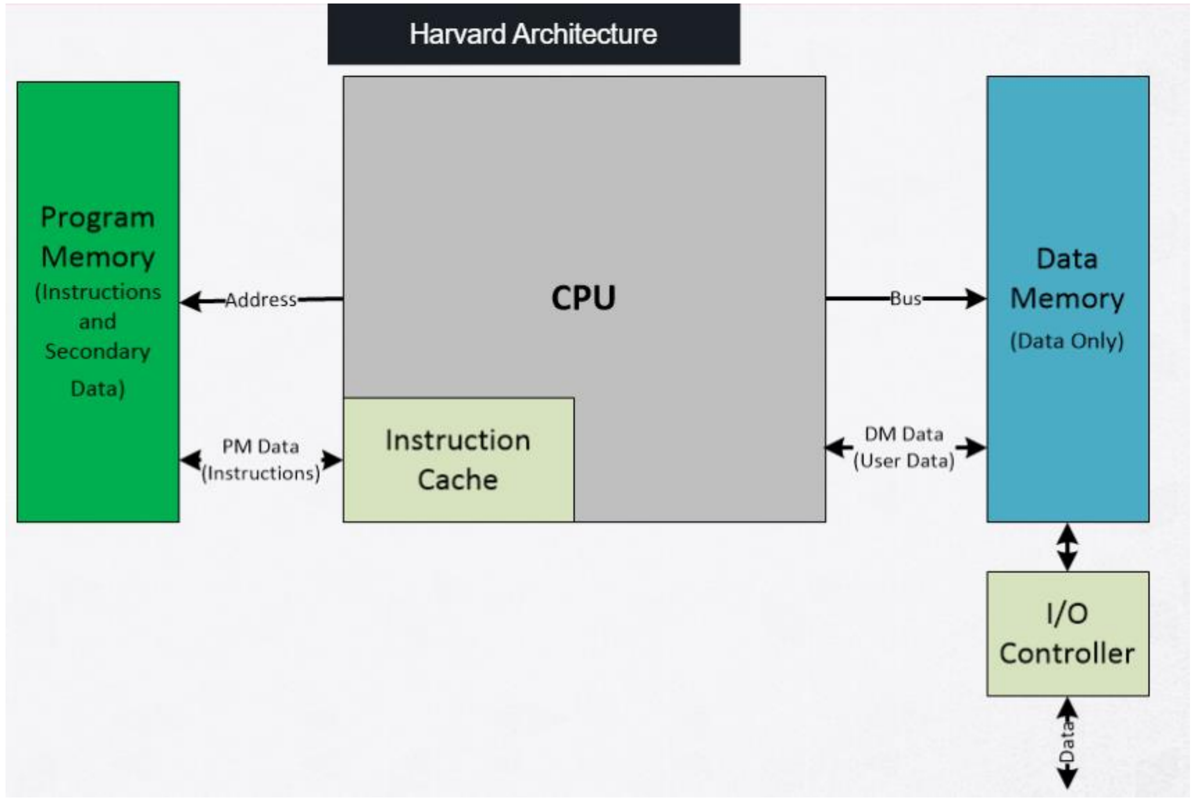
## Architecture: Architecture Von Neumann



| | Von Neuman |
|---|---|
| Disposition | CPU connected to a single memory for data and program information. |
| Buses | Requires one data and one adress bus |
| Execution | Data and instruction can not be read simultaneously |
| Efficiency | Efficient memory utilization. |

## Microblaze

## Architecture: Harvard Architecture

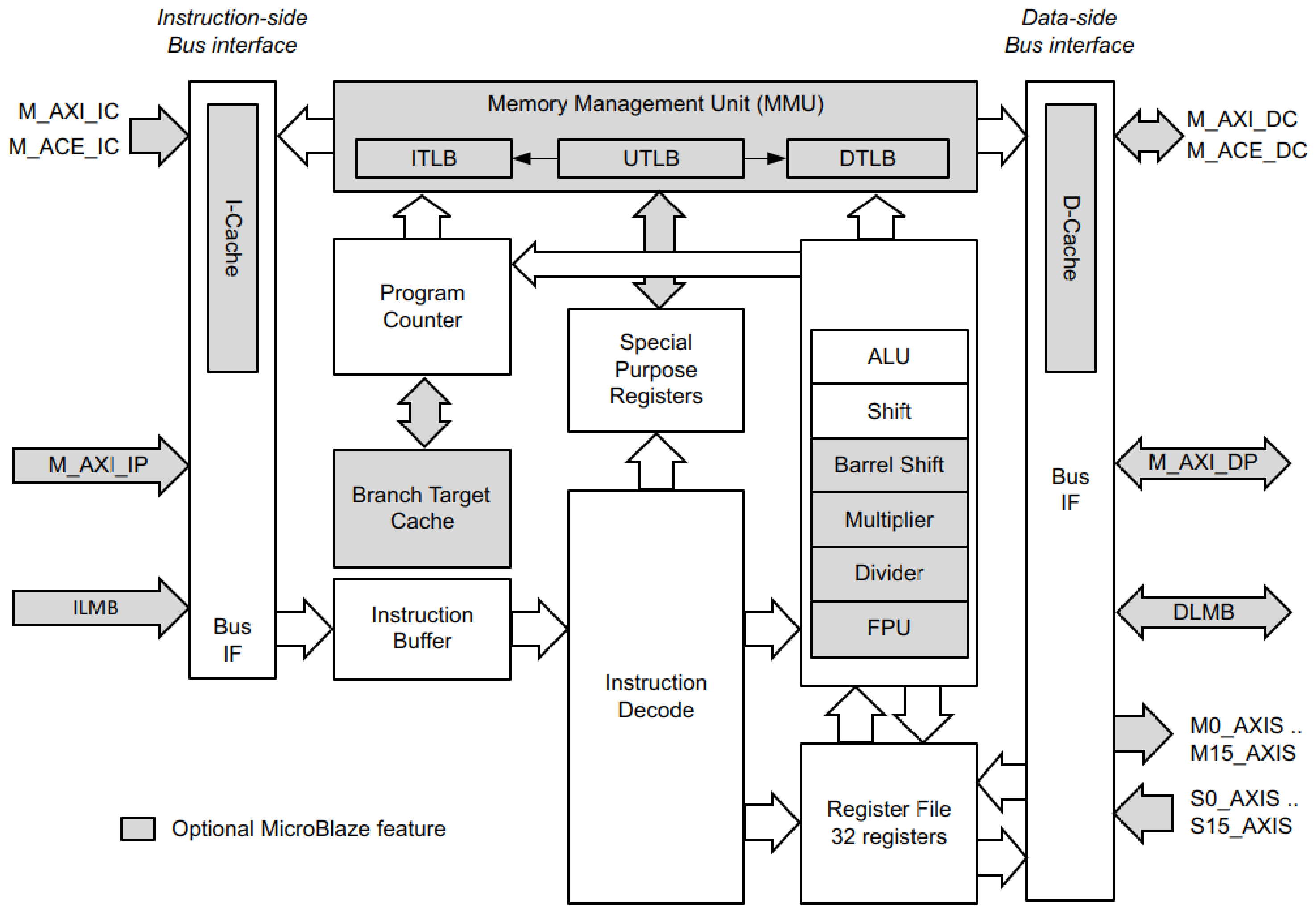


| | Harvard Arch |
|---|---|
| Disposition | CPU separately connected to program memory (ROM) and data memory (RAM). |
| Buses | Requires 2 adress and 2 data buses (more hardware) |
| Execution | Faster program execution as it can read data and instruction at same time. |
| Efficiency | Poor memory utilization. Free data memory can not used for program. |

Microblaze uses Harvard Architecture

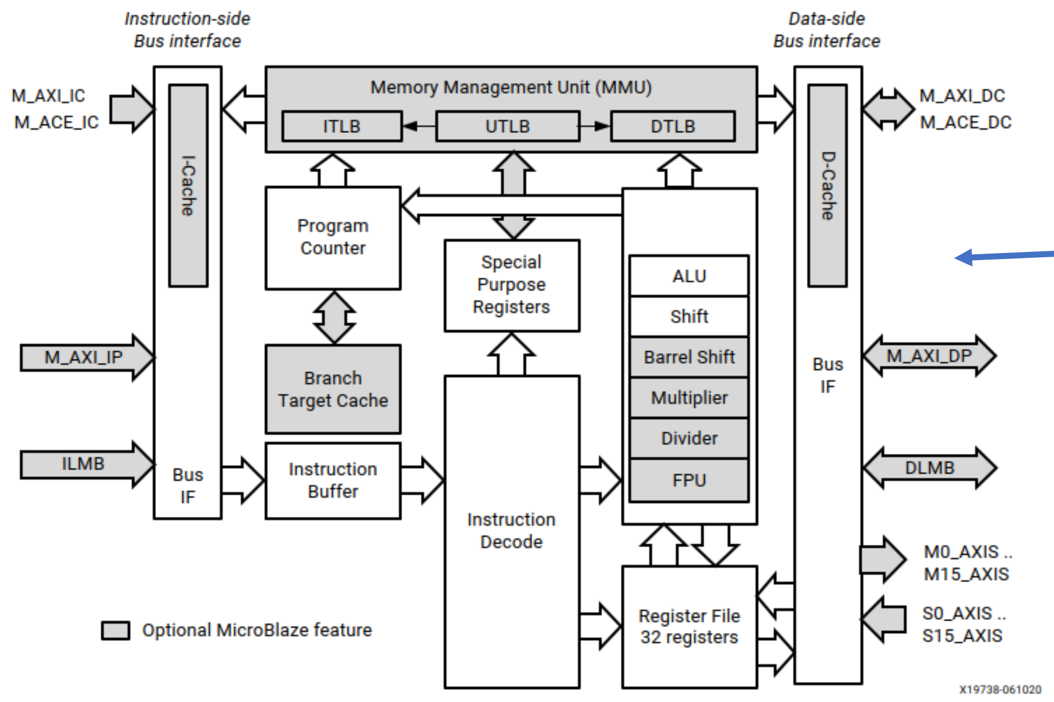
# Microblaze

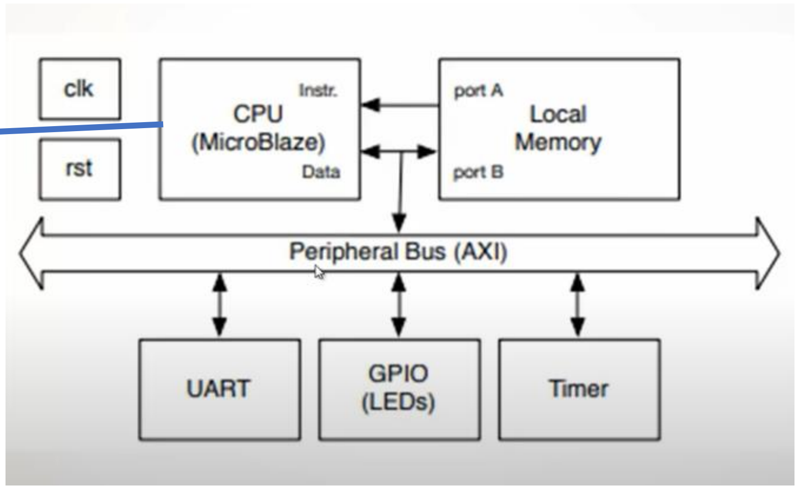
## Architecture: Microblaze is Harvard



**Register** and **Instructions** are the most important pieces to understand how the processor works.

## Microblaze
## Softcore Xilinx



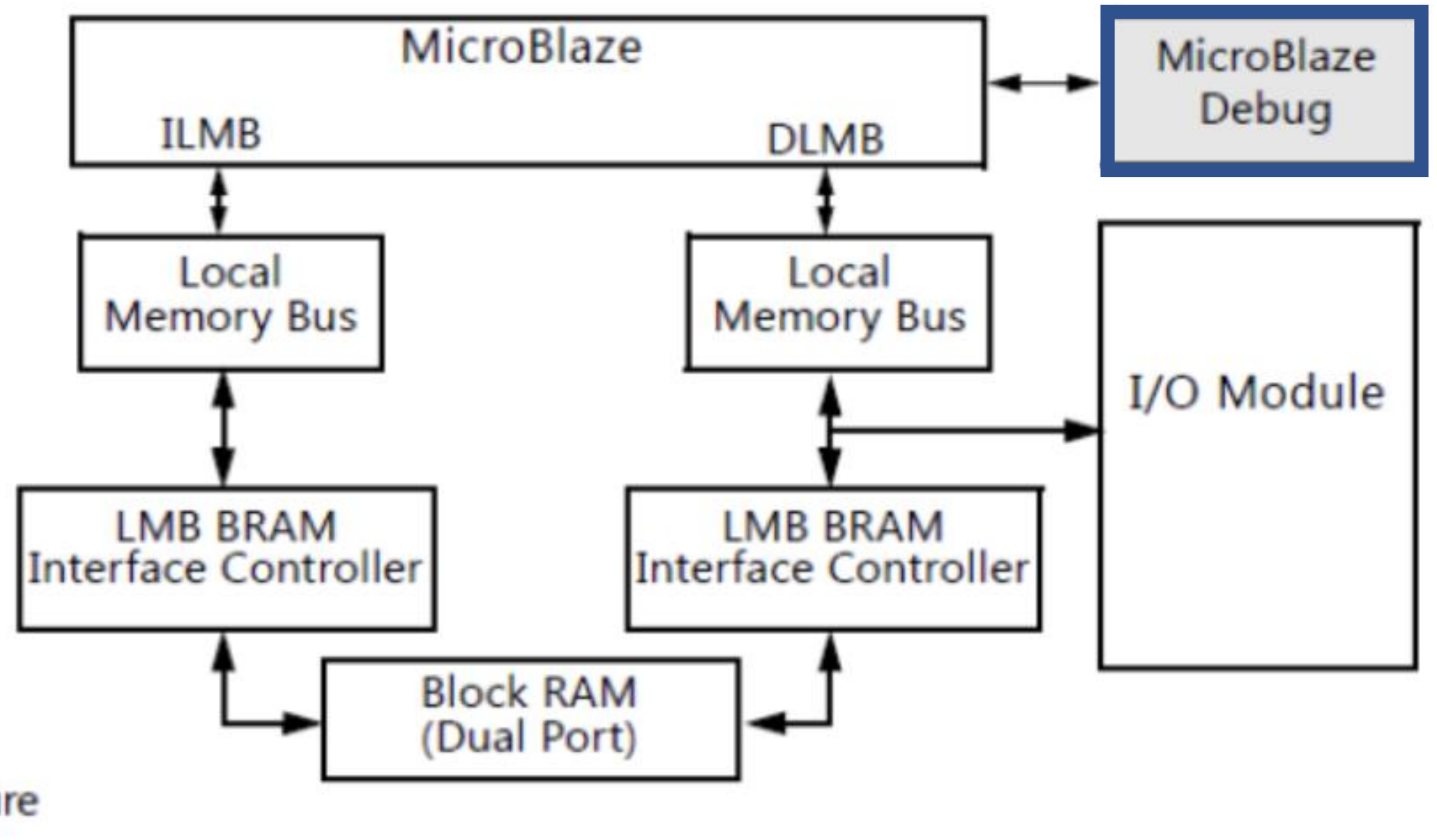


**Microblaze Softcore Structure**

**Microblaze Integrated with Peripherals**

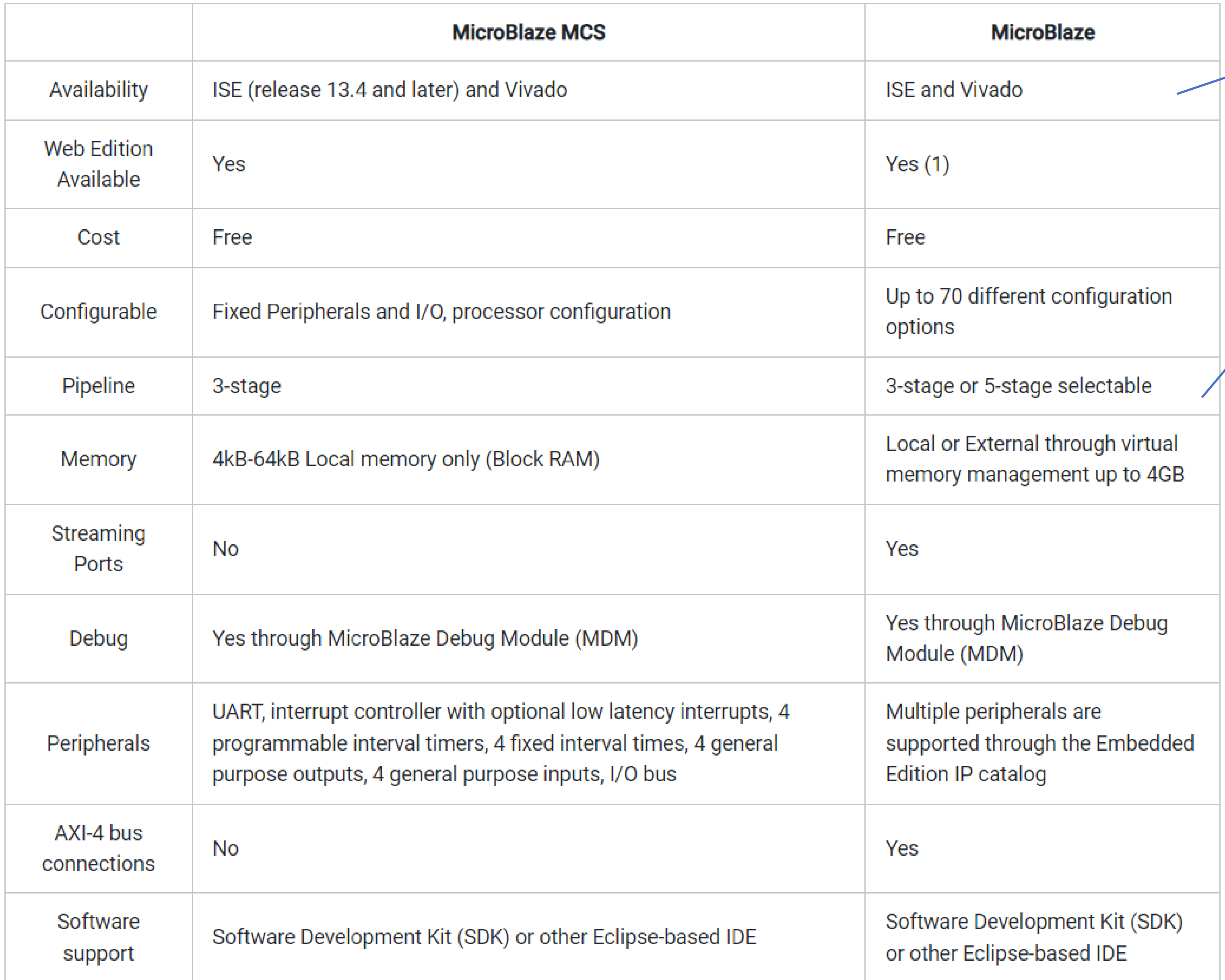
## Microblaze MCS

### Softcore Xilinx



## Microblaze

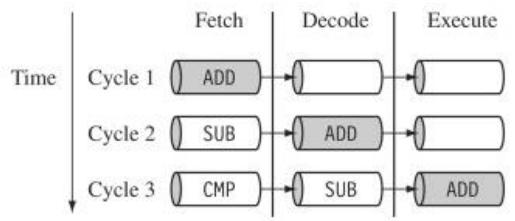
## Microblaze vs Microblaze MCS



| | MicroBlaze MCS | MicroBlaze |
|---|---|---|
| Availability | ISE (release 13.4 and later) and Vivado | ISE and Vivado |
| Web Edition Available | Yes | Yes (1) |
| Cost | Free | Free |
| Configurable | Fixed Peripherals and I/O, processor configuration | Up to 70 different configuration options |
| Pipeline | 3-stage | 3-stage or 5-stage selectable |
| Memory | 4kB-64kB Local memory only (Block RAM) | Local or External through virtual memory management up to 4GB |
| Streaming Ports | No | Yes |
| Debug | Yes through MicroBlaze Debug Module (MDM) | Yes through MicroBlaze Debug Module (MDM) |
| Peripherals | UART, interrupt controller with optional low latency interrupts, 4 programmable interval timers, 4 fixed interval times, 4 general purpose outputs, 4 general purpose inputs, I/O bus | Multiple peripherals are supported through the Embedded Edition IP catalog |
| AXI-4 bus connections | No | Yes |
| Software support | Software Development Kit (SDK) or other Eclipse-based IDE | Software Development Kit (SDK) or other Eclipse-based IDE |

Esta tabla no es tan actual. Por eso se menciona ISE, que es el predecesor de Vivado.

A **three stage RISC** pieline is:
▪ Fetch loads an instruction from memory.
▪ Decode identifies the instruction to be executed.
▪ Execute processes the instruction and writes the result back to a register



**The classic five stage RISC pipeline**
•Instruction fetch.
•Instruction decode.
•Execute.
•Memory access.
•Writeback.

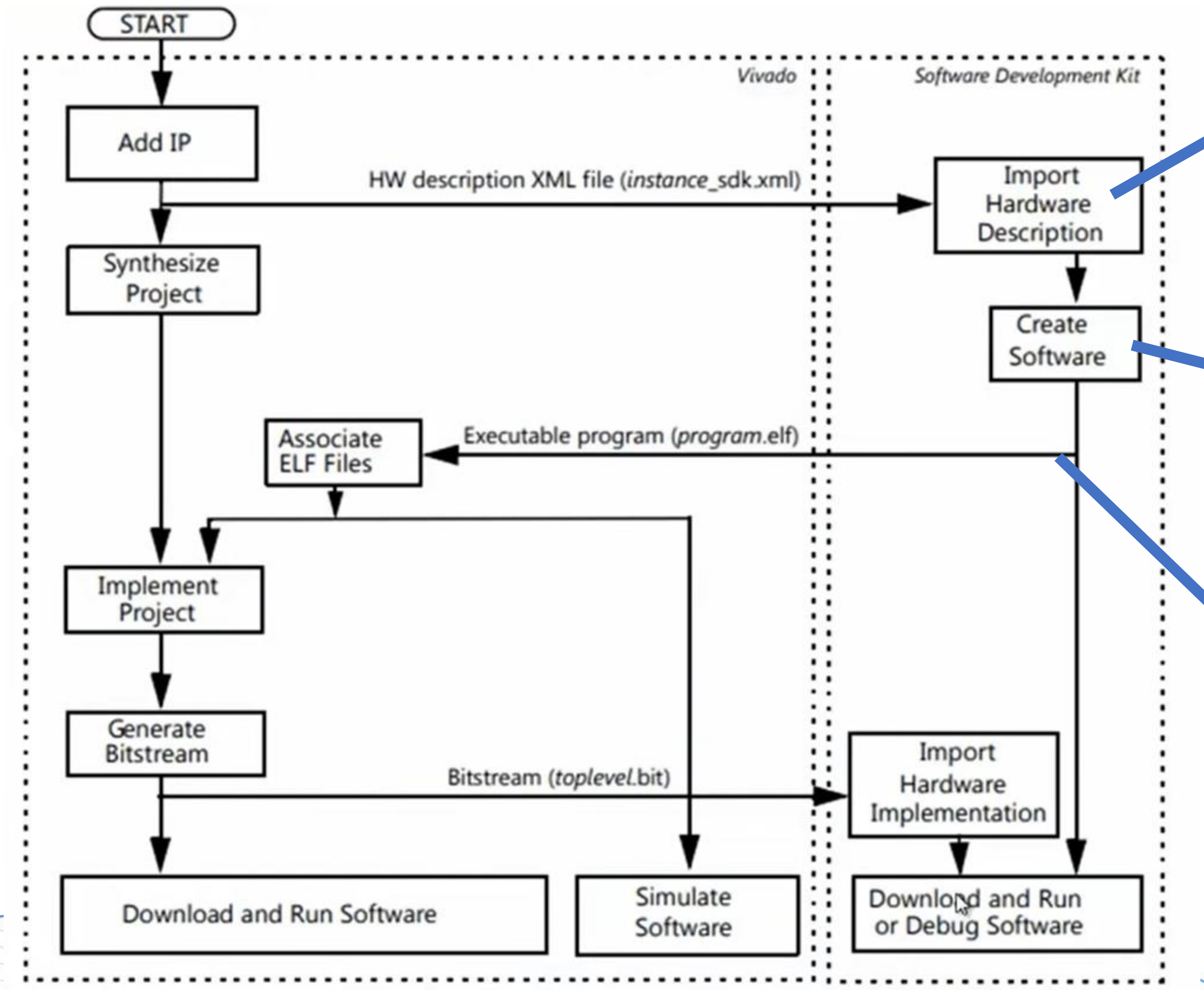
Fetch
Decode
Execute
Memory
Write

This allows the ARM9 to process on average 1.1 Dhrystone MIPS per MHz—an increase in instruction throughput by around 13% compared with an ARM7

For more info check here

# Microblaze

## Softcore Xilinx


START
Vivado
Software Development Kit
Add IP
HW description XML file (*instance_sdk.xml*)
Import Hardware Description
Synthesize Project
Create Software
Associate ELF Files
Executable program (*program.elf*)
Implement Project
Generate Bitstream
Bitstream (*toplevel.bit*)
Import Hardware Implementation
Download and Run Software
Simulate Software
Download and Run or Debug Software

**Import Hardware into Vitis (Optional)**

Hardware specification:

- Number and type of processors
- Memory capabilities
- Available peripherals
- Address ranges, etc.

**Create a Platform Project**

Platform project contains:

- Hardware specification
- Software components: domains and boot elements

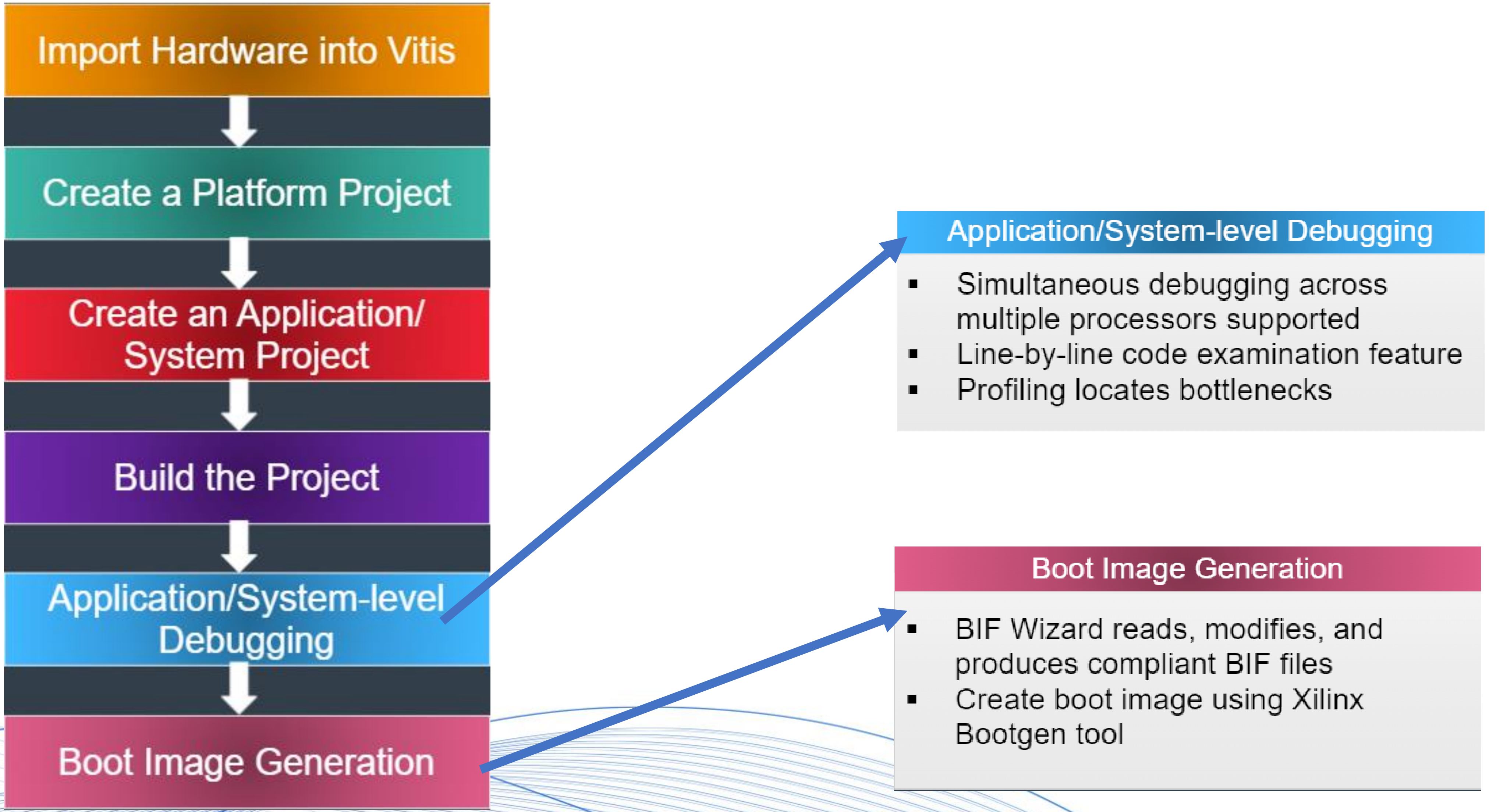
**Create an Application/System Project**

- Use a template or create an application with C/C++ code
- System project created by default
- Each project stored in a separate directory within the workspace

**Build the Project**

- ELF files are generated
- Executable code loaded to all specified processors in the platform

## Microblaze
## Softcore Xilinx


Import Hardware into Vitis
Create a Platform Project
Create an Application/ System Project
Build the Project
Application/System-level Debugging
Boot Image Generation

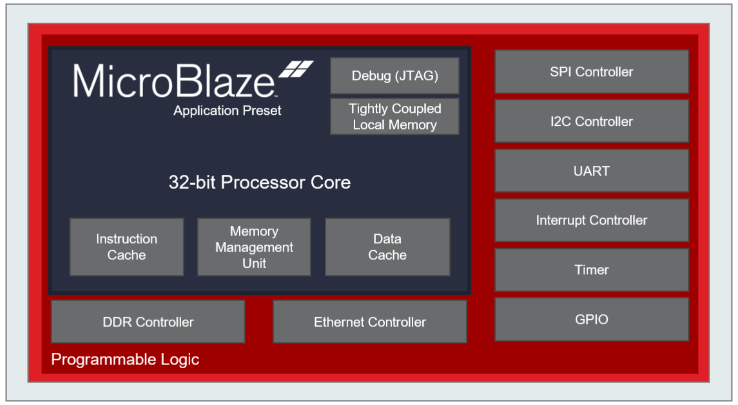
**Application/System-level Debugging**

- Simultaneous debugging across multiple processors supported
- Line-by-line code examination feature
- Profiling locates bottlenecks

**Boot Image Generation**

- BIF Wizard reads, modifies, and produces compliant BIF files
- Create boot image using Xilinx Bootgen tool

# Operation Mode
Overview



## Microblaze

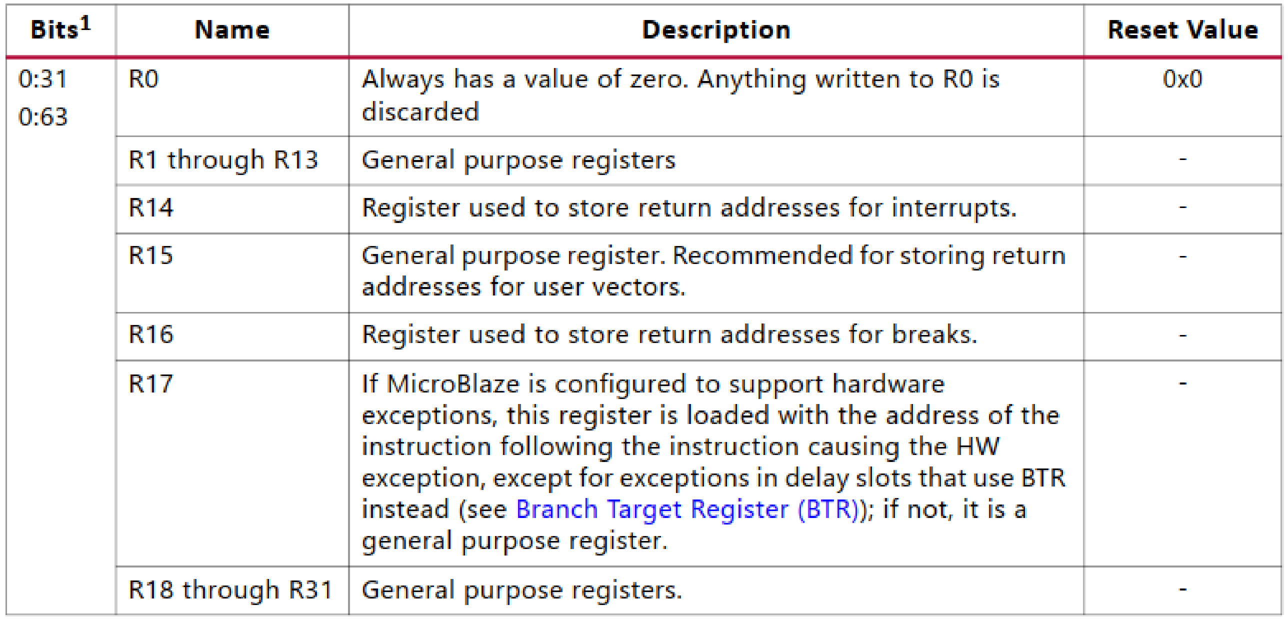
## Registers

- Registers are divided as General Purpose (user) or Special Purpose Registers.
- User Registers are used to hold temporary data being processed.

**In Microblaze:**

- In a 32bit configuration, instructions are 32bits wide. (64bits for 64bits configuraion)
- Insructions can take up to three operands.
- There are 32 general purpose registers (R0 – R31)
- Example: R0 is read-only register containing value 0.

*Table 2-8:* **General Purpose Registers (R0-R31)**



| Bits[1] | Name | Description | Reset Value |
|---|---|---|---|
| 0:31<br>0:63 | R0 | Always has a value of zero. Anything written to R0 is discarded | 0x0 |
| | R1 through R13 | General purpose registers | - |
| | R14 | Register used to store return addresses for interrupts. | - |
| | R15 | General purpose register. Recommended for storing return addresses for user vectors. | - |
| | R16 | Register used to store return addresses for breaks. | - |
| | 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. | - |
| | R18 through R31 | General purpose registers. | - |

1. 64 bits with 64-bit MicroBlaze (C_DATA_SIZE = 64) and 32 bits otherwise

# Microblaze

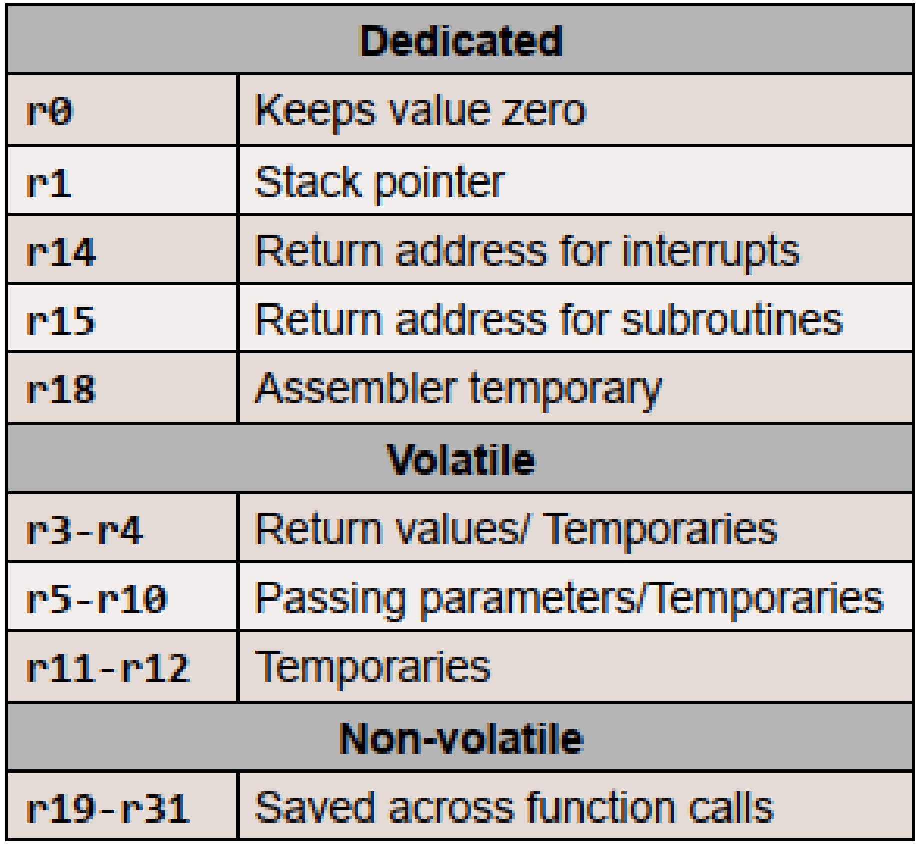
## Registers

- There are up tp 16 special purpose registers (depending on configuration)
- Some special Purpose Registers are:
  - rpc (**Program Counter**)
    - Keeps the address of the instruction being executed
    - Special purpose register 0.
    - Can be read with and **MFS** instruction.
  - rmsr (**Machine Status Register**)
    - Contains control and status bits for the processor
    - Special purpose register 1
    - Can be access with **MFS** and **MTS** instructions.
  - Exception Address Register (EAR)
  - Exception Status Register (ESR)
  - Branch Target Register (BTR)

## Microblaze

## Registers Usage Convention

- For programming in assembly language  a Microbaze we have to follow this convention.
- Volatile values:
  - do not retain the information across function calls.
  - Store temporary results
  - Passing parameters/return values.
- Non-volatile values:
  - Must be saved across function calls.
  - Saved by callee
- Stack Pointer: A stack pointer is a CPU register whose purpose is to keep track of a call stack. register that stores the memory **address of the last data element added to the stack** or, in some cases, the first available address in the stack.



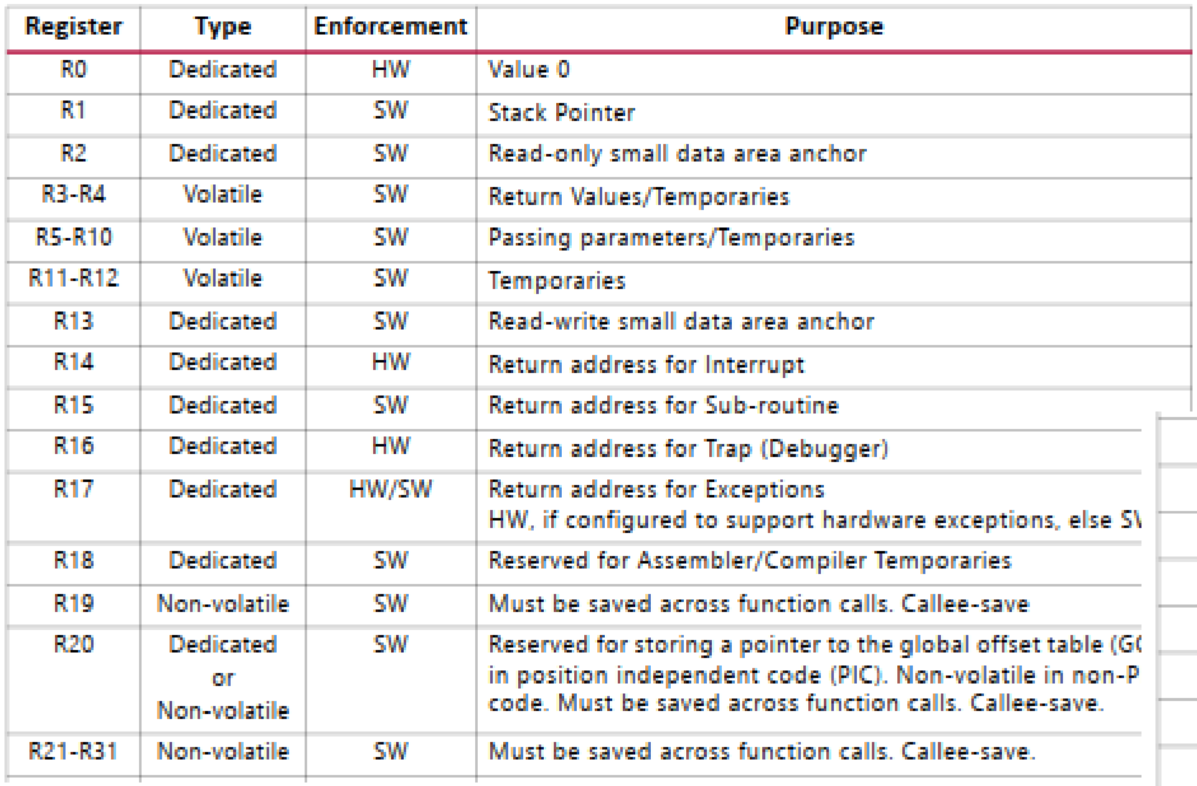
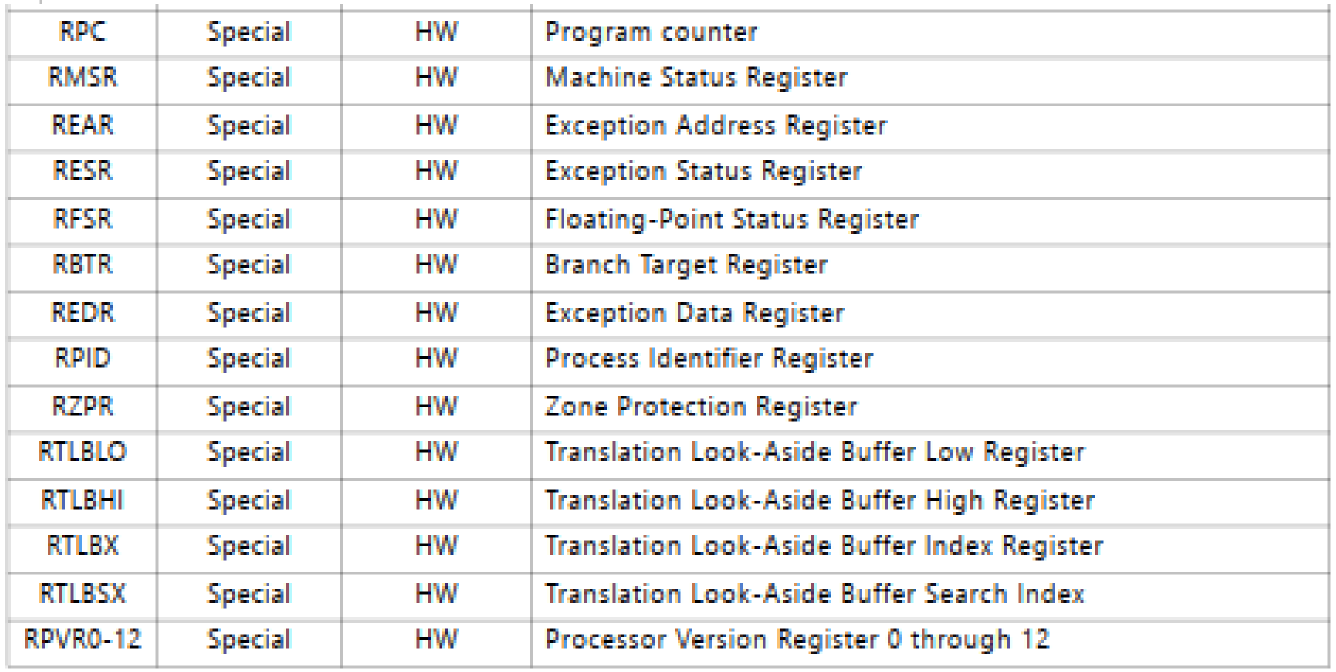
| Dedicated | |
|---|---|
| `r0` | Keeps value zero |
| `r1` | Stack pointer |
| `r14` | Return address for interrupts |
| `r15` | Return address for subroutines |
| `r18` | Assembler temporary |
| **Volatile** | |
| `r3-r4` | Return values/ Temporaries |
| `r5-r10` | Passing parameters/Temporaries |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

Microblaze guide p197

# Microblaze

## Registers Usage Convention

*Table 4-2:* **Register Usage Conventions**

| Register | Type | Enforcement | Purpose |
|---|---|---|---|
| R0 | Dedicated | HW | Value 0 |
| R1 | Dedicated | SW | Stack Pointer |
| R2 | Dedicated | SW | Read-only small data area anchor |
| R3-R4 | Volatile | SW | Return Values/Temporaries |
| R5-R10 | Volatile | SW | Passing parameters/Temporaries |
| R11-R12 | Volatile | SW | Temporaries |
| R13 | Dedicated | SW | Read-write small data area anchor |
| R14 | Dedicated | HW | Return address for Interrupt |
| R15 | Dedicated | SW | Return address for Sub-routine |
| R16 | Dedicated | HW | Return address for Trap (Debugger) |
| R17 | Dedicated | HW/SW | Return address for Exceptions<br>HW, if configured to support hardware exceptions, else S |
| R18 | Dedicated | SW | Reserved for Assembler/Compiler Temporaries |
| R19 | Non-volatile | SW | Must be saved across function calls. Callee-save |
| R20 | Dedicated<br>or<br>Non-volatile | SW | Reserved for storing a pointer to the global offset table (G<br>in position independent code (PIC). Non-volatile in non-P<br>code. Must be saved across function calls. Callee-save. |
| R21-R31 | Non-volatile | SW | Must be saved across function calls. Callee-save. |
| RPC | Special | HW | Program counter |
| RMSR | Special | HW | Machine Status Register |
| REAR | Special | HW | Exception Address Register |
| RESR | Special | HW | Exception Status Register |
| RFSR | Special | HW | Floating-Point Status Register |
| RBTR | Special | HW | Branch Target Register |
| REDR | Special | HW | Exception Data Register |
| RPID | Special | HW | Process Identifier Register |
| RZPR | Special | HW | Zone Protection Register |
| RTLBLO | Special | HW | Translation Look-Aside Buffer Low Register |
| RTLBHI | Special | HW | Translation Look-Aside Buffer High Register |
| RTLBX | Special | HW | Translation Look-Aside Buffer Index Register |
| RTLBSX | Special | HW | Translation Look-Aside Buffer Search Index |
| RPVR0-12 | Special | HW | Processor Version Register 0 through 12 |

## Microblaze
## Instructions

- Microblaze is a RISC (Reduced Instruction Set Computing) Processor:
  - Execute most instructions in a single clock cycle.
  - Each instruction is atomic and will be completed before handling an interrupt or exception.
  - It supports only few basic addressing modes. (how to calculate address of operands by using information in registers or constants)

- For your knowledge, CISCC (Complex Instructions Set Computing) are:
  - A single instruction can take 20 clock cycles.
  - A single instruction can use one or more levels of indirection to fin data to operate.
  - Unlike RISC, CISC was intended to be used by humans.
  - High level language instructions (such as C), can be broken into several small RISC instructions in a very optimal manner. Therefore, CISC processors are less common.

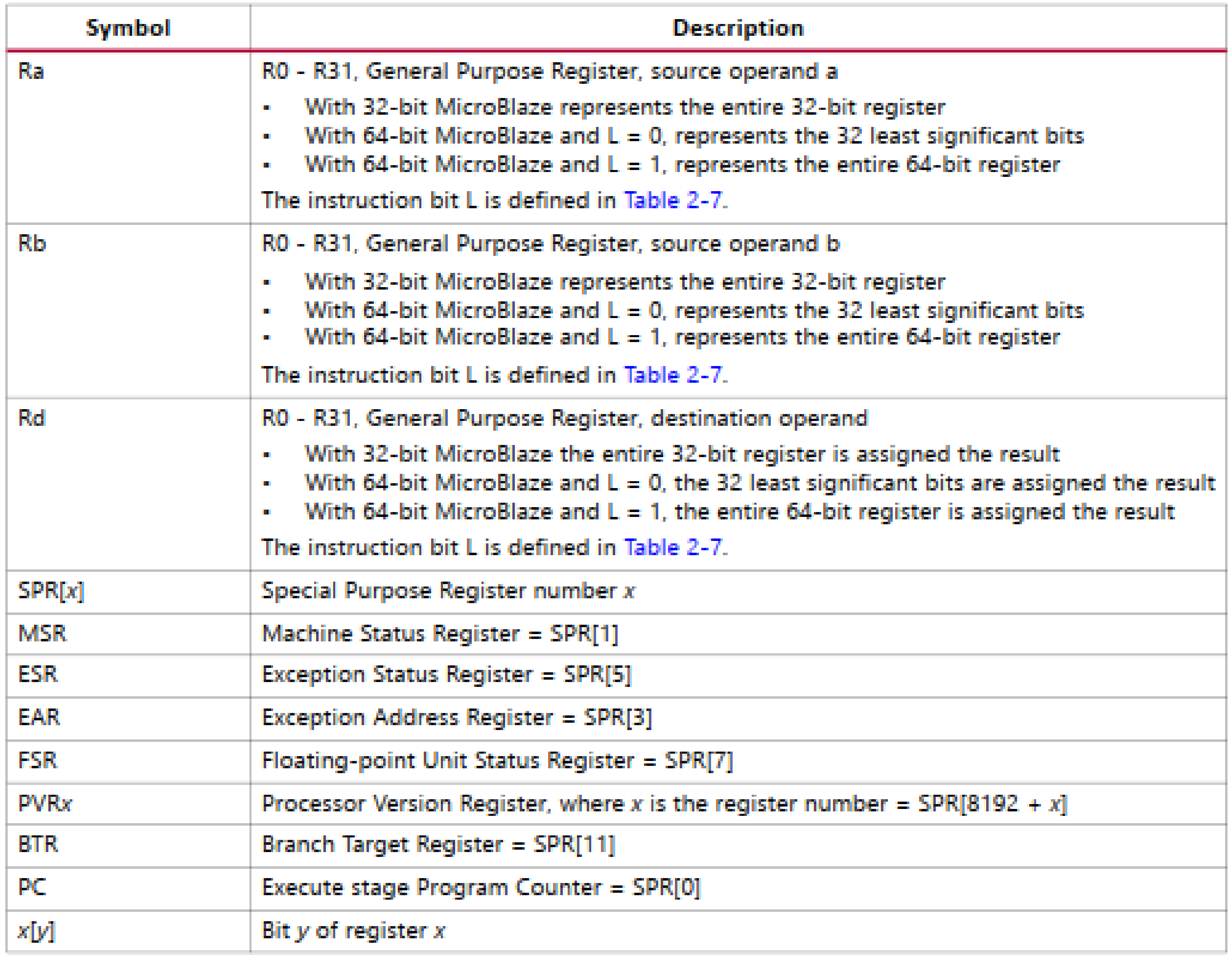
# Microblaze
## Instructions

1. **Set of Instructions:** Each microprocesor has its own set of instructions (machine lenguaje and equivalently in assembly). These instructions allow the processing of data.
2. Microblaze possesses 32-bits instructions. They are defined as Type A or B.
3. Type A instructions have up to two source registers and one destination register.
4. Type B instructions have one source register and a 16-bit inmediate operand. Also a destination register.
5. Instructions are classified as:
    1. Arithmetic
    2. Logical
    3. Branch
    4. Load/store
    5. Special

## Microblaze

## Instructions Nomenclature

*Table 2-6:* **Instruction Set Nomenclature**



| Symbol | Description |
|---|---|
| Ra | R0 - R31, General Purpose Register, source operand a<br>• With 32-bit MicroBlaze represents the entire 32-bit register<br>• With 64-bit MicroBlaze and L = 0, represents the 32 least significant bits<br>• With 64-bit MicroBlaze and L = 1, represents the entire 64-bit register<br>The instruction bit L is defined in Table 2-7. |
| Rb | R0 - R31, General Purpose Register, source operand b<br>• With 32-bit MicroBlaze represents the entire 32-bit register<br>• With 64-bit MicroBlaze and L = 0, represents the 32 least significant bits<br>• With 64-bit MicroBlaze and L = 1, represents the entire 64-bit register<br>The instruction bit L is defined in Table 2-7. |
| Rd | R0 - R31, General Purpose Register, destination operand<br>• With 32-bit MicroBlaze the entire 32-bit register is assigned the result<br>• With 64-bit MicroBlaze and L = 0, the 32 least significant bits are assigned the result<br>• With 64-bit MicroBlaze and L = 1, the entire 64-bit register is assigned the result<br>The instruction bit L is defined in Table 2-7. |
| SPR[*x*] | Special Purpose Register number *x* |
| MSR | Machine Status Register = SPR[1] |
| ESR | Exception Status Register = SPR[5] |
| EAR | Exception Address Register = SPR[3] |
| FSR | Floating-point Unit Status Register = SPR[7] |
| PVR*x* | Processor Version Register, where *x* is the register number = SPR[8192 + *x*] |
| BTR | Branch Target Register = SPR[11] |
| PC | Execute stage Program Counter = SPR[0] |
| *x*[*y*] | Bit *y* of register *x* |

## Microblaze
## Arithmetic Instructions

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

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

*Immediate* operand
Imm field: 16bit value to extend *immediate* operand.

## Microblaze

## Logic Instructions

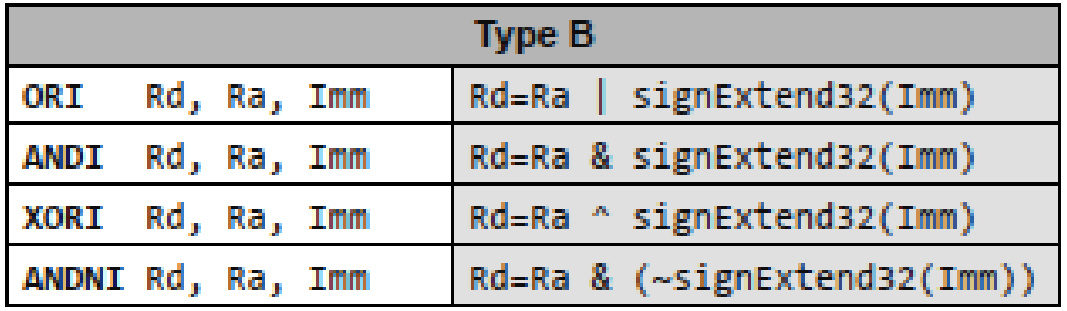
| Type A | |
|---|---|
| **OR** Rd, Ra, Rb | Rd=Ra \| Rb |
| **AND** Rd, Ra, Rb | Rd=Ra & Rb |
| **XOR** Rd, Ra, Rb | Rd=Rb ^ Ra |
| **ANDN** Rd, Ra, Rb | Rd=Ra & (~Rb) |

| | | | | | | |
|---|---|---|---|---|---|---|
| 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 $\overline{\text{Rb}}$ |

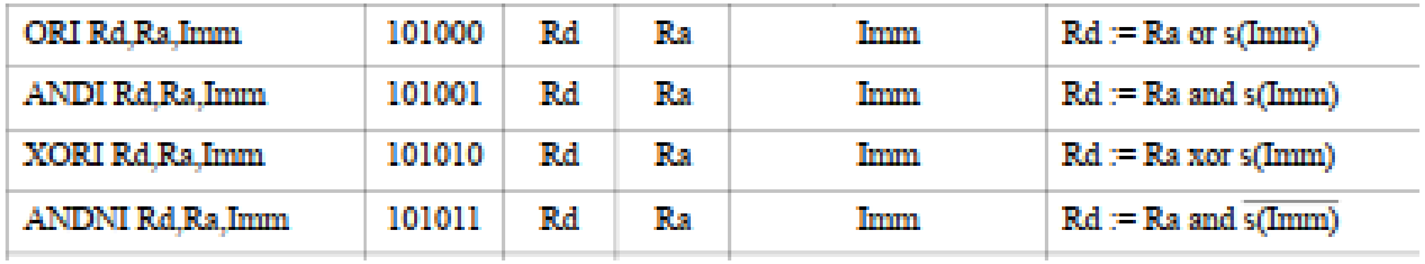
## Microblaze

## Logic Instructions

### 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 $\overline{s(Imm)}$ |

## Microblaze

## Branch Instructions - Unconditional Branch

**Modify the Program Counter (PC) register**

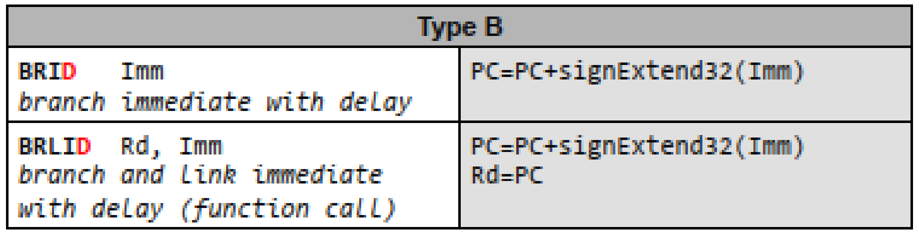
**Unconditional Branch Immediate**

| | | |
|---|---|---|
| **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 |

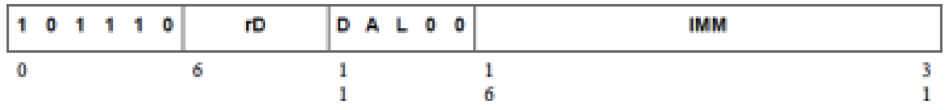
Branch to the instruction located at address determined by IMM, sign-extended to 32 bits.

## Microblaze

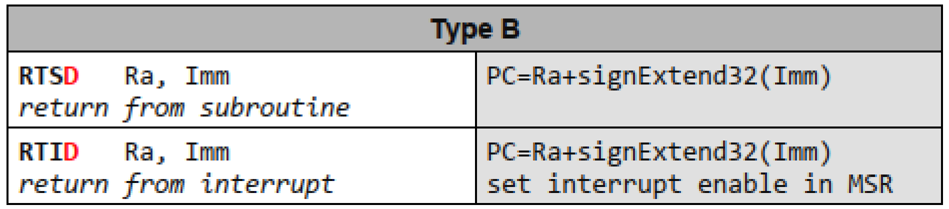
## Branch Instructions (modify PC) - Unconditional Branch



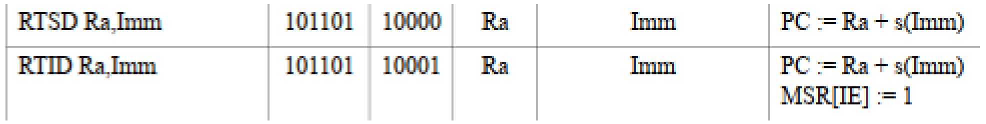
| Type B | |
|---|---|
| BRID Imm<br>*branch immediate with delay* | PC=PC+signExtend32(Imm) |
| BRLID Rd, Imm<br>*branch and link immediate with delay (function call)* | PC=PC+signExtend32(Imm)<br>Rd=PC |



| 1 0 1 1 1 0 | rD | D A L 0 0 | IMM |
|---|---|---|---|
| 0 | 6 | 11 | 16 … 31 |



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



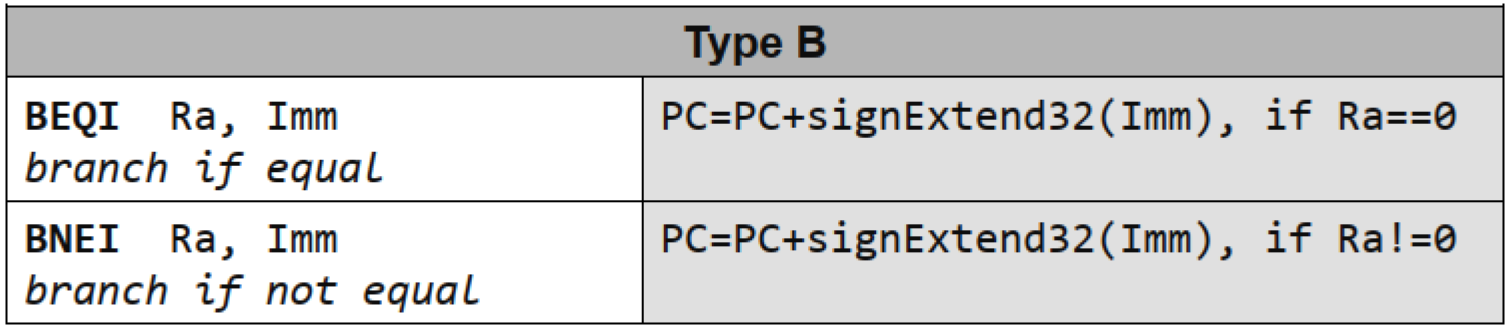
| | | | | | |
|---|---|---|---|---|---|
| RTSD Ra,Imm | 101101 | 10000 | Ra | Imm | PC := Ra + s(Imm) |
| RTID Ra,Imm | 101101 | 10001 | Ra | Imm | PC := Ra + s(Imm)<br>MSR[IE] := 1 |

If the D bit is set, it means that there is a delay slot and the instructions following the Branch is allowed to complete execution before executing the target instruction.

Return from subrutine will Branch to the location specified by the contents of RA plus the IMM field, sign-extended to 32bits.

## Microblaze

## Branch Instructions (modify PC) -  Unconditional Branch

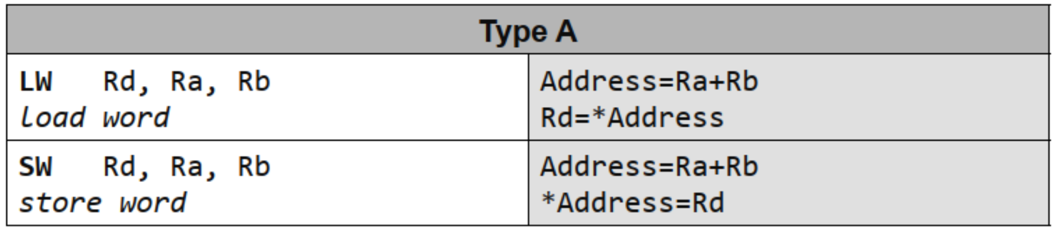


| Type B | |
|---|---|
| **BEQI** Ra, Imm<br>*branch if equal* | PC=PC+signExtend32(Imm), if Ra==0 |
| **BNEI** Ra, Imm<br>*branch if not equal* | PC=PC+signExtend32(Imm), if Ra!=0 |

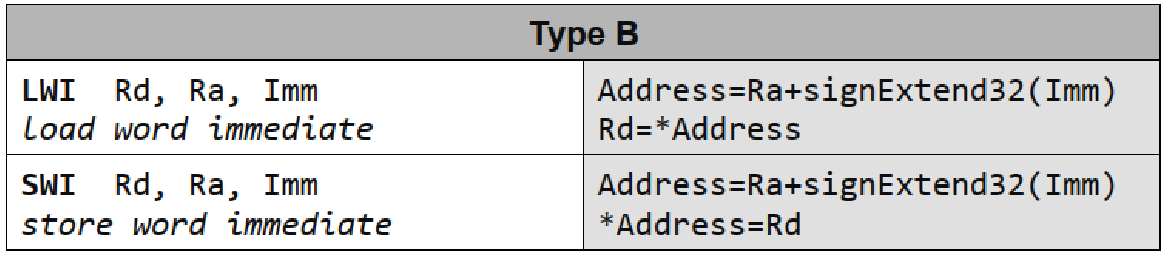
BEQI: If RA ==0 , modify programm counter to Imm offset.
BNEI: If RA !=0, modify programm counter to Imm offset.

# Microblaze

## LOAD/Store Instuctions



| Type A | |
|---|---|
| **LW** Rd, Ra, Rb<br>*load word* | Address=Ra+Rb<br>Rd=*Address |
| **SW** Rd, Ra, Rb<br>*store word* | Address=Ra+Rb<br>*Address=Rd |



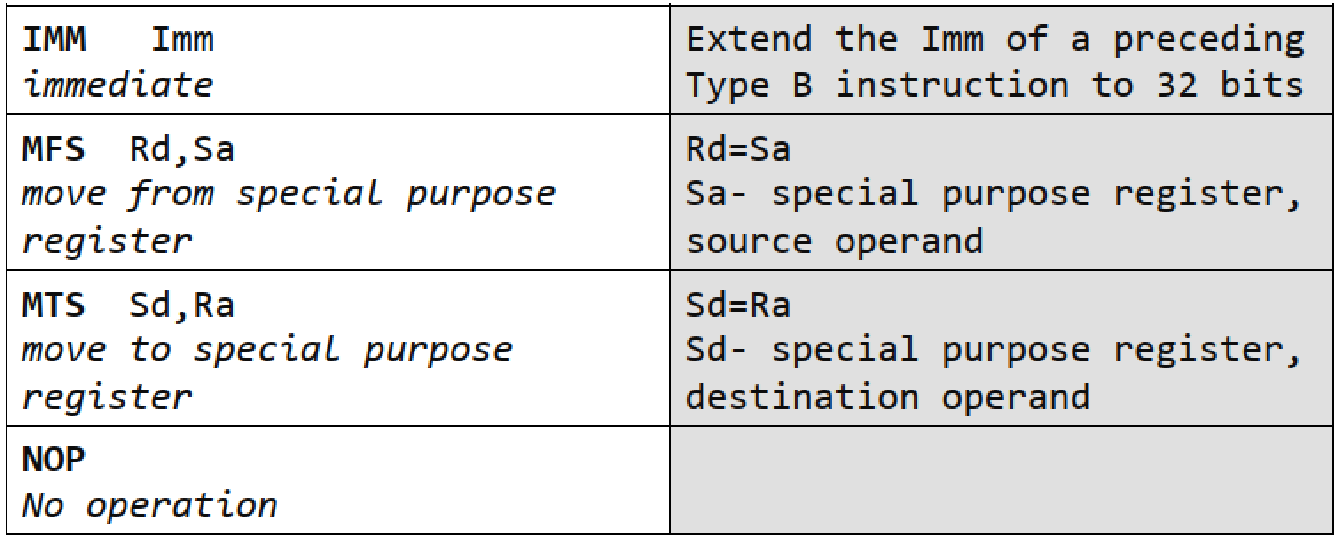
| Type B | |
|---|---|
| **LWI** Rd, Ra, Imm<br>*load word immediate* | Address=Ra+signExtend32(Imm)<br>Rd=*Address |
| **SWI** Rd, Ra, Imm<br>*store word immediate* | Address=Ra+signExtend32(Imm)<br>*Address=Rd |

LW: loads a word/data (32bits) from address RA+RB and it is placed in RD.
SW: Stores the value in RD into the address RA+RB.

LWI and SWI are similar to LW and SW but using immediate word Imm.

## Microblaze

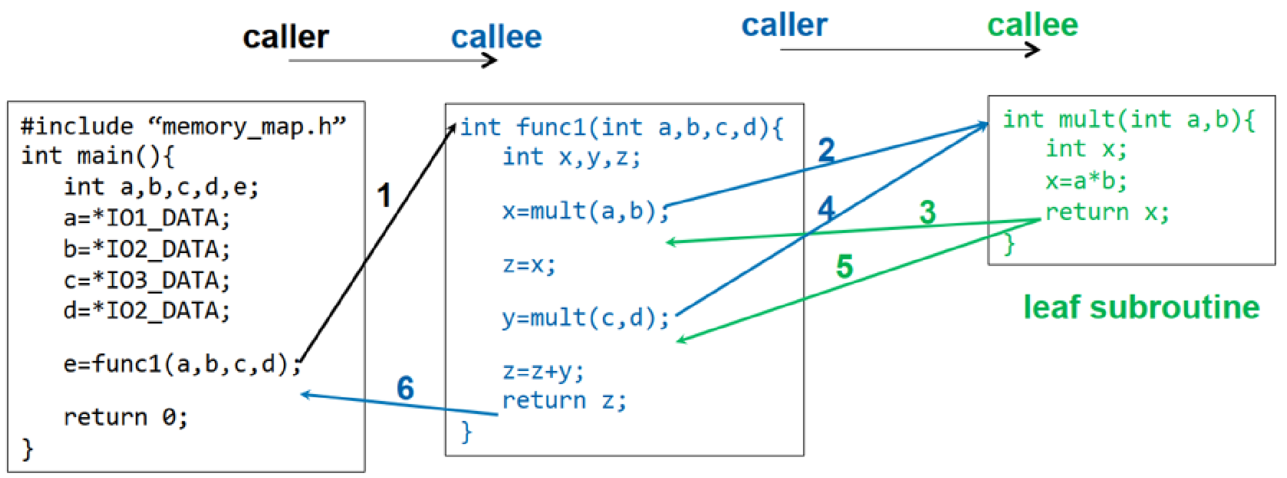
## Special instructions.



| | |
|---|---|
| **IMM** Imm<br>*immediate* | Extend the Imm of a preceding Type B instruction to 32 bits |
| **MFS** Rd,Sa<br>*move from special purpose register* | Rd=Sa<br>Sa- special purpose register, source operand |
| **MTS** Sd,Ra<br>*move to special purpose register* | Sd=Ra<br>Sd- special purpose register, destination operand |
| **NOP**<br>*No operation* | |

The value of special purpose registers can be transferred to or from a general purpose register using **MTS** and **MFS** instructions respectively.

## Microblaze
## Concept of functions and Subrutines



```
#include “memory_map.h”
int main(){
    int a,b,c,d,e;
    a=*IO1_DATA;
    b=*IO2_DATA;
    c=*IO3_DATA;
    d=*IO2_DATA;

    e=func1(a,b,c,d);

    return 0;
}
```

```
int func1(int a,b,c,d){
    int x,y,z;

    x=mult(a,b);

    z=x;

    y=mult(c,d);

    z=z+y;
    return z;
}
```

```
int mult(int a,b){
    int x;
    x=a*b;
    return x;
}
```

From this example we can see the need of branch instructions . So we can jump from one subrutine to another and back to the previous rutine. BUT:

- How to ensure that registers retain values across function calls?
- Where to return after a function has been executed?
- Where to store temporaty local variables of a function?

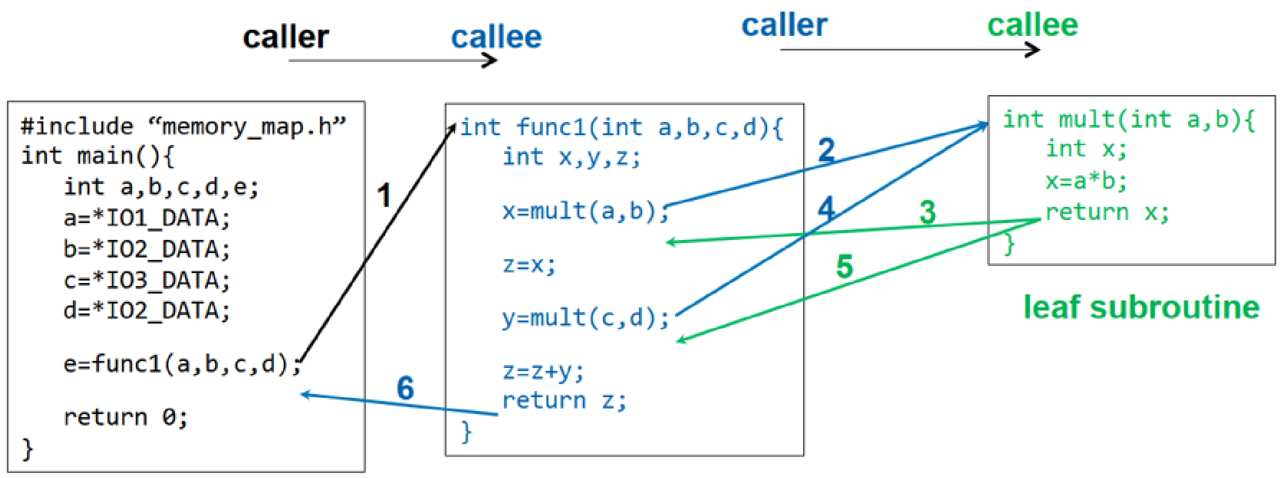
**USE THE STACK!**

- How to pass argumens to functions?
- How to return values from functions?

**FOLLOW A REGISTER USAGE CONVENTION**

## Microblaze

## Concept of functions and Subrutines



```
#include “memory_map.h”
int main(){
    int a,b,c,d,e;
    a=*IO1_DATA;
    b=*IO2_DATA;
    c=*IO3_DATA;
    d=*IO2_DATA;

    e=func1(a,b,c,d);

    return 0;
}
```

```
int func1(int a,b,c,d){
    int x,y,z;

    x=mult(a,b);

    z=x;

    y=mult(c,d);

    z=z+y;
    return z;
}
```

```
int mult(int a,b){
    int x;
    x=a*b;
    return x;
}
```

From this example we can see the need of branch instructions . So we can jump from one subrutine to another and back to the previous rutine. BUT:

- How to ensure that registers retain values across function calls?
- Where to return after a function has been executed?
- Where to store temporaty local variables of a function?

**USE THE STACK!**

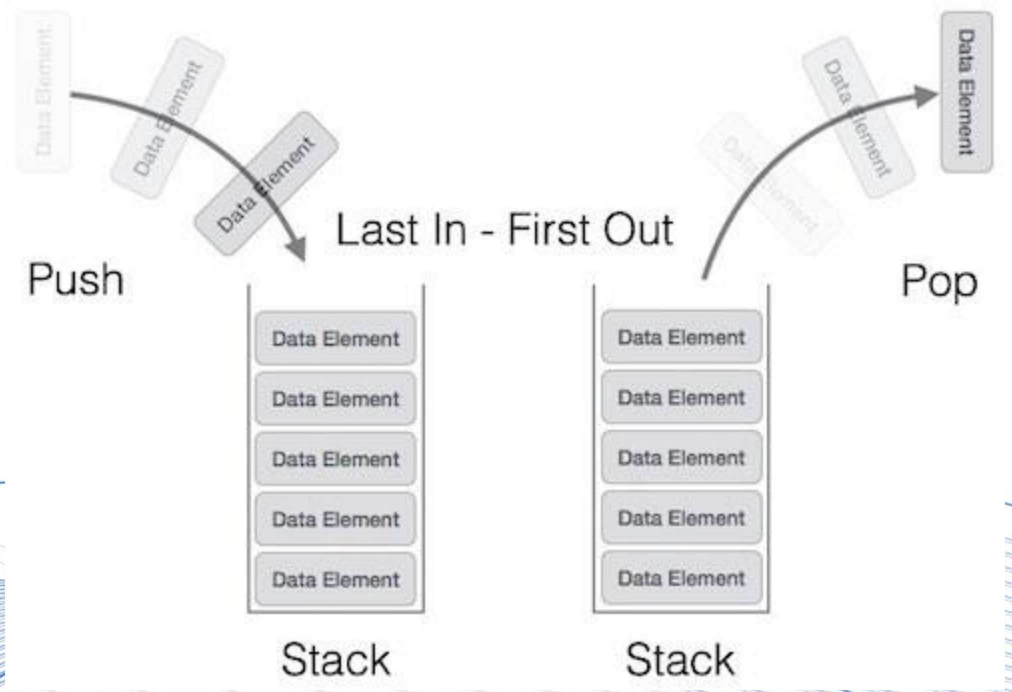
- How to pass argumens to functions?
- How to return values from functions?

**FOLLOW A REGISTER USAGE CONVENTION**

## Microblaze

## STACK of DATA

- It is a Memory segments
- Grows towards lower memory address (from 0xFFFFFFFF to 0x00000000)
- We access the stack through a Stack Pointer (Register)
- 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



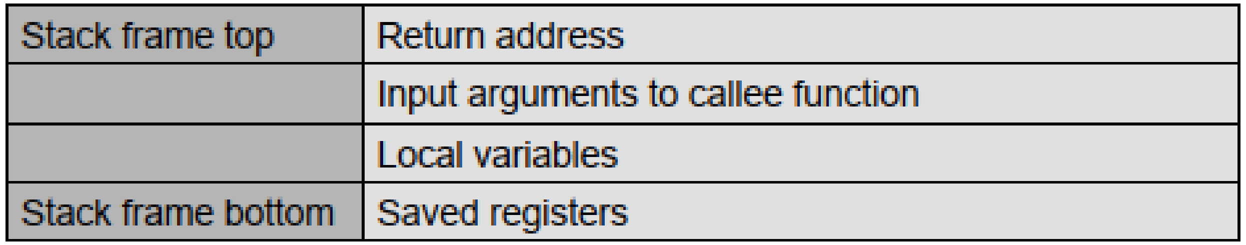
Stack Pointer →

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

## Microblaze

## Stack Frame

- A stack frame is a temporal storage or the function , it is composed of
  - Return Address
  - Local variables used by the function
  - Save registers that the function may modify, but caller function does not want changed.
  - Input arguments to the calle functions.

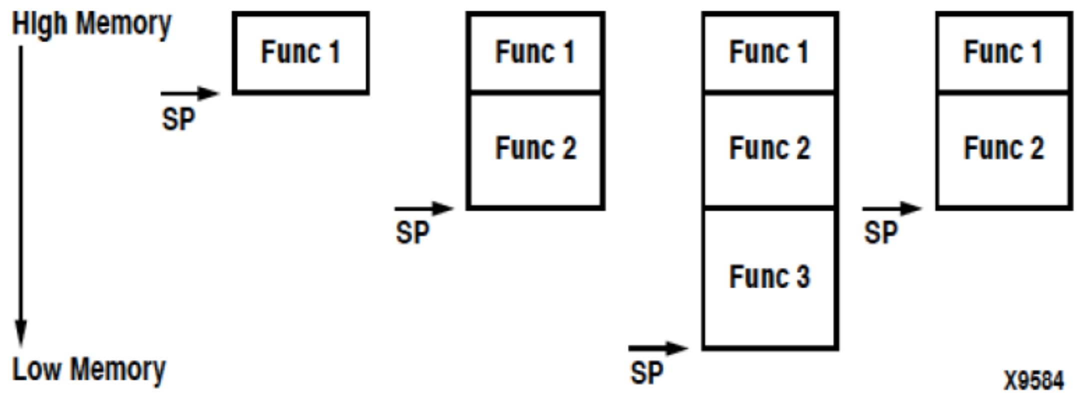


| | |
|---|---|
| Stack frame top | Return address |
| | Input arguments to callee function |
| | Local variables |
| Stack frame bottom | Saved registers |

Stack Pointer Points to **the top of the latest Stack Frame**

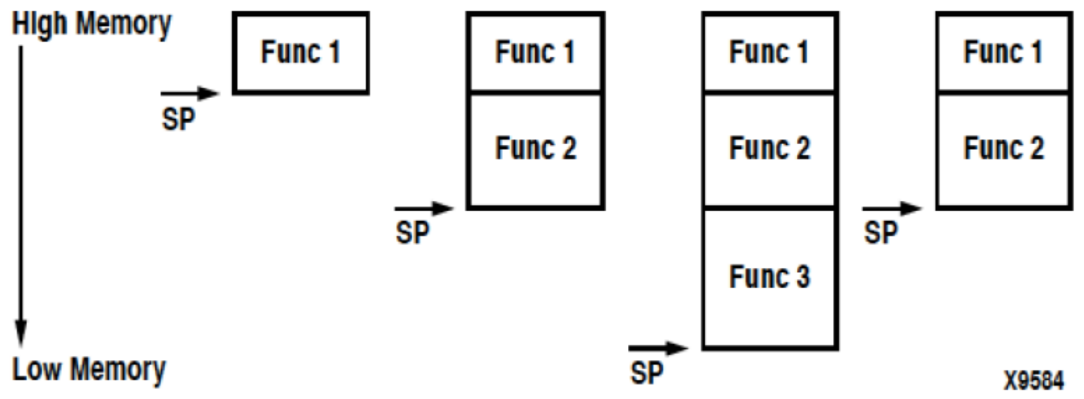
## Microblaze

### Stack Frame



- After a call from F1 to F2 the value of the SP is decremented.
- After call of F3 it is again decremented.
- After return from F3 to F2, SP is increased to its original value in F2.

## Microblaze
### Stack Frame



- Calling a Function:
  - Update the SP (PUSH)
  - Load the stack frame
- Returning from a function:
  - Restore the registers that have been previously saved.
  - Update the stack pointer (POP)

# Microblaze

## 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

## Microblaze

## Assembly Program Homework

```
.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
```

| | |
|---|---|
| 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 |

Microblaze instruction Memory

## Microblaze
## Assembly Program

Tips:

The C code equivalent of the previous assembly code is:

```
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;
}
```

Note:
**x>>=1** means that all 32 bits of x are shifted to the right (LSB is lost) and the result is saved in x.
**x & 1** produces a value that is either 1 or 0, depending on the least significant bit of x: if the last bit is 1, the result of x & 1 is 1; otherwise, it is 0. This is a bitwise AND operation.

## Microblaze
## Assembly Program Homework

Make a programm that identify if a number is palindrome. It means that the number is the same read from lef-to-rght and right-to-left.

```
.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
```

```
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;
}
```

peclab

POWER & ENERGY CONVERSION LABORATORY

Electrical Engineering Department

Pontificia Universidad Católica de Chile

peclab.ing.uc.cl