TU WIEN Informatics

Advanced Computer Architecture

Block B1: Processor, RISC-V ISA and Compiler Basics

Daniel Mueller-Gritschneder

Agenda

- Qick Recap: Processor Basics
- RISC-V Instruction Set Architecture
- Writing RISC-V Assembly Code
- Compiler Flow
- Compiler Frontend
- Intermediate Representation (IR)
- Example LLVM IR

B1-1 Recap - Processor Basics

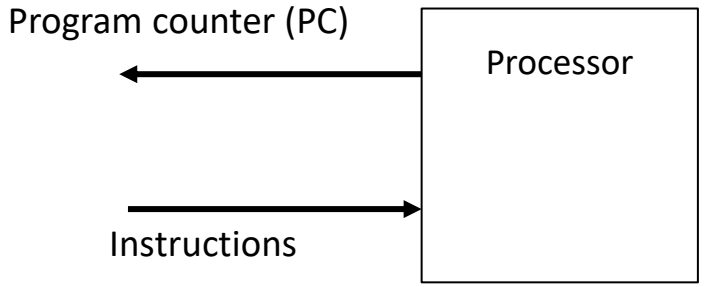
Processor: Instruction Interface

- A program consists of a list of instructions
- Typical an instruction is a 16/32/64 bit digital value
- The instruction is a single command the processor should execute, e.g., add two values (more on this later)
- The program counter (PC) points to the current position in the program, which is executed on the processor.
- PC is an address (16/32/64 bit digital value) where to load instructions from

Program

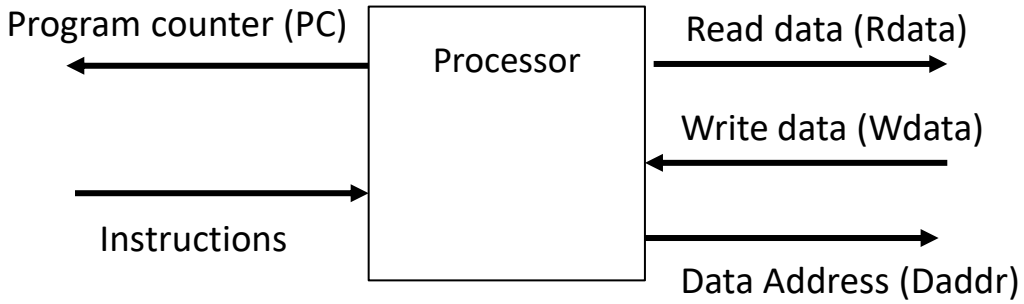
instr1 in machine code
instr2 in machine code
instr3 in machine code
instr4 in machine code
instr5 in machine code

…



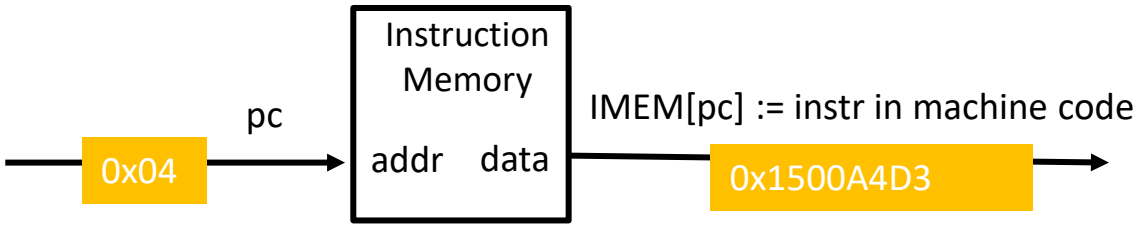
Processor: Data Interface

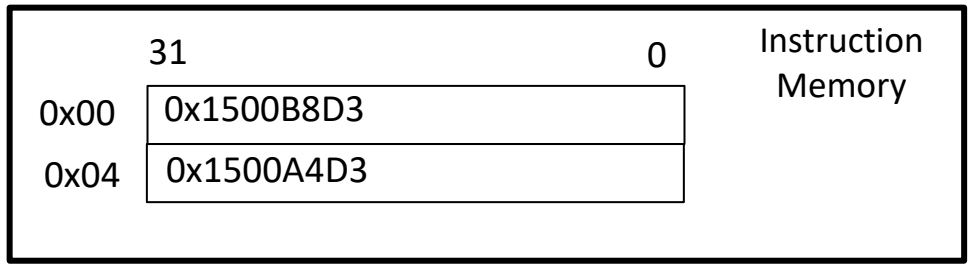
- Some instructions can read or write from the data interface
- Load instruction:
  - Puts an address (16/32/64 bit digital value) on the address bus
  - Receives a read value (16/32/64 bit digital value)on the Rdata bus
- Store instruction
  - Puts an address (16/32/64 bit digital value) on the address bus
  - Puts a write value (16/32/64 bit digital value)on the Wdata bus



Instruction Memory

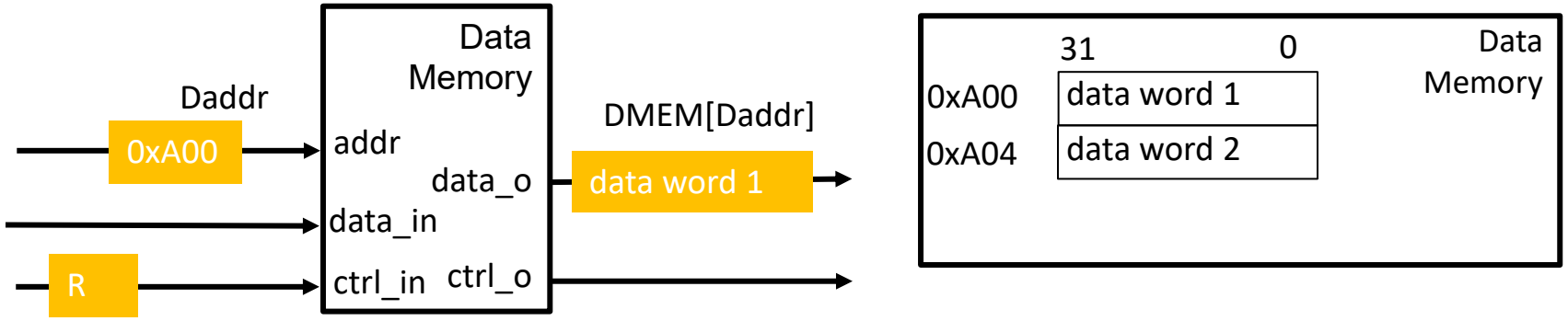
- Address is supplied (e.g. program counter PC)
- Returns the value stored at the input address mem[PC]



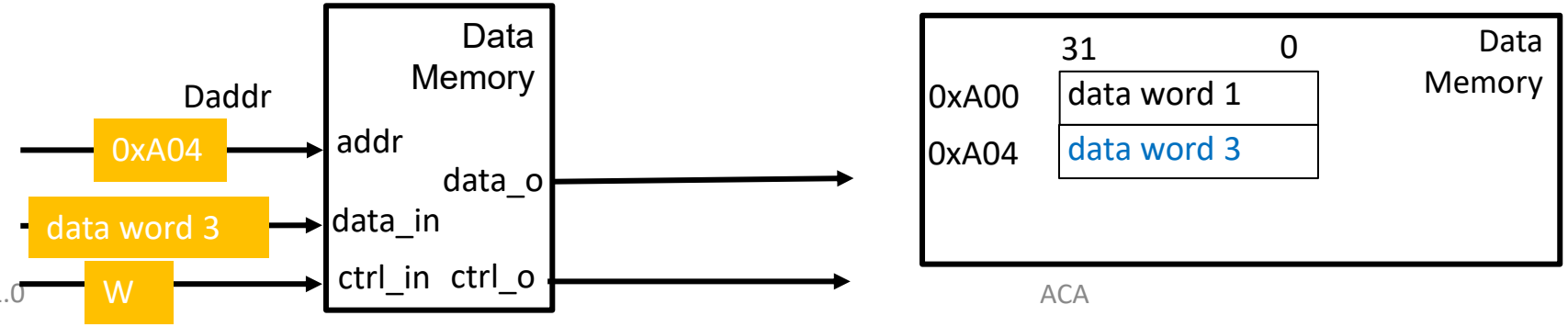


Data Memory

- **Read access**
  - Address is supplied
  - Returns the value stored at the input address mem[Daddr]

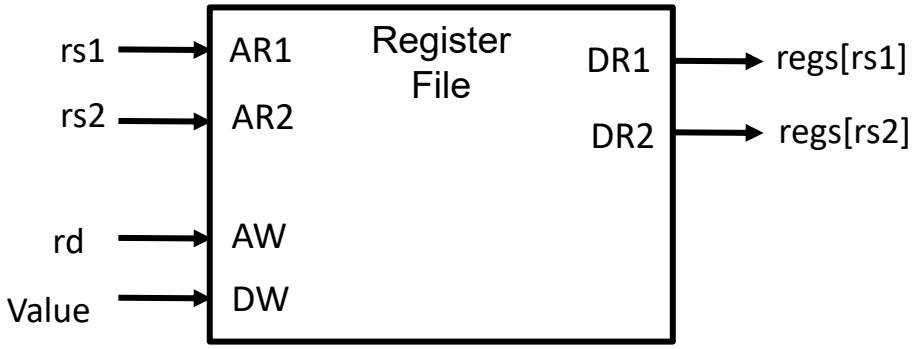


- **Write access**
  - Address is supplied
  - Returns the value stored at the input address mem[Daddr]

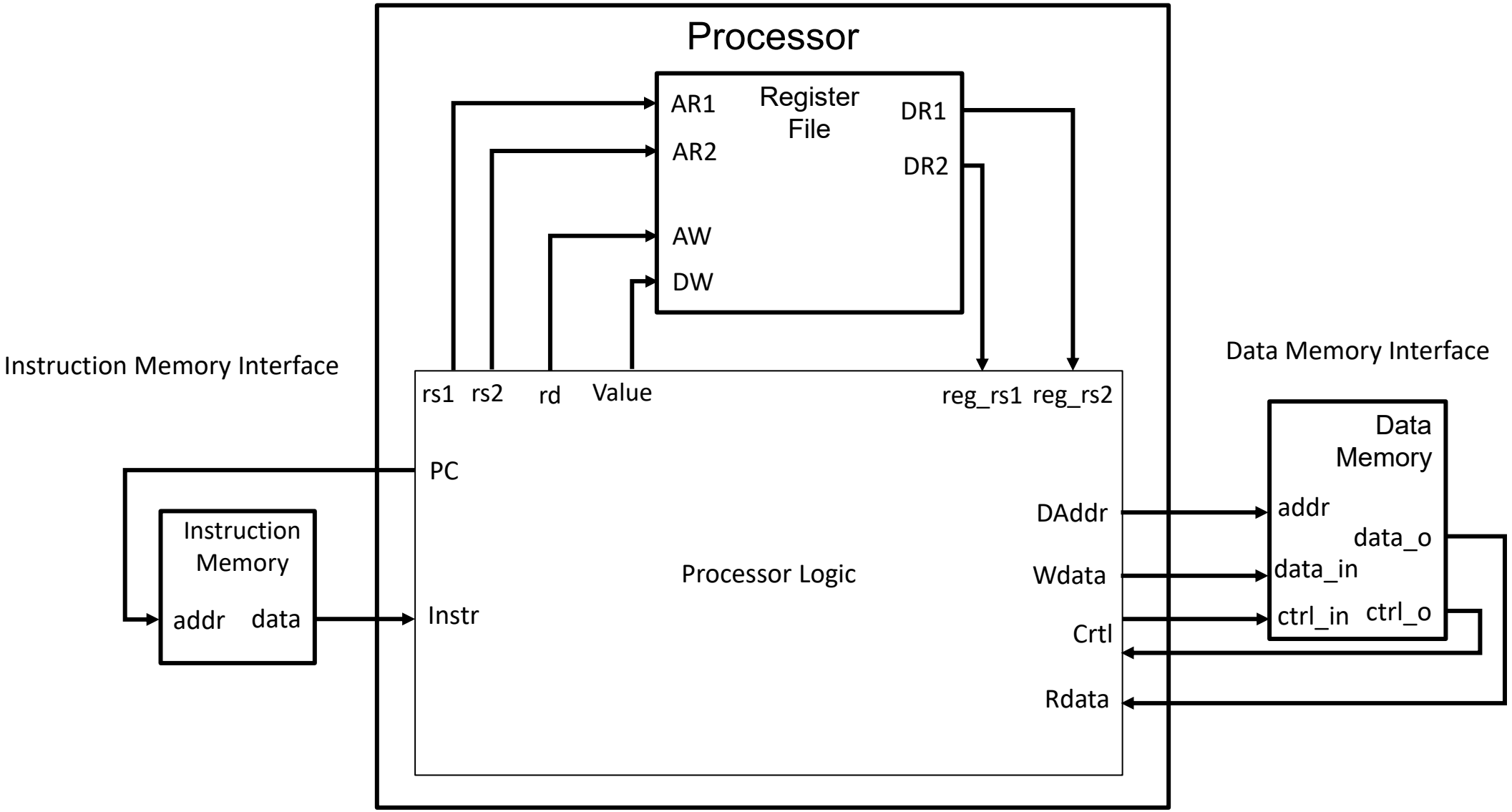


Register File Memory

- Processors save some values not in memory but in a bank of registers inside the processor
- The instructions needs to read usually more than one value from this bank and write usually one value
- The address is usually very small (4-5 bit) so the register file stores typically 16 or 32 values
- The register file is therefore usually realized with a special small SRAM, from which you can read two values and write one value
  - We give two read register addresses rs1 and rs2 and the memory returns to read values regs[rs1] and regs[rs2]
  - We give one write register address rd and the memory stores the write value regs[rD]=value



Register File is Inside Processor



B1-2 RISC-V Instruction Set Architecture

What is an Instruction Set Architecture (ISA)?

- The ISA describes:
  - the processor state organization (registers)
  - what instructions a processor executes.
    - How the instructions are encoded in machine code.
    - How the assembly of the instructions look likes.
  - Some more processor behavior (exceptions, …)
- A cross-compiler can generate the assembly code for a certain processor using the ISA (ISA is the interface between compiler and hardware)

- Instructions are the „*words*“ of the processor and the ISA is its „*vocabulary*“

Processor Terminology

- **Instruction set architecture (ISA):** Set of instructions that can be executed by the processor
- **Microarchitecture:** A processor model that describes the structure of the processor with the target ISA, e.g. nr. of pipeline stages
- **Implementation:** The processor implemented on a chip, e.g. in 90nm technology

Why RISC-V?

- Many embedded processors use the ARM ISA.
- ARM is an IP vendor and does not sell chips.
- Semiconductor companies such as NXP, TI, Qualcomm, Infineon buy ARM processor IPs and integrate them into their Micro-Controllers or System-on-chip (SoCs).

- RISC-V is an open ISA, which can be easily used in academia
- Invented by UC Berkeley.
- RISC-V is a very hot topic in industry and more and more RISC-V SoCs are becoming available.

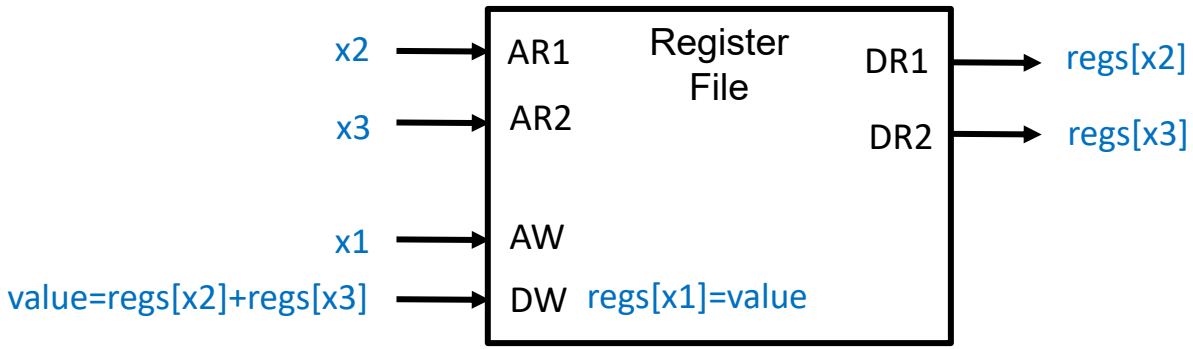
Assembly Instruction Built-up (1)

- Example instruction: `ADD x1,x2,x3`
  - ADD: Addition
  - x1: Destination register
  - x2,x3: Operand Registers
  - Behavior: Regs[x1] ← Regs[x2] + Regs[x3]

- We need to define:
  - The processor has a state Regs consisting of registers
  - Each register has an ID, e.g., here x1,x2,x3
  - Regs[ID] references the value stored in register ID
  - The assembly format of the ADD instruction, e.g., the first register ID is the destination, the second and third ID are the operands

Assembly Instruction Built-up (2)

- The state Regs consisting of registers (register file memory) is usually kept in an SRAM
- The Register-Register Instructions work on the register file memory inside the processor
- Example Addition: `ADD x1,x2,x3`
  - Behavior: Regs[x1] ← Regs[x2] + Regs[x3]



Registers of RISC-V

- RISC-V has 32 registers
- Each register can have different width, we look at RV32 with 32-bit width
- Each register has two IDs (x0-x31) and an ABI name that indicates its role

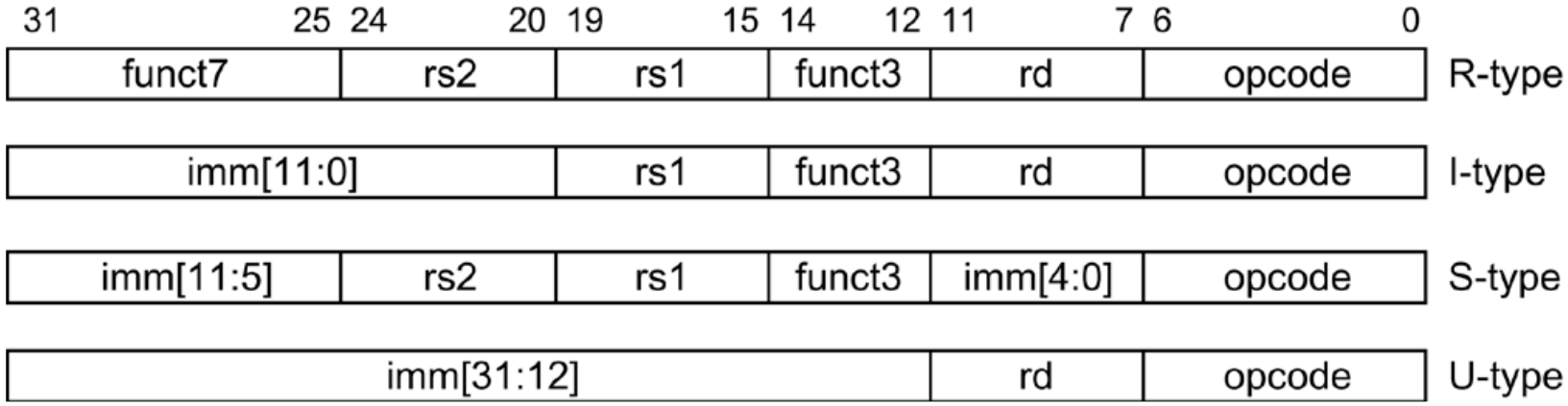
Register	ABI Name	Description	Saver
x0	Zero	Hard-wired zero	-
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	-
x4	tp	Thread pointer	-
x5-7	t0-2	Temporaries	Caller
x8	s0,fp	Saved register/frame pointer	Callee
x9	s1	Saved Register	Callee
x10-11	a0-1	Function arguments	Caller
x12-17	a2-7	Function arguments	Caller
x18-27	s2-11	Saved registers	Callee
x28-31	t3-6	Temporaries	Caller

Application Binary Interface (ABI)

- Specifies rules for register usage in passing arguments and results for function calls
  - Callee-saved registers: If function foo1 (caller) calls foo2 (callee), then foo2 is not allowed to modify this value (it needs to save it and restore it before returning to foo1
  - Caller-saved registers: If function foo1 (caller) calls foo2 (callee), then foo1 needs to save this register before calling foo2 if it wants to keep the value in it because foo1 is allowed to modify it
- Assigns aliases for registers x0-x31 (see table previous slide)

RISC-V Instruction Types

- Four types of instructions with different encoding in machine code



Instruction format	Primary use	rd	rs1	rs2	Immediate
R-type	Register-register ALU instructions	Destination	First source	Second source	
I-type	ALU immediates Load	Destination	First source base register		Value displacement
S-type	Store Compare and branch		Base register first source	Data source to store second source	Displacement offset
U-type	Jump and link Jump and link register	Register destination for return PC	Target address for jump and link register		Target address for jump and link

RISC-V Instructions

- The RISC-V ISA is structured into several instructions groups
  - We look at RV32IM
- 32-bit Integer Instruction RV32I
  - Integer Register-Register Instructions (R-type)
    - Runs an arithmetic or logical operation on registers
    - Both operands are values in registers
  - Integer Register-Intermediate Instructions (I-type)
    - Second operand is a immediate (constant) value
    - Immediate is encoded in the machine code
  - Control Transfer Instructions
    - Unconditional jumps
    - Conditional Branches
  - Load Store Instructions
    - Move data between memory and registers
    - Load-store Architecture: Operations on registers only
- 32-bit Integer Multiplication RV32M
  - Integer Multiplication Instructions
  - Integer Division Instructions

Integer Register-Register Instructions 1

- Instruction: `Instr xd,xm,xn`
- Instr: Assembly instruction name
- xd: Destination register
- xm,xn: Operand Registers
- Behavior: Regs[xd] ← Regs[xm] OP Regs[xn]

- Addition: `ADD a1,a2,a3`
  - Behavior: Regs[a1] ← Regs[a2] + Regs[a3]
- Subtraction: `SUB a1,a2,a3`
  - Behavior: Regs[a1] ← Regs[a2] - Regs[a3]

- Signed compare: `SLT a1,a2,a3`
  - Behavior: **if** (Regs[a2] < Regs[a3]) Regs[a1] ← 1 **else** Regs[a1] ← 0
- Unsigned compare: `SLTU a1,a2,a3`
  - Behavior: **if** (Regs[a2] < Regs[a3]) Regs[a1] ← 1 **else** Regs[a1] ← 0

Integer Register-Register Instructions 2

- Logical AND: `AND a1,a2,a3`
  - Behavior: Regs[a1] ← Regs[a2] & Regs[a3]
- Logical OR: `OR a1,a2,a3`
  - Behavior: Regs[a1] ← Regs[a2] | Regs[a3]
- Logical XOR: `XOR a1,a2,a3`
  - Behavior: Regs[a1] ← Regs[a2] ^ Regs[a3]

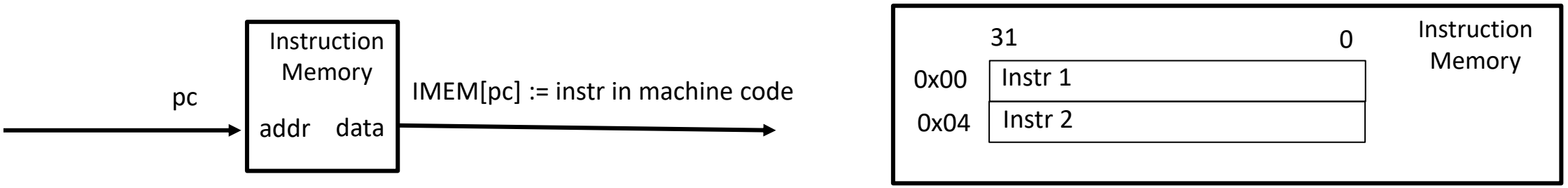
- Shift Left Logical: `SLL a1,a2,a3`
  - Behavior: Regs[a1] ← Regs[a2] << Regs[a3] (we shift 0s in from the right)
- Shift Right Logical: `SRL a1,a2,a3`
  - Behavior: Regs[a1] ← Regs[a2] >> Regs[a3] (we shift 0s in from the left)
- Shift Right Arithmetic: `SRA a1,a2,a3`
  - Behavior: Regs[a1] ← Regs[a2] >> Regs[a3] (if MSB is 1, we shift 1s in from the left, else 0s)

Integer Register-Immediate Instructions

- Format:
- Addition with immediate (constant value): Example: ADDI a1,a2,3
  - Behavior: Regs[a1] ← Regs[a2] + 3
- Further instructions as before but with immediate: SLTI, SLTIU, SLLI, SRLI, SRAI
- There is no SUBI: Use addition with negative immediate: ADDI a1,a2,-3
- No operation: NOP
  - Behavior: Does nothing
- Move: Example: MV a1,a2
  - Behavior: Regs[a1] ← Regs[a2]
  - Is a so-called pseudo instruction: The processor translates it to ADDI a1,a2,0

Program Counter and Instruction Memory

- The control transfer instructions change the program counter (pc)
- The pc tells from which address the next instruction should be fetched
- In RV32, each instruction is 32 bit or 4 byte
- The address is in bytes, so to jump from the one instruction to the next, the pc has to increment by 4.
- MEM[pc] is the instruction in the instruction memory stored at the address pc



- When there are Control Transfer Instructions, the PC is modified such that it jumps to another location different from the next instruction (PC+4) to implement
  - if, else, switch blocks
  - loops
  - function calls
  - function returns

Control Transfer Instructions - Jumps

- Unconditional Jump (PC relative): `J imm // pseudo instr. for J x0,imm`
  - Behavior: pc ← pc + (imm<<1)
  - Example: `J 8` has behavior: pc ← pc + (8<<1) = pc + 16
  - But we usually do not put the offset, but a symbol, e.g., start of loop that we want to jump to, e.g., pc = loop_start by writing `J loop_start`

- Unconditional Jump and Link (PC relative): Example: `JAL rd,imm`
  - Behavior: pc ← pc + (imm<<1), Regs[rd] ← pc + 4
  - Example: `JAL ra,8`
    - Behavior: pc ← pc + (8<<1) = pc + 16
      Regs[ra] ← pc + 4
    - Here, we want to jump to a function. In order to be able to return, we save the next instruction address (pc+4) in the return address (ra) register regs[ra] = regs[x1]
  - So again here we do not put the offset but the symbol of the function: `JAL foo1`

- Unconditional Jump Register (Register with offset): Example: `JALR rd,rs1,imm`
  - Behavior: pc ← Regs[rs1]+imm & ~1, Regs[rd] ← pc+4
  - This is usually used for function return. The return addressed is saved in register ra.
  - Example: A pseudo-instruction is `RET // pseudo instr. for JR x0,ra,0`
    Behavior: pc ← Regs[ra], x0 stay always 0

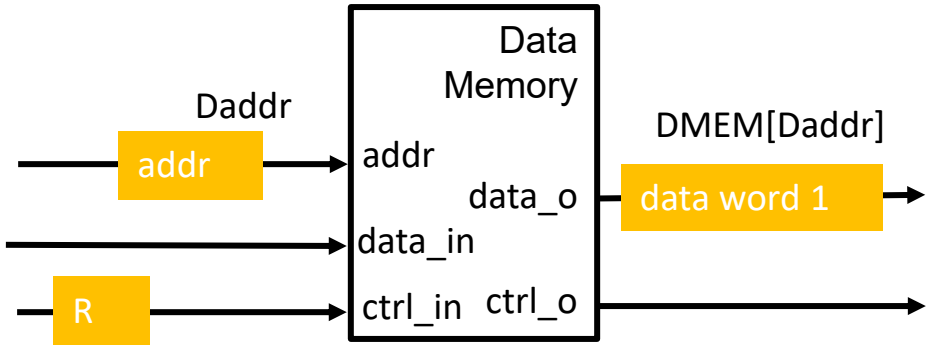
Control Transfer Instructions - Branches

- Conditional branch equal zero: `BEQ a1,a2, loop_start`
  - Behavior: **if** (regs[a1] == regs[a2]) pc = loop_start **else** nothing

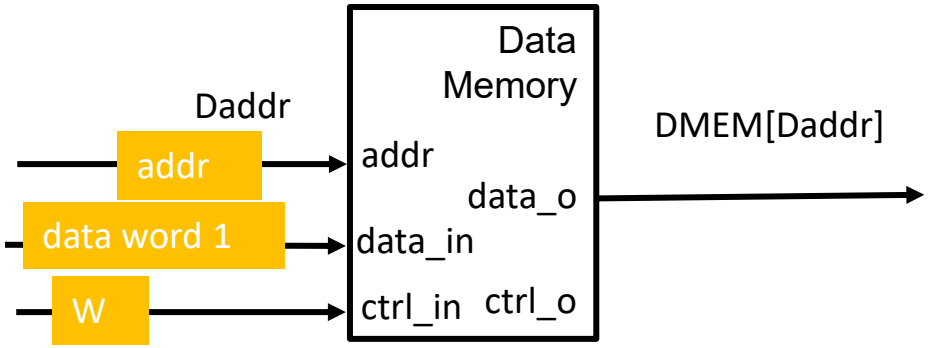
- Further branch instructions
  - not equal: `BNE a1,a2, loop_start`
  - Lesser than: `BLT a1,a2, loop_start`
  - Unsigned lesser than: `BLTU a1,a2, loop_start`
  - Greater or equal than: `BGE a1,a2, loop_start`
  - Unsigned greater of equal than: `BGEU a1,a2, loop_start`

Data Memory

- Load and store instructions access the data memory (data, stack or heap)
- MEM[Daddr] is the value in the data memory stored at the address addr
- The load instruction fetches a value on data memory

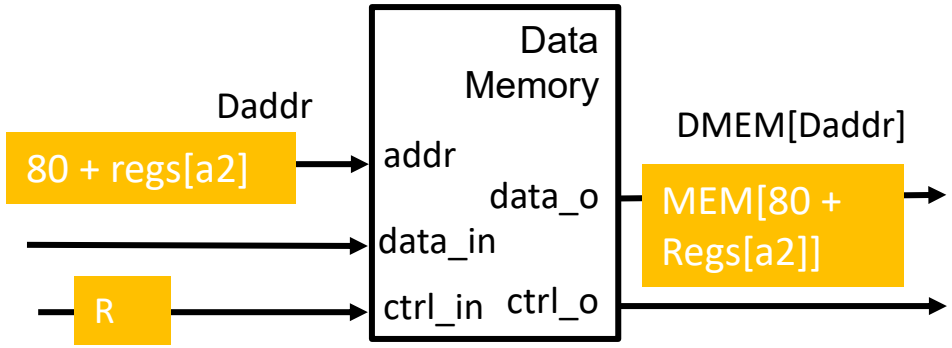


- The store instructions saves a value in data memory



Load and Store Instructions

- Load word: `LW a1,80(a2)`
  - Behavior: Regs[a1] ← MEM[80 + Regs[a2]]
  - We set `(a2)` to indicate that the value in a2 is used as an address, 80 is the offset
  - Loads a word (4 byte)



- Store word: `SW a1,80(a2)`
  - Behavior: MEM[80 + Regs[a2]] ← Regs[a1]
  - We set `(a2)` to indicate that the value in a2 is used as an address, 80 is the offset
  - Stores a word (4 byte)

- Other instructions used to store half words (2 byte) or bytes

Integer Multiplication Instructions

- Signed-signed Multiplication
  - Multiplying two 32bit values can result in a value of up to 64bit
  - `MUL rdl,rs1,rs2`
    - Behavior: regs[rdl] ← regs[rs1]*regs[rs2] // only the lower 32bit
  - `MULH rdh,rs1,rs2`
    - Behavior: regs[rdl] ← regs[rs1]*regs[rs2] // only the higher 32bit
  - Example:
    - ```
      MUL a3,a1,a2
      MULH a4,a1,a2
      ```
    - Behavior: [regs[a4] regs[a3]] = regs[ra1]*regs[a2] // full 64 bit

- Unsigned-unsigned multiplication `MULU, MULHU`
- Unsigned-signed multiplication `MULSU, MULHSU`
      ```

Integer Division Instructions

- Signed-signed Division
  - `DIV rdl,rs1,rs2`
    - Behavior: regs[rdl] ← regs[rs1] / regs[rs2]
  - `REM rdh,rs1,rs2`
    - Behavior: regs[rdl] ← regs[rs1] modulo regs[rs2] // remainder
  - Example: `DIV a3,a1,a2`
    Behavior: regs[a3] = regs[ra1] / regs[a2]

- Unsigned-unsigned division `DIVU, REMU`
- Unsigned-signed division `DIVSU, REMSU`

B1-3

Writing a function in assembly

- Start with a symbol with the function name `foo1:`
- The first function parameter is in a0, second in a1, …
- The return value should be in a0 before returning
- For returning use the `RET` instruction

- Guideline: Use tx registers for temporary local variables

Writing a small assembly function 1

- Example C-Code 1

```c
// Computes the absolute value
int abs_value(int a) {
    if (a<0)
        a=0-a;
    return a;
}
```

- RISC-V Code
  - According to ABI a is given to the function in register a0
  - The function should also return a in register a0

```
abs_value:
        BGE a0,zero,abs_value_return //if a>=0
        SUB a0,zero,a0      // a=0-a
abs_value_return:
        RET                 // JR x0,ra,0
```

function return

Writing a small assembly function 2

- Example C-Code 2

```c
// calculate greatest common divisor (gcd)
// require: x >= 0 && y > 0
int gcd(int a, int b) {
  int t;
  while(a != 0) {
    if(a >= b) {
      a = a - b;
    }
    else {
      t = a; a = b; b = t;
    }
  }
  return b;
}
```

RISC-V Code

```
//a: a0, b: a1, t: t0
gcd:
  BEQZ a0, gcd_done       // while(a!=0)
  BLT a0, a1, gcd_else    // a < b -> else
  SUB a0, a0, a1          // a = a-b
  J gcd                   // while loop
gcd_else:
  MV t0, a0               // t = a
  MV a0, a1               // a = b
  MV a1, t0               // b = t
  J gcd                   // while loop
gcd_done:                 // now a1 contains the gcd
  MV a0, a1               // move to a0 for returning
  RET                     // return (jr ra)
```

Writing a small assembly function 3

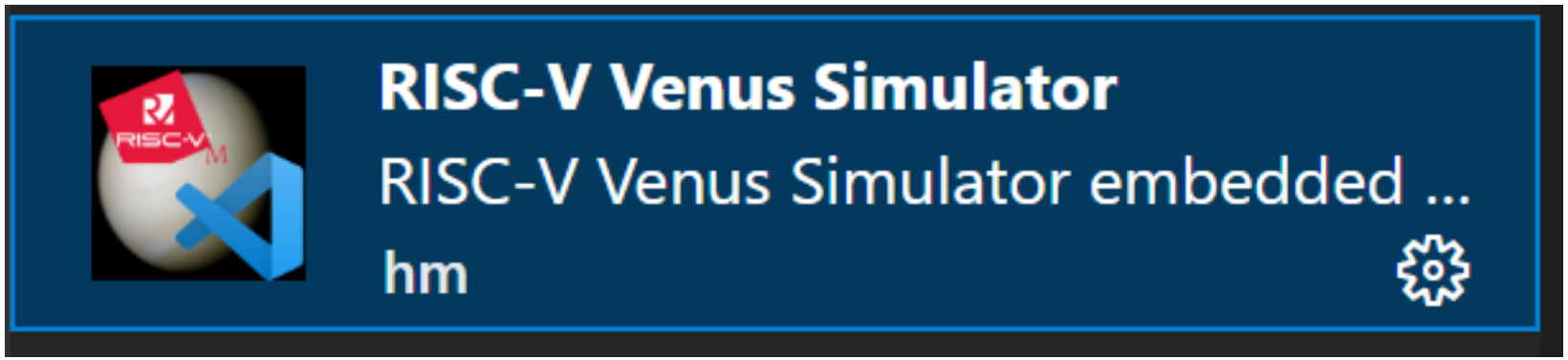
- Example C-Code 3

```
// vector addition of 4-element integer vectors
void vec_add(int[4] a, int[4] b, int[4] c) {
  unsigned int i;
  for (i=0;i<4;i++) {
    c[i] = a[i] + b[i];
  }
}
```

RISC-V Code

```
// base address of a: a0,
// base address of b: a1,
// base address of c: a2,
// i: t0,  constant 4: t3
vec_add:
  LI t0,0        // i=0
  LI t3,4        // t3=4
vec_add_for:
  LW t1,0(a0)     // t1 = a[i]
  LW t2,0(a1)     // t2 = b[i]
  ADD t1,t1,t2    // t1 = a[i] + b[i]
  SW t1,0(a2)     // c[i] = t1
  ADDI a0,a0,4    //next element is base address + 4
  ADDI a1,a1,4    //next element is base address + 4
  ADDI a2,a2,4    //next element is base address + 4
  ADDI t0,t0,1    // i++
  BLTU t0,t3,vec_add_for // for (i < 4)
  RET   // void return
```

- RISC-V Simulator



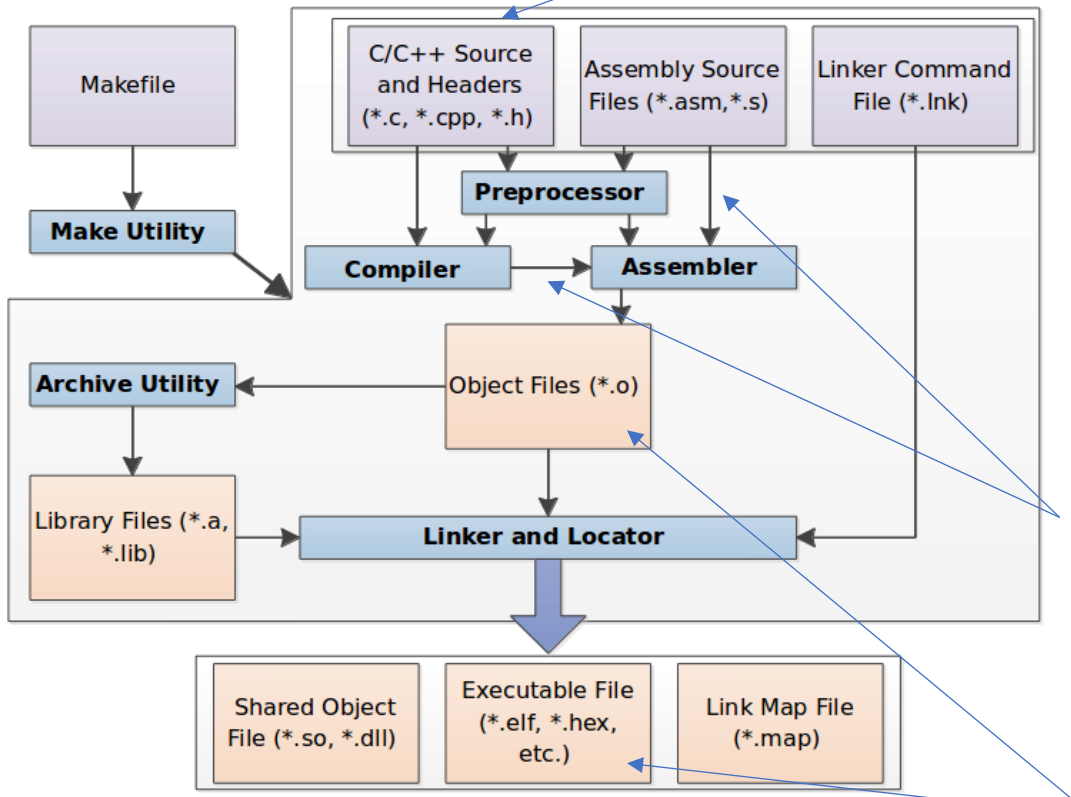
Extensions -> Venus RISC-V Simulator

B1-4 Compiler Flow

Compilation

C-code:

```
val1=val1+4;
```



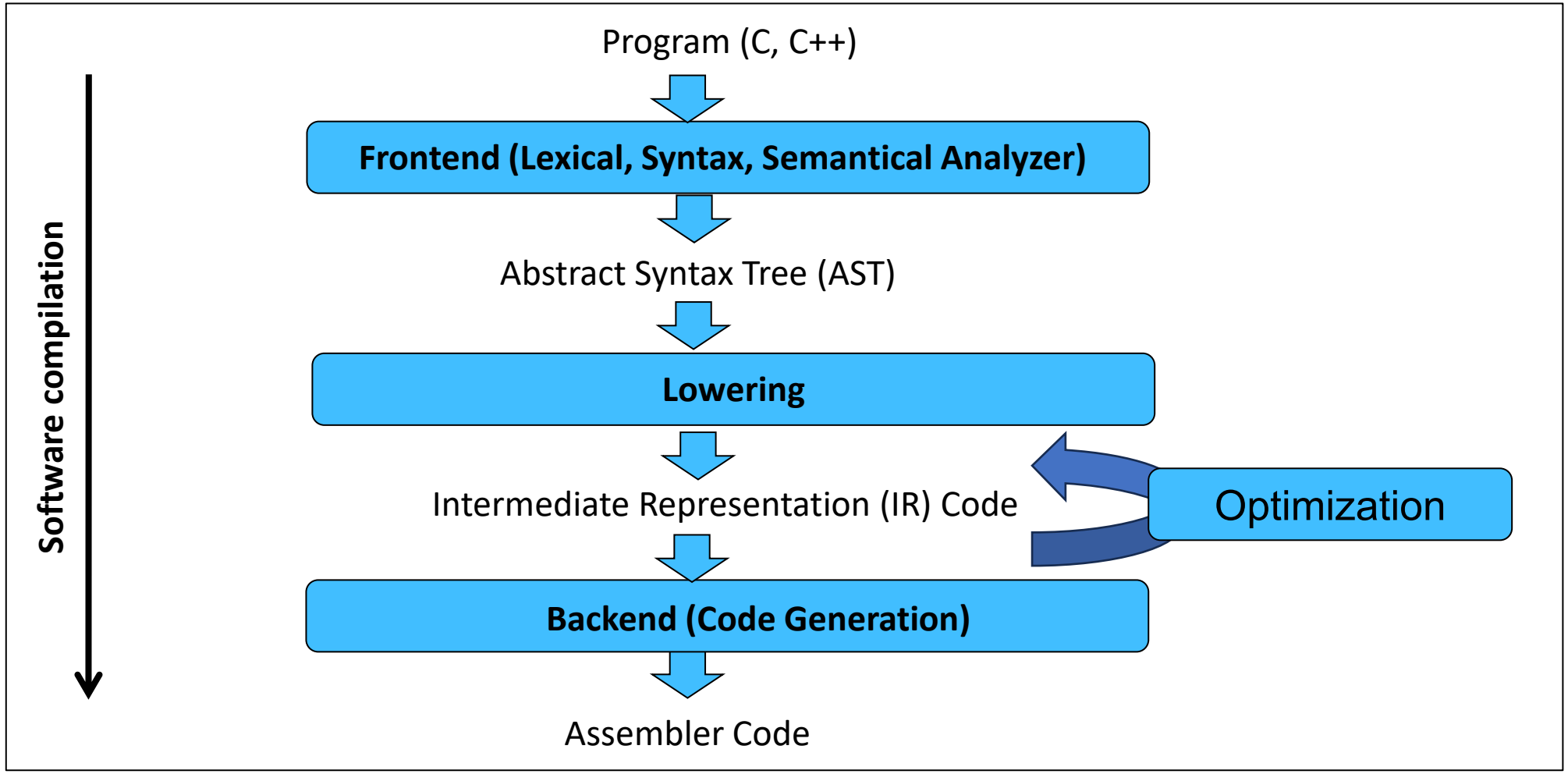
Assembly-code:

```
ADDI x10,x10,4
```

Machine code:

```
0x00450513
```

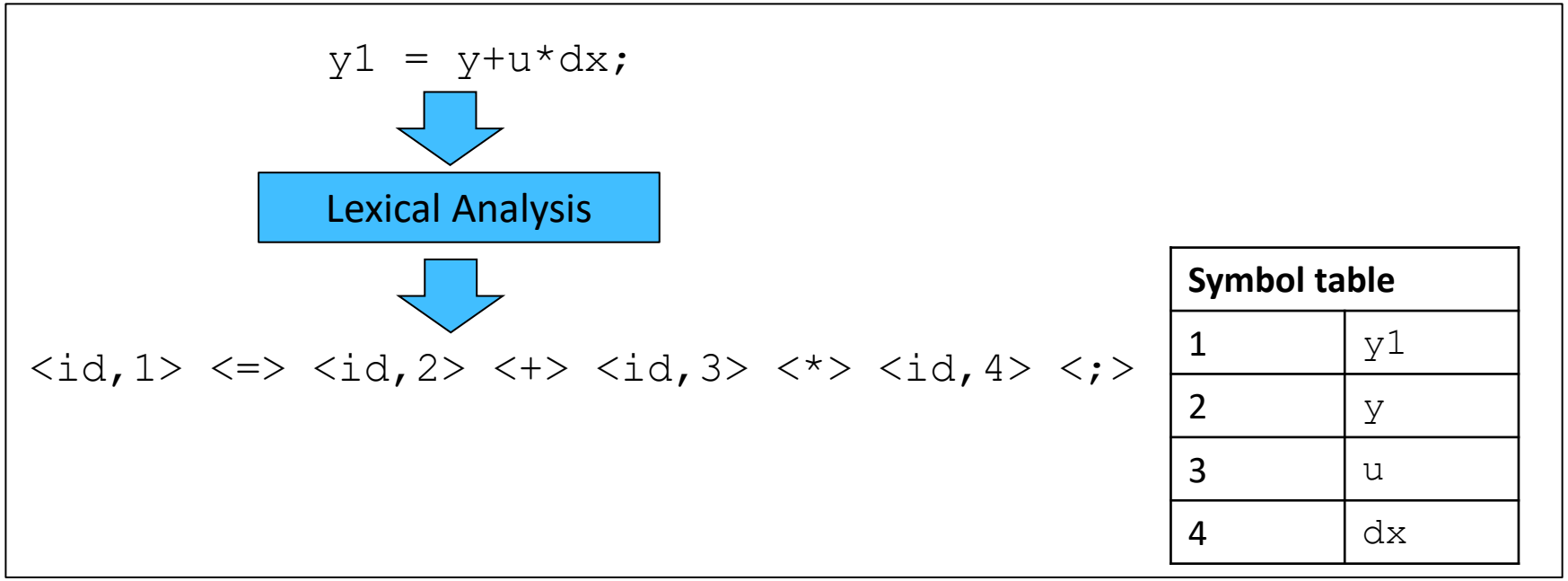
Compiler Frontend and Backend



B1-5 Compiler Frontend

Lexical Analysis (Scanning)

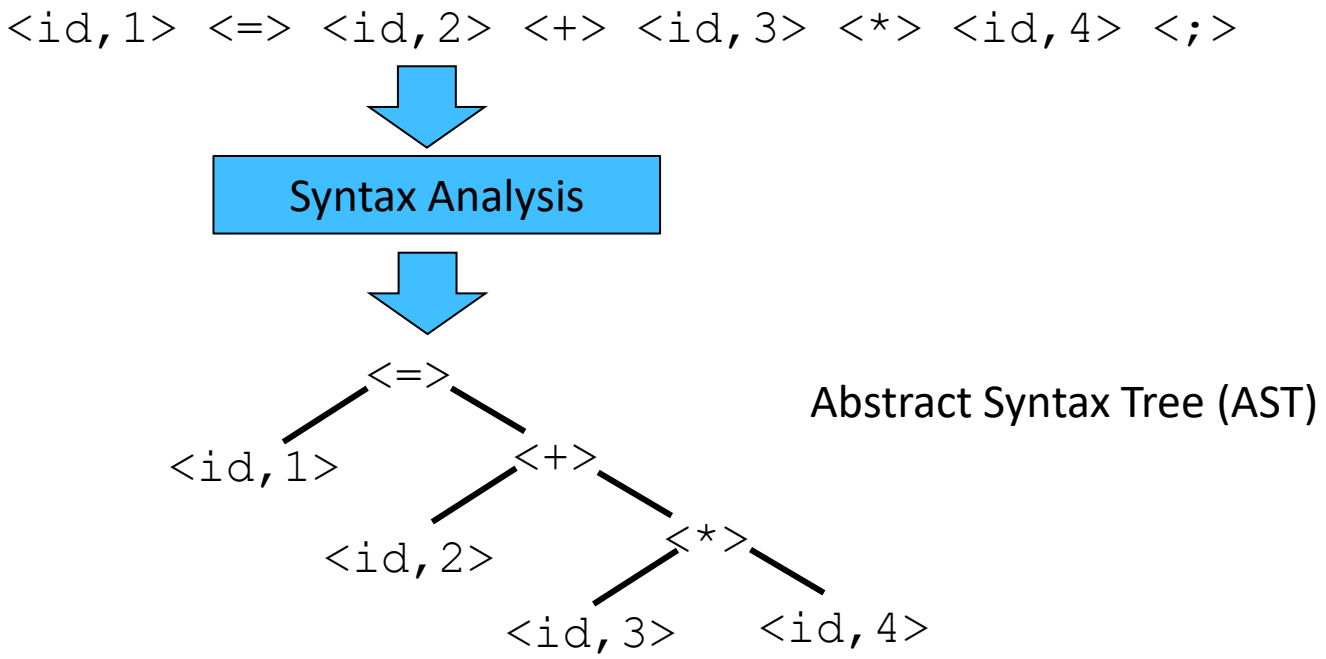
- Reads stream of characters
- Groups characters in meaningful sequences (lexemes)
- Outputs token stream and symbol table



Symbol table	
1	y1
2	y
3	u
4	dx

Syntax Analysis (Parsing)

- Reads token stream
- Outputs syntax tree (parse tree) that depicts syntactical structure of token stream



Semantical Analysis

- Reads abstract syntax tree
- Checks against semantics of programming language
- Inserts type casts.
- Outputs semantical correct syntax tree.

B1-6 Intermediate Representation (IR)

Three address code (1/4)

- Address: Reference to
  - variable name,
  - constant,
  - Compiler-generated temporary variable name.

- Maximal 3 addresses per operation.
- At most one operator at right side of operation.

Three address code (2/4)

- Assignment:

```
x := y op z   with op ∈ {+,-,*,\,^,&,…}
x := op y     with op ∈ {-,!,…}
```

- Copy:

```
x := y
```

- Unconditional jump:

```
goto Bx    Bx: label
```

- Conditional branch:

```
if x relop y goto Bx
with relop ∈ {=,<=,>=,<,>,!=,..}
```

- Label:

```
Bx: statement
```

Three address code (3/4)

- Procedure call:

```
param p1
param p2
…
y := call proc,n
```

with `n`: Number of parameters

- Return parameter

```
return y
```

- Indexed copy instruction:

```
x := y[i]
x[i]:= y
```

- Pointer assignment:

```
x := &y   //Get address of y and store it in x
x := *y   //Get value stored in address y and store it in x
```

Three address code (4/4)

- Example: Three address code for DE-Solver
- Compiler generated temporary variables: `t1 – t7`

C-Code section

```
repeat {
          x1 = x+dx;
          u1 = u-3*x*u*dx-3*y*dx;
          y1 = y+u*dx;
          x=x1;u=u1;y=y1;
} until (x1 < a);
```

Three address code

```
B1: x1 := x+dx;
    t1 := y*dx;
    t2 := 3*t1;
    t3 := u*dx;
    t4 := x*t3;
    t5 := 3*t4;
    t6 := u-t5;
    u1 := t6-t2;
    t7 := u*dx;
    y1 :=  y+t7;
    x:=x1;
    u:=u1;
    y:=y1;
    if x1 >= a goto B1;
```

Static Single Assignment Form (SSA)

- All assignments are to variables with distinct names

Three address code:

```
p := a+b
q := p-c
p := q*d
```

SSA:

```
p$1 := a+b
q := p$1-c
p$2 := q*d
```

- The Φ PHI-operator chooses the assigned value for recombination of two values of one variable:

SSA:

```
     if (a>b) goto B1
     p$1 := a-b
     goto B2
B1:  p$2 := a+b
B2:  p$3:=Φ(p$1,p$2)
```

B1-7 Example LLVM IR

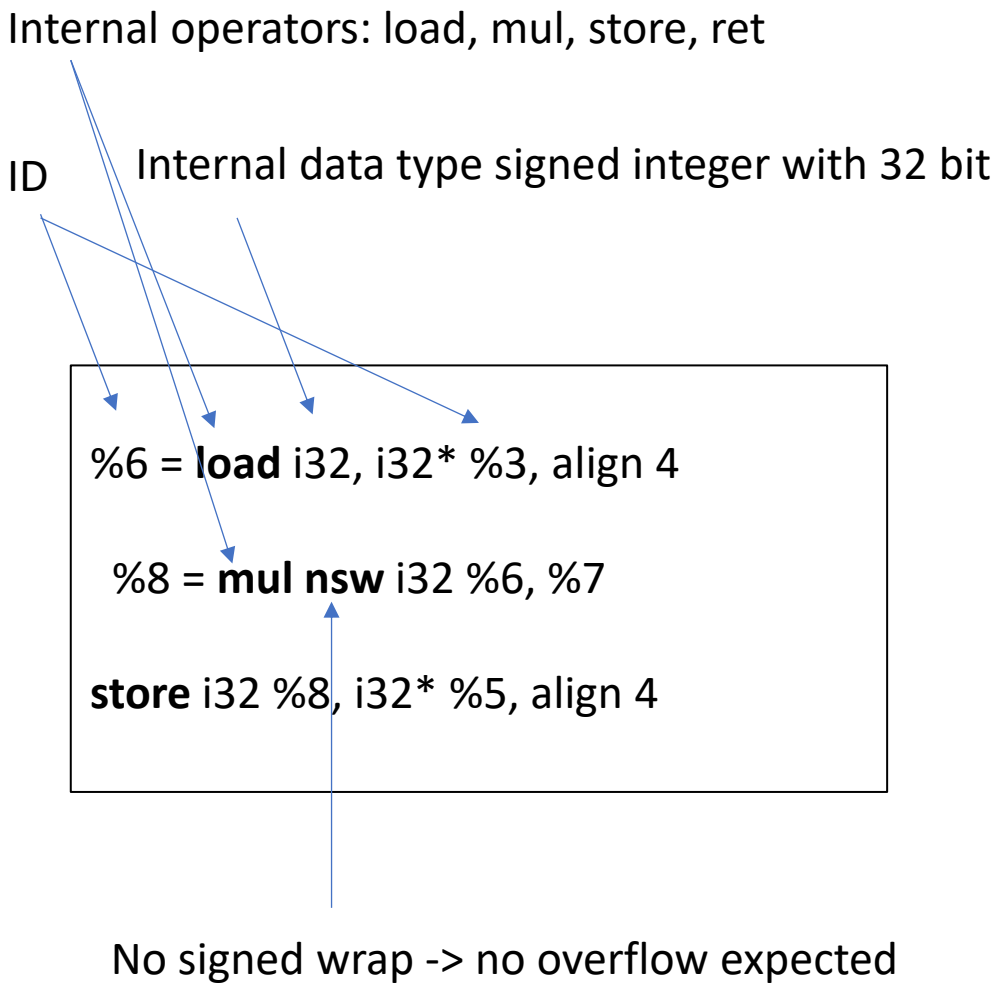
LLVM - IR

- LLVM
  - Compiler framework
  - CLANG is the C/C++ frontend
  - LLVM has Backends for many targets (x86, ARM, RISC-V, …)
  - Under active development: Currently LLVM 18

- LLVM IR
  - Intermediate representation
  - Static Single Assignment Form (SSA)
  - Evolves with LLVM versions, but usually minor changes

LLVM - IR - Syntax

- IDs are marked with %
- Has many buildin internal operators:
  - Arithmetic: **add**, **mul**
  - Memory: **load** and **store**
  - Stack allocation: **alloca**
  - Also for Vectors
- Attributes: Refine the behavior and define what the optimizer is allowed.
  - align 4: Address should be aligned to a 4-byte boundary
  - nsw: (No **Signed** Wrap).
    - If the mul overflows this would lead to so-called undefined behavior (UB) in C
    - nsw means that no overflow for mul is expected such that the compiler can optimize the code as if an overflow cannot happen.



LLVM – IR – Example with no Optimization

File test.c

```
int test1(int a, int b)
{
  int c = a*b;
  return c;
}
```

Compiler Frontend and Lowering

➢ `clang -S -emit-llvm test.c -o test_noopt.ll --target=riscv32`

File test_noopt.ll:

```
; ModuleID = 'test.c'
source_filename = "test.c"
(…)

; Function Attrs: noinline nounwind optnone uwtable
define dso_local i32 @test1(i32 noundef %0, i32 noundef %1) #0 {
  %3 = alloca i32, align 4
  %4 = alloca i32, align 4
  %5 = alloca i32, align 4
  store i32 %0, i32* %3, align 4
  store i32 %1, i32* %4, align 4
  %6 = load i32, i32* %3, align 4
  %7 = load i32, i32* %4, align 4
  %8 = mul nsw i32 %6, %7
  store i32 %8, i32* %5, align 4
  %9 = load i32, i32* %5, align 4
  ret i32 %9
}
(…)
```

LLVM – IR – Example with no Optimization

```
; ModuleID = 'test.c'
source_filename = "test.c"
(…)

; Function Attrs: noinline nounwind optnone uwtable
define dso_local i32 @test1(i32 noundef %0, i32 noundef %1) #0 {
  %3 = alloca i32, align 4
  %4 = alloca i32, align 4
  %5 = alloca i32, align 4
  store i32 %0, i32* %3, align 4
  store i32 %1, i32* %4, align 4
  %6 = load i32, i32* %3, align 4
  %7 = load i32, i32* %4, align 4
  %8 = mul nsw i32 %6, %7
  store i32 %8, i32* %5, align 4
  %9 = load i32, i32* %5, align 4
  ret i32 %9
}
(…)
```

Stack frame allocation, deallocation automatically at return

Store function parameters to stack

Load function parameters from stack

Store return value to stack

Load return value from stack

We have used no compiler optimization.

In this case, the compiler makes all operations explicit and does not optimize the code in any way (such as for reuse of registers).

All function parameters and return values are copied to and from the stack once. This is useful for running the program with a debugger as all data is available in memory.

LLVM – IR – Example with no Optimization

```
(…)

; Function Attrs: noinline nounwind optnone uwtable
define dso_local i32 @test1(i32 noundef %0, i32
noundef %1) #0 {
  %3 = alloca i32, align 4
  %4 = alloca i32, align 4
  %5 = alloca i32, align 4
  store i32 %0, i32* %3, align 4
  store i32 %1, i32* %4, align 4
  %6 = load i32, i32* %3, align 4
  %7 = load i32, i32* %4, align 4
  %8 = mul nsw i32 %6, %7
  store i32 %8, i32* %5, align 4
  %9 = load i32, i32* %5, align 4
  ret i32 %9
}
(…)
```

RISC-V Compiler Backend

Test_noopt.S

```
                .text
                .attribute      4, 16
                .attribute      5, "rv32i2p0_m2p0_a2p0_c2p0"
                .file           "test.c"
                .globl          test1
                .p2align        1
                .type           test1,@function
test1:
                addi            sp, sp, -32
                sw              ra, 28(sp)
                sw              s0, 24(sp)
                addi            s0, sp, 32
                sw              a0, -12(s0)
                sw              a1, -16(s0)
                lw              a0, -12(s0)
                lw              a1, -16(s0)
                mul             a0, a0, a1
                sw              a0, -20(s0)
                lw              a0, -20(s0)
                lw              ra, 28(sp)
                lw              s0, 24(sp)
                addi            sp, sp, 32
                ret
.Lfunc_end0:
                .size           test1, .Lfunc_end0-test1
(…)
```

```
> clang test.c -S -o test_noopt.S --target=riscv32
```

LLVM – IR – Example with Optimization

File test.c

```
int test1(int a, int b)
{
  int c = a*b;
  return c;
}
```

Compiler Frontend and Lowering

```
> clang -S -emit-llvm test.c
  -o test_opt.ll -O2 --target=riscv32
```

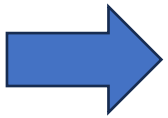
File test_opt.ll

```
; ModuleID = 'test.c'
source_filename = "test.c"
(…)

; Function Attrs: mustprogress nofree norecurse nosync nounwind
readnone uwtable willreturn
define dso_local i32 @test1(i32 noundef %0, i32 noundef %1)
local_unnamed_addr #0 {
  %3 = mul nsw i32 %1, %0
  ret i32 %3
}
(…)
```

LLVM – IR – Example with Optimization

test.c

```
; ModuleID = 'test.c'
source_filename = "test.c"
(…)

; Function Attrs: mustprogress nofree norecurse nosync
nounwind readnone uwtable willreturn
define dso_local i32 @test1(i32 noundef %0, i32
noundef %1) local_unnamed_addr #0 {
  %3 = mul nsw i32 %1, %0
  ret i32 %3
}
(…)
```



RISC-V Compiler Backend

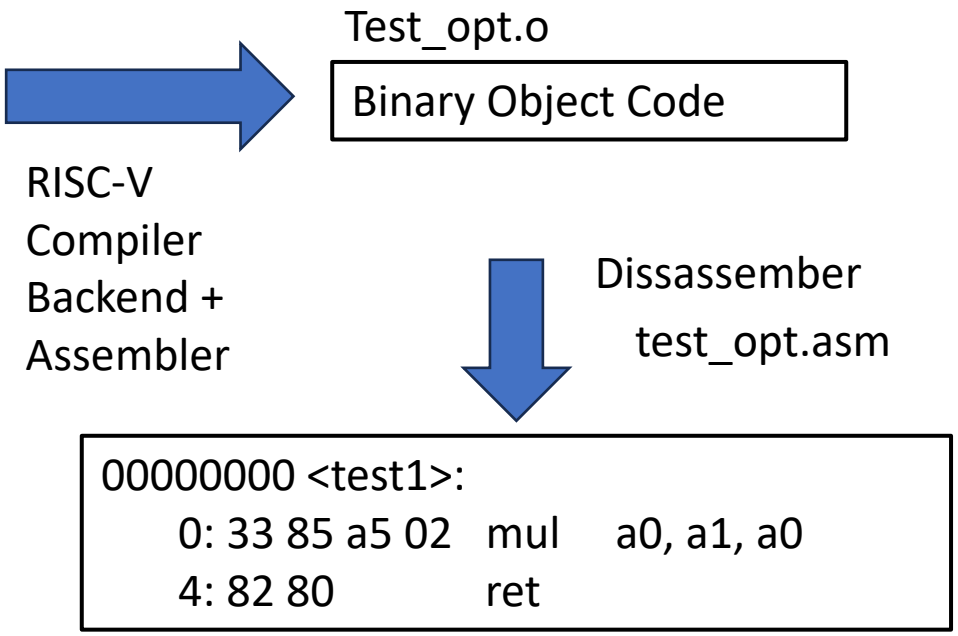
test_opt2.S

```
            .text
            .attribute 4, 16
            .attribute 5, "rv32i2p0_m2p0_a2p0_c2p0"
            .file      "test.c"
            .globl     test1
            .p2align   1
            .type      test1,@function
test1:
            mul        a0, a1, a0
            ret
.Lfunc_end0:
            .size      test1, .Lfunc_end0-test1

            (…)
```

```
> clang test.c -S -o test_opt2.S -O3
--target=riscv32
```

LLVM – IR – Example with Optimization

```
; ModuleID = 'test.c'
source_filename = "test.c"
(…)

; Function Attrs: mustprogress nofree norecurse nosync nounwind
readnone uwtable willreturn
define dso_local i32 @test1(i32 noundef %0, i32 noundef %1)
local_unnamed_addr #0 {
  %3 = mul nsw i32 %1, %0
  ret i32 %3
}
(…)
```




```
>clang -c test_opt.ll --target=riscv32 -o test_opt.o

>llvm-objdump -d test_opt.o > test_opt.asm
```

Summary

- Quick Recap: Processor Basics
- RISC-V Instruction Set Architecture
- Writing RISC-V Assembly Code
- Compiler Flow
- Compiler Frontend
- Intermediate Representation (IR)
- Example LLVM IR

Outlook

- How does the assembly code look like?
- How does assembly code look like for the RISC-V processor?

- How do we compile from C to Assembly Code.