CGmips

March 12, 2018

0.0.1 P0 Code Generator for MIPS

Emil Sekerinski, March 2017 The generated code is kept in memory and all code generation procedures continuously append to that code: procedure `genProgStart` initializes the generator, then `gen`-prefixed procedures are to be called for P0 constructs as in order in which they are recognized by a recursive descent parser, and finally procedure `genProgExit` returns the generated code in assembly language as a string in a format that can be read in by the SPIM simulator. The generation procedures are: - `genBool`, `genInt`, `genRec`, `genArray` - `genProgStart`, `genGlobalVars`, `genProgEntry`, `genProgExit` - `genProcStart`, `genFormalParams`, `genLocalVars`, `genProcEntry`, `genProcExit` - `genSelect`, `genIndex`, `genVar`, `genConst`, `genUnaryOp`, `genBinaryOp`, `genRelation` - `genAssign`, `genActualPara`, `genCall`, `genRead`, `genWrite`, `genWriteln` - `genSeq`, `genCond`, `genIfThen`, `genThen`, `genIfElse`, `genTarget`, `genWhile`

Errors in the code generator are reported by calling `mark` of the scanner. The data types of the symbol table are used to specify the P0 constructs for which code is to be generated.

```
In [ ]: import nbimporter
        from SC import TIMES, DIV, MOD, AND, PLUS, MINUS, OR, EQ, NE, LT, GT, LE, \
             GE, NOT, mark
        from ST import Var, Ref, Const, Type, Proc, StdProc, Int, Bool
```

Following variables determine the state of the code generator:

- `curlev` is the current level of nesting of P0 procedures
- `regs` is the set of available MIPS registers for expression evaluation
- `label` is a counter for generating new labels
- `asm` is a list of triples; each triple consists of three (possibly empty) strings:
- a label
- an instruction, possibly with operands
- a target (for branch and jump instructions)

Procedure `genProgStart()` initializes these variables. Registers `$t0` to `$t9` are used as general-purpose registers.

```
In [9]: GPregs = {'$t0', '$t1', '$t2', '$t3', '$t4', '$t5', '$t6', '$t7', '$t8'}


        def genProgStart():
            global asm, curlev, label, regs
            asm, curlev, label = [], 0, 0
            regs = set(GPregs) # make copy
            putInstr('.data')
```

Reserved registers are $0 for the constant 0, $fp for the frame pointer, $sp for the stack pointer, and $ra for the return address (dynamic link).

```
In [10]: R0 = '$0'; FP = '$fp'; SP = '$sp'; LNK = '$ra'

          def obtainReg():
              if len(regs) == 0: mark('out of registers'); return R0
              else: return regs.pop()

          def releaseReg(r):
              if r in GPregs: regs.add(r)
```

```
In [ ]: def putLab(lab, instr = ''):
            """Emit label lab with optional instruction; lab may be a single
            label or a list of labels"""
            if type(lab) == list:
                for l in lab[:-1]: asm.append((l, '', ''))
                asm.append((lab[-1], instr, ''))
            else: asm.append((lab, instr, ''))

        def putInstr(instr, target = ''):
            """Emit an instruction"""
            asm.append(('', instr, target))

        def putOp(op, a, b, c):
            """Emit instruction op with three operands, a, b, c; c can be register or immediate"
            putInstr(op + ' ' + a + ', ' + b + ', ' + str(c))

        def putBranchOp(op, a, b, c):
            putInstr(op + ' ' + a + ', ' + b, str(c))

        def putMemOp(op, a, b, c):
            """Emit load/store instruction at location or register b + offset c"""
            if b == R0: putInstr(op + ' ' + a + ', ' + str(c))
            else: putInstr(op + ' ' + a + ', ' + str(c) + '(' + b + ')')
```

Following procedures "generate code" for all P0 types by determining the size of objects and store in the size field. - Integers and booleans occupy 4 bytes - The size of a record is the sum of the sizes of its field; the offset of a field is the sum of the size of the preceding fields - The size of an array is its length times the size of the base type.

```
In [ ]: def genBool(b):
            b.size = 4; return b

        def genInt(i):
            i.size = 4; return i

        def genRec(r):
            # r is Record
            s = 0
            for f in r.fields:
                f.offset, s = s, s + f.tp.size
            r.size = s
            return r

        def genArray(a):
            # a is Array
            a.size = a.length * a.base.size
            return a
```

For each global variable, `genGlobalVars(sc, start)` generates the assembler *directive* `.space`, consisting of the identifier as the label and the size of the variable as the operand. The parameter `sc` contains the top scope with all declarations parsed so far; only variable declarations from index `start` on in the top scope are considered. As MIPS instructions are not allowed to be identifiers, all variables get _ as suffix to avoid a name clash.

```
In [ ]: def genGlobalVars(sc, start):
            for i in range(len(sc) - 1, start - 1, - 1):
                if type(sc[i]) == Var:
                    sc[i].reg, sc[i].adr = R0, sc[i].name + '_'
                    putLab(sc[i].adr, '.space ' + str(sc[i].tp.size))
```

Procedure `genProgEntry(ident)` takes the program's name as a parameter. Directives for marking the beginning of the main program are generated; the program's name is it not used.

```
In [ ]: def genProgEntry(ident):
            putInstr('.text')
            putInstr('.globl main')
            putInstr('.ent main')
            putLab('main')
```

Procedure `genProgExit(x)` takes parameter `x` with the result of previous `gen-` calls, generates code for exiting the program, directives for marking the end of the main program, and returns the complete assembly code.

```
In [11]: def assembly(l, i, t):
             """Convert label l, instruction i, target t to assembly format"""
             return (l + ':\t' if l else '\t') + i + (', ' + t if t else '')

         def genProgExit(x):
             putInstr('li $v0, 10')
             putInstr('syscall')
             putInstr('.end main')
             return '\n'.join(assembly(l, i, t) for (l, i, t) in asm)
```

Procedure `newLabel()` generates a new unique label on each call.

```
In [ ]: def newLabel():
            global label
            label += 1
            return 'L' + str(label)
```

The code generator *delays the generation of code* until it is clear that no better code can be generated. For this, the not-yet-generated result of an expressions and the location of a variable is stored in *items*. In addition to the symbol table types `Var`, `Ref`, `Const`, the generator uses two more item types: - `Reg(tp, reg)` for integers or boolean values stored in a register; the register can be `$0` for constants `0` and `false` - `Cond(cond, left, right)` for short-circuited Boolean expressions with two branch targets. The relation `cond` must be one of `'EQ'`, `'NE'`, `'LT'`, `'GT'`, `'LE'`, `'GE'`. The operands `left`, `right` are either registers or constants, but one has to be a register. The result of the comparison is represented by two branch targets, stored as fields, where the evaluation continues if the result of the comparison is true or false. The branch targets are lists of unique labels, with targets in each list denoting the same location. If `right` is `$0`, then `'EQ'` and `'NE'` for `cond` can be used for branching depending on whether `left` is `true` or `false`.

```
In [ ]: class Reg:
            def __init__(self, tp, reg):
                # tp is Bool or Int
                self.tp, self.reg = tp, reg

        class Cond:
            # labA, labB are lists of branch targets for when the result is true or false
            def __init__(self, cond, left, right):
                self.tp, self.cond, self.left, self.right = Bool, cond, left, right
                self.labA, self.labB = [newLabel()], [newLabel()]
```

Procedure `loadItemReg(x, r)` generates code for loading item `x` to register `r`, assuming `x` is `Var`, `Const`, or `Reg`. If a constant is too large to fit in 16 bits immediate addressing, an error message is generated.

```
In [12]: def testRange(x):
             if x.val >= 0x8000 or x.val < -0x8000: mark('value too large')

         def loadItemReg(x, r):
             if type(x) == Var:
                 putMemOp('lw', r, x.reg, x.adr); releaseReg(x.reg)
             elif type(x) == Const:
                 testRange(x); putOp('addi', r, R0, x.val)
             elif type(x) == Reg: # move to register r
                 putOp('add', r, x.reg, R0)
             else: assert False
```

Procedure `loadItem(x)` generates code for loading item `x`, which has to be `Var` or `Const`, into a new register and returns a `Reg` item; if `x` is `Const` and has value `0`, no code is generated and register `R0` is used instead. For procedure `loadBool(x)`, the type of item `x` has to be `Bool`; if `x` is not a constant, it is loaded into a register and a new `Cond` item is returned.

```
In [13]: def loadItem(x):
             if type(x) == Const and x.val == 0: r = R0 # use R0 for "0"
             else: r = obtainReg(); loadItemReg(x, r)
             return Reg(x.tp, r)

         def loadBool(x):
             if type(x) == Const and x.val == 0: r = R0 # use R0 for "false"
             else: r = obtainReg(); loadItemReg(x, r)
             return Cond(NE, r, R0)
```

Procedure `put(cd, x, y)` generates code for `x op y`, where `op` is an operation with mnemonic `cd`. Items `x`, `y` have to be `Var`, `Const`, `Reg`. An updated item `x` is returned.

```
In [ ]: def put(cd, x, y):
            if type(x) != Reg: x = loadItem(x)
            if x.reg in (R0, '$a0', '$a1', '$a2', '$a3'): # find new destination register
                r = x.reg; x.reg = obtainReg()
            else: r = x.reg # r is source, x.reg is destination
            if type(y) == Const:
                testRange(y); putOp(cd, x.reg, r, y.val)
            else:
                if type(y) != Reg: y = loadItem(y)
                putOp(cd, x.reg, r, y.reg); releaseReg(y.reg)
            return x
```

Procedures `genVar`, `genConst`, `genUnaryOp`, `genBinaryOp`, `genRelation`, `genSelect`, and `genIndex` generate code for expressions (e.g. right hand side of assignments) and for locations (e.g. left hand side of assignments).

Procedure `genVar(x)` allows `x` to refer to a global variable, local variable, or procedure parameter: the assumption is that `x.reg` (which can be `R0`) and `x.adr` (which can be 0) refer to the variable. References to variables on intermediate level is not supported. For global variables, the reference is kept symbolic, to be resolved later by the assembler. Item `x` is `Var` or `Ref`; if it is `Ref`, the reference is loaded into a new register. A new `Var` item with the location is returned.

```
In [ ]: """
        def genVar(x): # version not supporting parameters in registers
            if 0 < x.lev < curlev: mark('level!')
            y = Var(x.tp); y.lev = x.lev
            if type(x) == Ref: # reference is loaded into register
                y.reg, y.adr = obtainReg(), 0 # variable at M[y.reg]
                putMemOp('lw', y.reg, x.reg, x.adr)
            elif type(x) == Var:
                y.reg, y.adr = x.reg, x.adr
            else: assert False
            return y
        """
        def genVar(x): # version supporting parameters in registers
            if 0 < x.lev < curlev: mark('level!')
            if type(x) == Ref:
```

```
            y = Var(x.tp); y.lev = x.lev
            if x.reg in ('$a0', '$a1', '$a2', '$a3'): # reference already in register, use i
                y.reg, y.adr = x.reg, 0 # variable at M[y.reg]
            else: # reference is loaded into register
                y.reg, y.adr = obtainReg(), 0 # variable at M[y.reg]
                putMemOp('lw', y.reg, x.reg, x.adr)
        elif type(x) == Var:
            if x.reg in ('$a0', '$a1', '$a2', '$a3'): # value already in register, use it
                y = Reg(x.tp, x.reg) #; y.lev, x.adr = x.lev, x.adr
            else:
                y = Var(x.tp); y.lev, y.reg, y.adr = x.lev, x.reg, x.adr
        else: assert False
        return y
```

Procedure genConst(x) does not need to generate any code.

```
In [1]: def genConst(x):
            # x is Const
            return x
```

Procedure genUnaryOp(op, x) generates code for op x if op is MINUS, NOT and x is Int, Bool; if op is AND, OR, item x is the first operand. If it is not already a Cond item, it is made so which is loaded into a register. A branch instruction is generated for OR and a branch instruction with a negated condition for AND.

```
In [ ]: def negate(cd):
            return {EQ: NE, NE: EQ, LT: GE, LE: GT, GT: LE, GE: LT}[cd]

        def condOp(cd):
            return {EQ: 'beq', NE: 'bne', LT: 'blt', LE: 'ble', GT: 'bgt', GE: 'bge'}[cd]

        def genUnaryOp(op, x):
            if op == MINUS: # subtract from 0
                if type(x) == Var: x = loadItem(x)
                putOp('sub', x.reg, R0, x.reg)
            elif op == NOT: # switch condition and branch targets, no code
                if type(x) != Cond: x = loadBool(x)
                x.cond = negate(x.cond); x.labA, x.labB = x.labB, x.labA
            elif op == AND: # load first operand into register and branch
                if type(x) != Cond: x = loadBool(x)
                putBranchOp(condOp(negate(x.cond)), x.left, x.right, x.labA[0])
                releaseReg(x.left); releaseReg(x.right); putLab(x.labB)
            elif op == OR: # load first operand into register and branch
                if type(x) != Cond: x = loadBool(x)
                putBranchOp(condOp(x.cond), x.left, x.right, x.labB[0])
                releaseReg(x.left); releaseReg(x.right); putLab(x.labA)
            else: assert False
            return x
```

Procedure `genBinaryOp(op, x, y)` generates code for `x op y` if `op` is `PLUS`, `MINUS`, `TIMES`, `DIV`, `MOD`. If `op` is `AND`, `OR`, operand `y` is made a `Cond` item it if is not so already and the branch targets are merged.

```
In [ ]: def genBinaryOp(op, x, y):
            if op == PLUS: y = put('add', x, y)
            elif op == MINUS: y = put('sub', x, y)
            elif op == TIMES: y = put('mul', x, y)
            elif op == DIV: y = put('div', x, y)
            elif op == MOD: y = put('mod', x, y)
            elif op == AND: # load second operand into register
                if type(y) != Cond: y = loadBool(y)
                y.labA += x.labA # update branch targets
            elif op == OR: # load second operand into register
                if type(y) != Cond: y = loadBool(y)
                y.labB += x.labB # update branch targets
            else: assert False
            return y
```

Procedure `genRelation(op, x, y)` generates code for `x op y` if `op` is `EQ`, `NE`, `LT`, `LE`, `GT`, `GE`. Items `x` and `y` cannot be both constants. A new `Cond` item is returned.

```
In [ ]: def genRelation(op, x, y):
            if type(x) != Reg: x = loadItem(x)
            if type(y) != Reg: y = loadItem(y)
            return Cond(op, x.reg, y.reg)
```

Procedure `genSelect(x, f)` "generates code" for `x.f`, provided `f` is in `x.fields`. Only `x.adr` is updated, no code is generated. An updated item is returned.

```
In [ ]: def genSelect(x, f):
            x.tp, x.adr = f.tp, x.adr + f.offset if type(x.adr) == int else \
                                x.adr + '+' + str(f.offset)
            return x
```

Procedure `genIndex(x, y)` generates code for `x[y]`, assuming `x` is `Var` or `Ref`, `x.tp` is `Array`, and `y.tp` is `Int`. If `y` is `Const`, only `x.adr` is updated and no code is generated, otherwise code for array index calculation is generated.

```
In [ ]: def genIndex(x, y):
            if type(y) == Const:
                offset = (y.val - x.tp.lower) * x.tp.base.size
                x.adr = x.adr + (offset if type(x.adr) == int else '+' + str(offset))
            else:
                if type(y) != Reg: y = loadItem(y)
                putOp('sub', y.reg, y.reg, x.tp.lower)
                putOp('mul', y.reg, y.reg, x.tp.base.size)
                if x.reg != R0:
                    putOp('add', y.reg, x.reg, y.reg); releaseReg(x.reg)
                x.reg = y.reg
            x.tp = x.tp.base
            return x
```

Procedure genAssign(x, y) generates code for x := y, provided x is Var. Item x is loaded into a register if it is not already there; if x is Cond, then either 0 or 1 is loaded into a register.

```
In [ ]: def genAssign(x, y):
            if type(x) == Var:
                if type(y) == Cond:
                    putBranchOp(condOp(negate(y.cond)), y.left, y.right, y.labA[0])
                    releaseReg(y.left); releaseReg(y.right); r = obtainReg()
                    putLab(y.labB); putOp('addi', r, R0, 1) # load true
                    lab = newLabel()
                    putInstr('b', lab)
                    putLab(y.labA); putOp('addi', r, R0, 0) # load false
                    putLab(lab)
                elif type(y) != Reg: y = loadItem(y); r = y.reg
                else: r = y.reg
                putMemOp('sw', r, x.reg, x.adr); releaseReg(r)
            else: assert False
```

The procedure calling convention is as follows: - last parameter at 0($fp), 2nd last at 4($fp), ... - previous frame pointer at -4($fp) - return address at -8($fp) - 1st local at -12($fp), ...

The Stack pointer $sp points to last used location on the stack.

On procedure entry: - caller pushes 1st parameter at -4($sp), 2nd at -8($sp), ... - caller calls callee - callee saves $fp at $sp - parameter size - 4 - callee saves $ra at $sp - parameter size - 8 - callee sets $fp to $sp - parameter size - callee sets $sp to $fp - local var size - 8

On procedure exit: - callee sets $sp to $fp + parameter size - callee loads $ra from $fp - 8 - callee loads $fp from $fp - 4 - callee returns

For each local variable, genLocalVars(sc, start) updates the entry of the variable with the FP-relative address and returns their total size. The parameter sc contains the top scope with all local declarations parsed so far; only variable declarations from index start on in the top scope are considered.

```
In [ ]: def genLocalVars(sc, start):
            s = 0 # local block size
            for i in range(start, len(sc)):
                if type(sc[i]) == Var:
                    s = s + sc[i].tp.size
                    sc[i].reg, sc[i].adr = FP, - s - 8
            return s
```

Procedure genProcStart() generate the directive for starting instructions.

```
In [ ]: def genProcStart():
            global curlev
            curlev = curlev + 1
            putInstr('.text')
```

Procedure genFormalParams(sc) determines the FP-relative address of all parameters in the list sc of procedure parameters. Each parameter must be of type Int or Bool or must be a reference parameter.

```
In [1]: def genFormalParams(sc):
            n = len(sc) # parameter block length
            for i in range(n):
                if sc[i].tp in (Int, Bool) or type(sc[i]) == Ref:
                    sc[i].reg, sc[i].adr = FP, (n - i - 1) * 4
                else: mark('no structured value parameters')
            return n * 4
```

Procedures genProcEntry(ident, parsize, localsize) and genProcExit(x, parsize, localsize) generate the procedure prologue and epilogue.

```
In [ ]: def genProcEntry(ident, parsize, localsize):
            putInstr('.globl ' + ident)          # global declaration directive
            putInstr('.ent ' + ident)            # entry point directive
            putLab(ident)                        # procedure entry label
            putMemOp('sw', FP, SP, - parsize - 4)  # push frame pointer
            putMemOp('sw', LNK, SP, - parsize - 8) # push return address
            putOp('sub', FP, SP, parsize)        # set frame pointer
            putOp('sub', SP, FP, localsize + 8)  # set stack pointer

        def genProcExit(x, parsize, localsize):
            global curlev
            curlev = curlev - 1
            putOp('add', SP, FP, parsize) # restore stack pointer
            putMemOp('lw', LNK, FP, - 8)  # pop return address
            putMemOp('lw', FP, FP, - 4)   # pop frame pointer
            putInstr('jr $ra')            # return
```

Procedure genActualPara(ap, fp, n) assume that ap is an item with the actual parameter, fp is the entry for the formal parameter, and n is the parameter number. The parameters are pushed SP-relative on the stack. The formal parameter is either Var or Ref.

```
In [ ]: def genActualPara(ap, fp, n):
            if type(fp) == Ref:  #  reference parameter, assume p is Var
                if ap.adr != 0:  #  load address in register
                    r = obtainReg(); putMemOp('la', r, ap.reg, ap.adr)
                else: r = ap.reg  #  address already in register
                putMemOp('sw', r, SP, - 4 * (n + 1)); releaseReg(r)
            else:  #  value parameter
                if type(ap) != Cond:
                    if type(ap) != Reg: ap = loadItem(ap)
                    putMemOp('sw', ap.reg, SP, - 4 * (n + 1)); releaseReg(ap.reg)
                else: mark('unsupported parameter type')
```

Procedure genCall(pr, ap) assumes pr is Proc and ap is a list of actual parameters.

```
In [ ]: def genCall(pr, ap):
            putInstr('jal', pr.name)
```

Procedures `genRead(x)`, `genWrite(x)`, `genWriteln()` generate code for SPIM-defined "syscalls"; `genRead(x)` and assumes that `x` is `Var` and `genWrite(x)` assumes that `x` is `Ref`, `Var`, `Reg`.

```
In [2]: def genRead(x):
            putInstr('li $v0, 5'); putInstr('syscall')
            putMemOp('sw', '$v0', x.reg, x.adr)

        def genWrite(x):
            loadItemReg(x, '$a0'); putInstr('li $v0, 1'); putInstr('syscall')

        def genWriteln():
            putInstr('li $v0, 11'); putInstr("li $a0, '\\n'"); putInstr('syscall')
```

For control structures: - `genSeq(x, y)` generates `x ; y`, assuming `x`, `y` are statements - `genCond(x)` generates code for branching on `x`, assuming that `x` is of type `Bool` - `genIfThen(x, y)` generates code for `y` in `if x then y` - `genThen(x, y)` generates code for `y` in `if x then y else z` - `genIfElse(x, y, z)` generates code for `z` in `if x then y else z` - `genTarget()` generates and returns a target for backward branches - `genWhile(lab, x, y)` generates code for `y` in `while x do y`, assuming that target `lab` was generated before `x`.

```
In [ ]: def genSeq(x, y):
            pass

        def genCond(x):
            if type(x) != Cond: x = loadBool(x)
            putBranchOp(condOp(negate(x.cond)), x.left, x.right, x.labA[0])
            releaseReg(x.left); releaseReg(x.right); putLab(x.labB)
            return x

        def genIfThen(x, y):
            putLab(x.labA)

        def genThen(x, y):
            lab = newLabel()
            putInstr('b', lab)
            putLab(x.labA);
            return lab

        def genIfElse(x, y, z):
            putLab(y)

        def genTarget():
            lab = newLabel()
            putLab(lab)
            return lab
```

```
def genWhile(lab, x, y):
    putInstr('b', lab)
    putLab(x.labA);
```