Recitation 5

Assembly programing

March 10, 2014

Announcements

- PS5 is out, it consists of two parts:
    - Code, graded, individual, deadline 19.03
    - Theory, pass/fail, may work in groups, deadline 12.03
- We'll use different servers for the last two assignments, `gpuXX.idi.ntnu.no`, XX = 01,02,...,05 (details below).
- Grades will be out later today.

Typechecking

- We need to check that the number of arguments, and their types, matches the function declaration for function and method calls. This should be done in `typecheck_expression()`. This function should also return the return type of the function/method.
- And that the left and right hand side of assignments match, this should be done in `typecheck_assignment()`. Checking whether two types are equal should be done in `equal_types()`.
- You'll also need to add code to compute the type of class_field_access expressions.

Typechecking

- We could do a lot more typecheking, for instance catching the problem in err_callFunc.vsl
- There is a theory question about what additional typechecking we could do.
- You don’t have to implement this.

Why stop at assembly?

- The output of our compiler will be assembly, not machine code.
- Assembly is very close to machine code, the assembler simply replaces text with numbers.
- Which is very simple, but requires a lot of work to implement.
- So we'll just output assembly, and use an already existing assebler for the last step.

Assembly isn't one language

- Each processor/machine architecture has its own machine code, and hence its own assembly language. Some popular ones:
  - IA32 (aka x86)
  - x86_64 (aka AMD64, Intel 64)
  - ARM
  - Java bytecode, LLVM bytecode.
- We'll use ARM, because it is increasingly popular, especially smart phones, tablets etc.

ARM hardware

- x86(_64) is still dominant for PCs/servers though.
- So we don't have an actual ARM server for you to use, we'll use an emulator.
- If you want to run it on actual hardware, it is at least theoretically possible to do it on a smart phone, or on something like the Raspberry Pi.

Server

- The emulator is not installed on `login.stud.ntnu.no`, we'll use `gpuXX.idi.ntnu.no`, XX = 01-05, instead.
- If you want to run it on your own machine, you'll have to install the following packages (for Ubuntu 12.04):
    - gcc-4.6-arm-linux-gnueabihf
    - qemu-user

Assembly

Assembly example, a function adding two numbers

```
_add:                      # Label
    push {lr}              # Setting up stack frame
    push {fp}
    mov fp, sp
    ldr r1, [fp, #12]      # Loading variables
    ldr r0, [fp, #8]
    add r0, r1             # Adding
    mov sp, fp             # Removing stack frame
    pop {fp}
    pop {pc}
```

Assembly

- Basically a list of instructions, each instruction is something the CPU can do natively, like arithmetic, or loading/storing data to/from memory.
- Each instruction has 0 or more operands, which are registers or memory locations. E.g. in the code above, the registers `r0` and `r1` are the operands of the `add` instruction. The values in these registers should be added.
- By default, the instructions are executed one by one, in the order they appear.
- Lables can be placed in the code, and special jump/goto instructions can be used to jump to a label, instead of the next instruction. This can be used to implement if/else statements, loops, and functions.

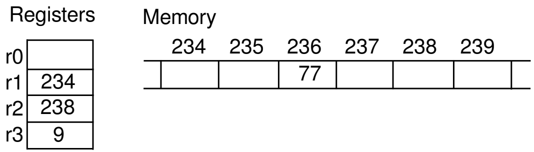
Registers

- ARM has 37 registers. However, some are special purpose, and not all are available in all processor modes.
- We will only use the following:
  - `r0-r7` General purpose registers
  - `pc/r15` The program counter
  - `fp/r11` The frame pointer
  - `sp/r13` The stack pointer
  - `lr/r14` The link register
- The last three are used to manage the *stack*, and for function calls.

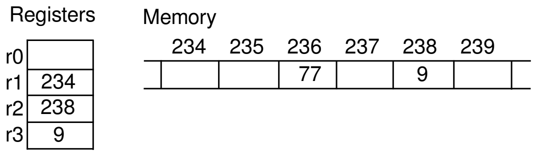
Data movement

- Data can be moved between registers (only) with the `MOV` instruction, and to and from memory with the `LDR` (load) and `STR` (store) instructions.
- `MOV r0, r3` moves/copies the data in r3 to r0 (r0 := r3)
- `LDR r0, [r3]` loads the value at the address stored in r3 to r0.
- `LDR r0, [r3,#4]` loads the value at address stored in r3 + 4 to r0.
- `STR r0, [r3]` stores r0 to the value at the address stored in r3.
- `STR r0, [r3,#4]` stores r0 to the value at address stored in r3 + 4.
- So for MOV and LDR, the first operand is the destination, for STR the first operand is the source.

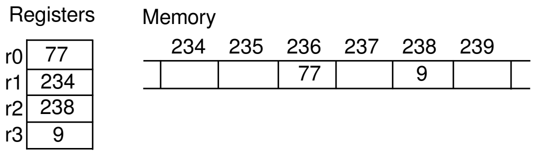
Example



```
STR r3, [r2]
```



```
LDR r0, [r1, #2]
```



The stack

- The stack is just a region in memory. In principle, we can store whatever we want there, and use it however we like.
- The instructions `PUSH` and `POP` can be used to access the stack. They're equivalent to a `STR`/`LDR` followed by updating the stack pointer.
- The stack grows downwards, towards lower addresses.
- Note that if we overwrite `sp` with something, the stack may no longer work as expected.
- `PUSH { r0 }` stores the contents of `r0` at the top of the stack.
- `POP { r0 }` removes the content from the top of the stack, and puts it in `r0`.

Flow of control, functions

- As mentioned, we’ll use lables to implement functions. We’ll put a label in front of the code of the function. Whenever we want to call the function, we jump to the label.
- Before we generate the code for the function, we must add the label.
- We can really pick anything (unique) for the label, but the function name is an obvious candidate.

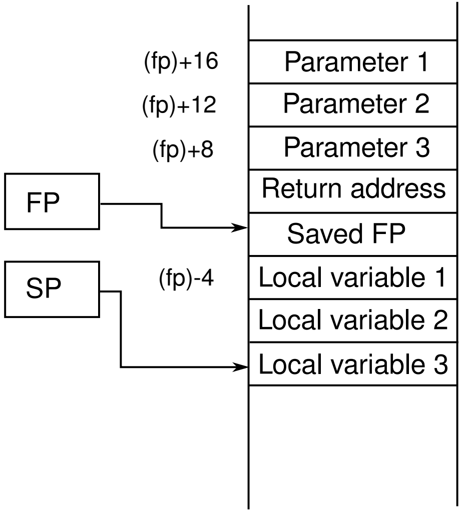
Calling conventions

- Just jumping to lables is not enough to implement functions, we must also be able to
  - Pass parameters
  - Return results
  - Have local variables (which doesn’t overwrite those of the caller).
  - Allow nested function calls, and recursion.
- We’ll use the stack to facilitate all of this.

Stack frames

- The space used by a function on the stack is called a frame (aka activation record).
- When a function is called, a new frame is created and pushed on top of the stack.
- When the function returns, the frame is popped.
- As mentioned `sp` points to the top of the stack. `fp` points to the start of the current frame.

Stack layout



Function call protocoll

- Caller saves registers on stack.
- Caller pushes parameters on stack.
- Caller saves return address in link register.
- Caller jumps to called function address.
- Callee saves link register on stack.
- Callee saves old `fp` on stack
- Callee sets new `fp` to top of stack.
- Callee executes function (overwriting registers if necessary.)
- Callee sets `sp` to its `fp`
- Callee sets restores old `fp`
- Callee stores result/return value in `r0`
- Callee jumps back to caller, and pops return address.
- Caller removes parameters, restores registers, uses result.

Inconsistencies

- The call convention used by Linux/GCC/ARM is somewhat different, the first arguments are not passed on the stack, but in registers, and the arguments are pushed in the oposite order.
- This doesn't matter to us as long as we only call our own functions, and they follow our convention.
- If, however, we want to call library functions, like `printf` or `malloc`, we'll have to follow that convention.
- The code for print statements have already been implemented, and `malloc` will not be needed in this assignment, so for now, you should allways follow our convention.

Functions

- The `bl` instruction will jump to a label, and store the address of the next instruction (e.g. return address) in `lr`.
- The first thing the callee should do is to push the return address, in `lr` on the stack.
- It should then set up a new stack frame, by pushing (the old) `fp`, and setting `fp` to the current `sp`.
- Before leaving the function, the stack frame must be removed, by setting `sp` to `fp` and setting `fp` to the old frame pointer.
- To return, you move the return address to `pc`, e.g `POP {pc}`, (if the return address is on the top of the stack.

Declarations

- VSL doesn’t make any guarantees about the values of uninitalized variables.
- Hence, for a declaration, we only need to make space on the stack, for instance by pushing some value.
- Declarations can be anywhere in the code though.
- So if you use the stack between two declarations, make sure you clean up, elsewise, the addresses of local variables will be wrong.

Accessing variables

- Variables in the current scope can be accessed by using their offset from the current `fp`.
- E.g. `LDR r0, [fp,#-4]`, will load what is stored with an offset of -4 from the frame pointer into r0.
- That’s what the offsets in the symbol table are for.

Expressions

- The only expressions we'll handle in this assignment are variables, constants, and function calls. (variables and constants have their own nodes, and are not handled by the `gen_EXPRESSION` function but are still kinds of expressions.
- The general strategy is to always place the value of the expression on the top of the stack.
- Then, nodes higher up in the tree will know where the vaue of the expression is.
- E.g. in an assignment statement, we would first generate code for the right hand side, which will place the vaule on the top of the stack. We can then simply move the value on top of the stack to the address of the left hand side.

Constants

The following instruction will load the constant 5 into `r0`

```
mov r0, #5
```

Since all ARM instructions are 32 bit long, this only works if the constant is less than 8 bits. For larger constants, we need to do it in two instructions:

```
movw r0, #:lower16:5
movt r0, #:upper16:5
```

The MOVE32 opcode in the framework will generate these two instructions for you automatically.

Constants

For string constans, you'll need to use .STRINGX, where X is the index in the string table:

```
movw r0, #:lower16:.STRING0
movt r0, #:upper16:.STRING0
```

For booleans, you should use 0 for false and 1 for true. Floating point numers should be on hexadecimal format, this is a bit ugly to do, so you don't have to do it.

Instruction cheat sheet

These are all the instructions you’ll need for this assignment

push, pop	Move to/from stack and registers. Modifies `sp`.
mov	Move data between registers.
ldr, str	Move data to/from memory
bl	Jump to label and store return address in `lr`.
	(Obviously ARM has more instructions, and if you really want to, you can use them, but these are the only ones you need)

Framework

- This will be yet another tree traversal, for each node the corresponding instruction(s) should be generated. This time using the `gen_X` functions in `generator.c`
- The instructions are stored in a linked list before they are printed to a file.
- During the tree traversal, you should therefore add the instructions to this list.

Linked list

```
static void instruction_add ( opcode_t op,
char *arg1,
char *arg2,
int32_t off1,
int32_t off2 )
```

- The opcode is the instruction, e.g. MOV or PUSH, it is a enum, defined at the start of the file. (Note that the names are not identical to the actual instructions, in particluar, `BL` is called `CALL`)
- The args are optional operands to the instruction, variables for the registers are included. Use NULL if no operands are needed.
- The offsets are optional offsets from the operands, when indirect addressing is used.

Example

So the code

```
instruction_add(PUSH, r0, NULL, 0, 0);
instruction_add(LOAD, r0, fp, 0, 4);
```

Will generate.

```
push {r0}
ldr r0, [fp,#4]
```

Example, ”instructions”

In addition to intructions, we’ll need to add labels and strings:

```
instruction_add(STRING, STRDUP("_func:"),
    NULL, 0, 0);
instruction_add(LABEL, STRDUP("func"),
    NULL, 0, 0);
```

Will generate.

```
_func:
_func:
```

(LABEL adds the underscore and colon)

Example, ”instructions”

As mentioned, the MOVE32 opcode will generate both move instructions for constants:

```
instruction_add(MOVE32, STRDUP("#5"), r0, 0,0)
```

Will generate.

```
movw r0, #:lower16:5
movt r0, #:upper16:5
```

(Note that the order of the arguments is wrong/oposite)

Going beyond the framework

- The framework is somewhat restrictive, it is not possible to generate all versions of all instructions. It should be enough for the most straightforeward solution of this assignment though.
- If there is some additional feature you want to use, the STRING opcode essentially makes it possible to add arbitrary instructions.

`_printf` VS `printf`

- To distinguish between our functions, and those in libc, the labels for our functions should (always) start with a underscore.
- When you use the `CALL` opcode, this underscore will be added automatically. But you need to add it manually when making the label.
- When you use the `SYSCALL` ”opcode”, no underscore will be added. It can be used to call libc functions like printf. (you will not need it in this assignment).

Macros

- `TEXT_HEAD` and `TEXT_TAIL` generate code which should be at the start and end of your output.
- It is for parsing command-line arguments, and calling exit at the end.

Sumary

- You'll need to implement:
- `gen_PROGRAM()`
  - Mostly done, you only need to insert a call to the frist function (not neccesarilly called main).
- `gen_FUNCTION()`
  - Make label.
  - Set up stack frame.
  - Generate code for body of function.
  - Remove stack frame, jump to return address.
- `gen_DECLARATION_STATEMENT()`
  - Make space on the stack for local variables.
- `gen_EXPRESSION()`
  - Currently, only function calls.
  - Push all the arguments.
  - Jump to the label.
  - Handle returned value.

Sumary cont.

- `gen_VARIABLE()`
  - Load from memory using stack offset form symbol table.
- `gen_CONSTANT()`
  - Move constant to a register, and then to the top of the stack.
- `gen_ASSIGNMENT_STATEMENT()`
  - Generate code for right hand side.
  - Store result in address on left hand side.
- `gen_RETURN()`
  - Generate code for right hand side expression.
  - Place resutl in `r0`.

Testing

- For a given VSL program, there are many correct assembly programs, so the tests will not check the output directly.
- It will however compile the sample VSL programs with your compiler, and check their output.
- Most of the test programs used so far are to complex for us to currently be able to generate assembly, so they have not been included. This indcludes all the programs with type errors.

Testing typechecking

- The following test programs (included in previous assingments) have type errors:
- addingBool.vsl
- addingFloat2.vsl
- addingFloat3.vsl
- err_addingText.vsl
- err_addingVoid.vsl
- err_callFunc2.vsl
- Other programs, e.g. err_callFunc.vsl have type errors which we will not catch.

Technicalities

Most compilers can output assembly, for GCC, use the -S flag:

```
gcc -S mycode.c
```

This will give you x86(_64) assembly. To get ARM assembly, you need to use the cross compiler:

```
arm-linux-gnueabihf-gcc-4.6 -S mycode.c
```

Technicalities

To compiler/assemble assembly, put the assembly in a file ending with .s, and use the compiler as normal (for x86/ARM respectively):

```
gcc myassembly.s
arm-linux-gnueabihf-gcc-4.6 myassembly.s
```

To run the ARM binary using the emulator:

```
qemu-arm ./a.out
```