Compiler Design - WS 2017/18, TU Berlin - Lecturer: Dr. Biagio Cosenza

Exercise 4: LLVM MIPS backend

In this assignment you will implement an LLVM backend for the 32-bit MIPS architecture, which converts LLVM IR into MIPS assembly. To simplify this assignment, we will not use the LLVM backend mechanism (which uses several lower-level IRs which have not been discussed in the lecture) and instead use a module pass to directly print out assembly to the error stream. Additionally, we will mostly neglect issues relating to register allocation, calling convention and instruction scheduling.

As an example, the backend should convert the LLVM IR of the following test function into MIPS32 assembly similar (but not necessarily identical) to the one shown below. Finally, the SPIM simulator is used to execute the assembly. The output of the simulator will also be used for grading.

```
# Input C code
int main() {
    int a = 1;
    int b = 2;
    print_int(a + b);
    print_nl();
    print_int(a - b);
    return 0;
}

# Generated MIPS assembly
main:
  addi $s0, $zero, 1  # $s0 = 1
  addiu $s1, $s0, 2   # $s1 = $s0 + 2
  move $a0, $s1       # $a0 = $s1
  jal print_int       # print_int($a0)
  jal print_nl        # print_nl()
  addi $s2, $zero, 1  # $s2 = 1
  addiu $s3, $s2, -2  # $s3 = $s2 + (-2)
  move $a0, $s3       # $a0 = $s3
  jal print_int       # print_int($a0)
  addi $a0, $zero, 0  # exit(0)
  addi $v0, $zero, 17 # (17 is the exit syscall ID)
  syscall

# Output of MIPS simulator (SPIM)
3
-1
```

Task

The provided pass.cpp template provides a number of helper functions and already implements handling for some instruction types, including addition, call, return and getelementptr instructions. Your task is to complete the implementation with support for the following instructions:

- Basic binary operators: Subtraction, Multiplication, Signed/Unsigned Division and Modulo, Bitwise And/Or/Xor, Shift Left, Shift Right Arithmetic/Logical.
- Unconditional and conditional branches. This will also require appropriate handling of Phi nodes.
- Memory load operations, supporting different bit-width (8, 16, 32 bit) and signedness.

Apart from using immediate operands where possible, your implementation does not have to be particularly optimized. It is important that the generated assembly is correct, not that it is efficient.

The archive with the assignment data contains a number textN.c test codes. For each test code, a sample assembly output testN.s.hint is provided, which shows the assembly our reference implementation generates. You can use these files for inspiration, but do not need to generate the same assembly. Finally, a testN.exp file specifies the expected output of simulating your assembly using the SPIM simulator. The test.sh script can be used to run your pass and check your output against the expected output.

Notes

- This time we're working on SSA form (after the `mem2reg` pass runs), which means that there will be phi nodes. You will have to convert phi nodes into an appropriate sequence of moves when handling branches.
- Tests 1-5 cover basic operations, tests 6-7 branches, test 8 loads and test 9 checks an edge-case of phi elimination. The test cases are sorted roughly in order of hardness and you should focus on making tests 1-7 work before tackling tests 8 and 9.
- A MIPS cheatsheet listing relevant instructions is attached to this document. The pass.cpp file contains a number of hints for your implementation and also lists LLVM methods that may be useful when handling certain instruction types.
- The assignment uses a very simple register allocator, which assigns a new register for each instruction result. The assignment also ignores the calling convention. The test cases are chosen, so this does not become an issue. Of course, this is not how a realistic backend would work.
- We use SPIM in `-asm` configuration. This means pseudo-instructions can be used and there is no branch delay slot.

Submission

- Use the ISIS website
- Submit the project as a single file, extending the given pass.cpp file.
- File name has to be: `lastname1_lastname2.cpp`.
- First line of the `cpp` file should include first name, surname and student id, for each group participant, e.g. `/* Diego Maradona 10, Juergen Klinsmann 18 */`

MIPS cheatsheet

The MIPS architecture provides 32 integer registers, which are listed under their symbolic names in the following table. Relevant for us are the `$zero, $vN, $aN, $tN, $sN, $ra` registers, there should be no need to use any of the other registers. As we do not respect the calling convention, the distinction between caller-save and callee-save registers is not important to us.

Register	Use
`$zero`	Hardwired constant zero
`$at`	Reserved for use by pseudo-instructions
`$v0-1`	Function return value
`$a0-3`	Function arguments
`$t0-9`	Caller-save registers
`$s0-7`	Callee-save register
`$k0-1`	Reserved for kernel
`$gp`	Global area pointer
`$sp`	Stack pointer
`$fp`	Frame pointer
`$ra`	Return address
`HI, LO`	Hidden registers used by instructions with 64-bit result

The following table contains a list of all MIPS instructions, including pseudo-instructions, that we believe to be relevant for this assignment. Not all of these instructions have to be used, and it is permitted to use instructions outside this list, as long as they are supported by the SPIM simulator. In the table `ImmI`*n* refers to an *n*-bit signed immediate and `ImmU`*n* to an *n*-bit unused immediate.

Pseudo	Instruction	Operation
	`add Rd, Rs, Rt`	`Rd = Rs + Rt  (overflow exception)`
	`addi Rd, Rs, ImmI16`	`Rd = Rs + ImmI16  (overflow exception)`
	`addu Rd, Rs, Rt`	`Rd = Rs + Rt  (overflow silent)`
	`addiu Rd, Rs, ImmI16`	`Rd = Rs + ImmI16  (overflow silent)`
	`sub Rd, Rs, Rt`	`Rd = Rs - Rt  (overflow exception)`
	`subu Rd, Rs, Rt`	`Rd = Rs - Rt  (overflow silent)`
	`and Rd, Rs, Rt`	`Rd = Rs & Rt`
	`andi Rd, Rs, ImmU16`	`Rd = Rs & ImmU16`
	`or Rd, Rs, Rt`	`Rd = Rs | Rt`
	`ori Rd, Rs, ImmU16`	`Rd = Rs | ImmU16`
	`xor Rd, Rs, Rt`	`Rd = Rs ^ Rt`
	`xori Rd, Rs, ImmU16`	`Rd = Rs ^ ImmU16`
	`sll Rd, Rs, ImmU5`	`Rd = Rs << ImmU5`
	`sllv Rd, Rs, Rt`	`Rd = Rs << Rt`
	`sra Rd, Rs, ImmU5`	`Rd = Rs >> ImmU5  (arithmetic)`
	`srav Rd, Rs, Rt`	`Rd = Rs >> Rt  (arithmetic)`
	`srl Rd, Rs, ImmU5`	`Rd = Rs >> ImmU5  (logical)`
	`srlv Rd, Rs, Rt`	`Rd = Rs >> Rt  (logical)`
	`div Rd, Rs`	`LO = Rd / Rs, HI = Rd % Rs  (signed)`

	divu Rd, Rs	LO = Rd / Rs, HI = Rd % Rs (unsigned)
	mult Rd, Rs	HI.LO = Rd * Rs
	mfhi Rd	Rd = HI
	mflo Rd	Rd = LO
	j Label	goto Label
	jal Label	$ra = …; goto Label
	jr Rs	goto Rs
	beq Rd, Rs, Label	if (Rd == Rs) goto Label
	bne Rd, Rs, Label	if (Rd != Rs) goto Label
	bgez Rs, Label	if (Rs >= 0) goto Label (signed)
	bgtz Rs, Label	if (Rs > 0) goto Label (signed)
	blez Rs, Label	if (Rs <= 0) goto Label (signed)
	bltz Rs, Label	if (Rs < 0) goto Label (signed)
	slt Rd, Rs, Rt	Rd = Rs < Rt (signed)
	slti Rd, Rs, ImmI16	Rd = Rs < ImmI16 (signed)
	sltu Rd, Rs, Rt	Rd = Rs < Rt (unsigned)
	sltiu Rd, Rs, ImmU16	Rd = Rs < ImmU16 (unsigned)
Yes	bge Rd, Rs, Label	if (Rd >= Rs) goto Label (signed)
Yes	bgeu Rd, Rs, Label	if (Rd >= Rs) goto Label (unsigned)
Yes	bgt Rd, Rs, Label	if (Rd > Rs) goto Label (signed)
Yes	bgtu Rd, Rs, Label	if (Rd > Rs) goto Label (unsigned)
Yes	ble Rd, Rs, Label	if (Rd <= Rs) goto Label (signed)
Yes	bleu Rd, Rs, Label	if (Rd <= Rs) goto Label (unsigned)
Yes	blt Rd, Rs, Label	if (Rd < Rs) goto Label (signed)
Yes	bltu Rd, Rs, Label	if (Rd < Rs) goto Label (unsigned)
	lb Rd, ImmI16(Rs)	Rd = SExt(8-bit @ MEM[Rs+ImmI16])
	lbu Rd, ImmI16(Rs)	Rd = ZExt(8-bit @ MEM[Rs+ImmI16])
	lh Rd, ImmI16(Rs)	Rd = SExt(16-bit @ MEM[Rs+ImmI16])
	lhu Rd, ImmI16(Rs)	Rd = ZExt(16-bit @ MEM[Rs+ImmI16])
	lw Rd, ImmI16(Rs)	Rd = 32-bit @ MEM[Rs+ImmI16]
	lui Rd, ImmU16	Rd = ImmU16 << 16
Yes	li Rd, ImmI32	Rd = ImmI32
Yes	la Rd, Label	Rd = Label
Yes	move Rd, Rs	Rd = Rs
	syscall	Perform a syscall