Spamulet0 Balsa Example types.balsa page 1/2

```
(--
    Spamulet0 processor description
    for ACiD workshop Balsa demo

    'types.balsa'
    Types and constants
--)

type word is 32 bits
type sword is 32 signed bits

-- INSTRUCTION TYPES

-- condition: instruction conditions
type condition is enumeration
    EQ, NE, CS, HS=CS, CC, LO=CC,
    MI, PL, VS, VC,
    HI, LS, GE, LT,
    GT, LE, AL, NV
end

-- regNo: 16 registers R[0..15]
type regNo is 4 bits

-- instType: 'inst' field common values
type instType is enumeration
    dataOpReg,     -- Data op reg <= reg op reg
    dataOpImm,     --     "   reg <= reg op immediate
    loadStoreImm,  -- Load/Store    [reg op immediate]
    loadStoreReg,  --     "         [reg op reg]
    loadStoreMult, -- NOT IMPLEMENTED, your mission ...
    branch,
    coprocessor,   -- NOT IMPLEMENTED
    swi            -- NOT IMPLEMENTED
over 3 bits

-- inst: ALU op/LDR/STR instruction encoding
--         (dataOpReg, dataOpImm, loadStoreImm, loadStoreReg)
type inst is record
    rhs : 12 bits;        -- [11..0] right hand side specifier
    Rd : regNo;           -- [15..12] destination reg
    Rn : regNo;           -- [19..16] left hand side reg
    options : 5 bits;     -- [24..20] varies by operation
    instType : instType;  -- [27..25] instruction class
    condition : condition -- [31..28] condition, all instructions
                          --          have them
over word

-- opcode: dataOp opcode
type opcode is enumeration
    AND, EOR, SUB, RSB,
    ADD, ADC, SBC, RSC,
    TST, TEQ, CMP, CMN,
    ORR, MOV, BIC, MVN
end

-- INSTRUCTION options FIELDS

-- dataOpOptions: options field for dataOpImm, dataOpReg
type dataOpOptions is record
    S : bit;
    opcode : opcode
over 5 bits

-- loadStoreOptions: options flags for loadStoreImm, loadStoreReg
type loadStoreOptions is record
    L,      -- 1: load 0: store
    W,      -- 1: write back EA to reg Rn (do this if P=0 too)
            --    0: don't ...
    B,      -- 1: byte 0: word
    U,      -- 1: add offset reg/imm to Rn 0: subtract ...
    P : bit -- 1: pre (EA is Left op Right) 0: post (EA is Left)
over 5 bits

-- INSTRUCTION RIGHT-HAND SIDES

-- shiftType/shift: shift fields for right hand sides
type shiftType is enumeration
    LSL, LSR, ASR, ROR, RRX=ROR
end

type shift is record
    shiftType : shiftType;
    shiftDist : 5 bits -- shift distance
end

type immShiftRHS is record
    Rm : regNo;    -- [3..0] Right hand reg
    regNimm : bit; -- [4] 0: immediate shift
    shift : shift  -- [11..5] shift
over 12 bits

type regShiftRHS is record
    Rm : regNo;    -- [3..0] Right hand reg
    regNimm : bit; -- [4] 1: register shift
    shiftType : shiftType; -- [6..5]
    _7 : bit;      -- [7] empty
    Rs : regNo     -- [11..8] distance to shift by (in reg Rs)
over 12 bits

type immRHS is record
    immediate : 8 bits; -- [7..0] 8b unsigned immediate
    rotateDist : 4 bits -- [11..8] rotate right distance (/2)
over 12 bits

-- instBranch: branch encoding (branch)
type instBranch is record
    offset : 24 signed bits; -- [23..0] signed wordwise offset
                             --         from PC+8
    L : bit;                 -- [24] 0: Branch 1: Branch with Link
    instType : instType;     -- [27..25] branch
    condition : condition    -- [31..28] required condition
over word

-- ldmStmOptions: options flags for loadStoreMult
type ldmStmOptions is record
    L,      -- 1: load 0: store
    W,      -- 1: write back EA to reg Rn (do this if P=0 too)
            --    0: don't ...
    S,      -- Supervisor mode flag, IGNORE
    U,      -- 1: add offset reg/imm to Rn 0: subtract ...
    P : bit -- 1: pre (EA is Left op Right) 0: post (EA is Left)
over 5 bits

-- instLdmStm: load/store multiple
type instLdmStm is record
    regs : array 16 of bit; -- [15..0] register select bit vector
    Rn : regNo;             -- [19..16] base register
    options: ldmStmOptions; -- [24..20] options L,W,S,U,P
    instType : instType;    -- [27..25] branch
    condition : condition   -- [31..28] required condition
over word
```

```
-- Branch helper types/functions
type branchShift is record
    _ : 2 bits;
    offset : 30 signed bits
end

function BranchOffset (instruction : word) =
    (branchShift {0, ((instruction as instBranch).offset
        as 30 signed bits)} as word)

function InstructionConditionInvertBit (instruction : word) =
    ((instruction as inst).condition as array 4 of bit)[0]

-- INTERNAL TYPES

type conditionCode is record z, v, n, c : bit end

type rhsType is enumeration
    immShiftRHS, regShiftRHS, imm8RHS, imm12RHS
end

-- INTERFACE TYPES

type rNw is enumeration write, read end

type address is record
    a : word;
    rNw : rNw
end
```

```
(--
    Spamulet0 processor description
    for ACiD workshop Balsa demo

    'shift.balsa'
    Shifter for execute unit
--)

import [types]

procedure shift (
    input command : shift;
    input i : word;
    output o : word
) is
begin
    select command then
        local
            procedure shift_n (
                parameter distanceBit : cardinal;
                parameter distance : cardinal;
                input i : word;
                output o : word
            ) is
            local
                type pairLeft is record
                    lsw : distance bits;
                    msw : 32 - distance bits
                end
                function PackWordLeft (lsw : distance bits;
                    msw : 32 - distance bits) =
                    (pairLeft {lsw, msw} as word)

                type pairRight is record
                    lsw : 32 - distance bits;
                    msw : distance bits
                end
                function PackWordRight (lsw : 32 - distance bits;
                    msw : distance bits) =
                    (pairRight {lsw, msw} as word)

                channel c : word

                procedure shift_body (
                    output o : word
                ) is
                begin
                    select i then
                        local
                            function i_lswLeft = (i as pairRight).lsw
                            function i_mswLeft = (i as pairRight).msw
                            function i_lswRight = (i as pairLeft).lsw
                            function i_mswRight = (i as pairLeft).msw

                            shared lsr is begin
                                o <- PackWordRight (i_mswRight (), 0)
                            end
                            shared asr is begin
                                o <- PackWordRight (i_mswRight (),
                                    ( -1 as distance bits))
                            end
                            shared lsl is begin
                                o <- PackWordLeft (0, i_lswLeft ())
                            end
                            shared ror is begin
                                o <- PackWordRight (i_mswRight (),
                                    i_lswRight ())
                            end
                        begin
                            if #(command.shiftDist)[distanceBit] then
                                case command.shiftType of
                                  LSR then lsr ()
                                | LSL then lsl ()
                                | ASR then if #i[31] then asr ()
                                    else lsr () end
                                | ROR then ror ()
                                end
                            else o <- i
                            end
                        end
                    end
                end
            begin
                if distance > 1 then
                    shift_n (distanceBit - 1, distance / 2, c, o) ||
                    shift_body (c)
                else
                    shift_body (o)
                end
            end
        begin
            shift_n (4, 16, i, o)
        end
    end
end
```

```
(--
    Spamulet0 processor description
    for ACiD workshop Balsa demo

    'alu.balsa'
    Arithmetic and Logic unit
--)

import [types]

procedure alu (
    input opcode : opcode;
    input l, r : word;
    input ci : bit;
    output o : word;
    output cco : conditionCode
) is
begin
    select opcode, l, r, ci then
        local
            variable ci_v : bit
            variable l_v, r_v : word
            variable cc : conditionCode
            variable result : word

            type addArg is record
                ci : bit;
                arg : word
            over 33 bits

            type addResult is record
                _ : bit;
                result : word;
                co : bit
            over 34 bits

            shared Add is
            local channel addResult : addResult
            begin
                addResult <- ((addArg {ci_v, l_v} as 33 bits)
                    + (addArg {ci_v, r_v} as 33 bits) as addResult)
                || select addResult then
                    result := addResult.result ;
                    cc.c := addResult.co ;
                    cc.v := #(addResult.result)[31] /=
                        (#r_v[31] /= #l_v[31]) /= addResult.co
                end
            end
        begin
            cc.c := 0 || cc.v := 0 ||
            case opcode of
              AND, EOR, SUB, ADD, ADC, SBC, TST, TEQ,
              CMP, CMN, ORR, MOV, BIC then
                  l_v := l
            else l_v := not l
            end ||
            case opcode of
              AND, EOR, RSB, ADD, ADC, RSC, TST, TEQ,
              ORR, MOV then
                  r_v := r
            else r_v := not r
            end ;
            case opcode of
              AND, BIC then result := l_v and r_v
            | EOR then result := l_v xor r_v
            | ORR then result := l_v or r_v
            | SUB, RSB, CMP then ci_v := 1; Add ()
            | ADD, CMN then ci_v := 0; Add ()
            | ADC, SBC, RSC then ci_v := ci; Add ()
            | TST then continue -- FIXME
            | TEQ then continue -- FIXME
            | MOV, MVN then result := r_v
            end ;
            cc.z := result = 0 || cc.n := #result[31] ;
            o <- result ||
            cco <- cc
        end
    end
end
```

```
(--
    Spamulet0 processor description
    for ACiD workshop Balsa demo

    'spamulet.balsa'
    Top level, i.e. most of the processor
--)

import [types]
import [shift]
import [alu]

procedure spamulet (
    output a : address;
    output do : word;
    input di : word
) is
    -- 'REAL' PROCESSOR STATE
    variable R : array 16 of word -- register bank for user mode
    variable ir : word            -- instruction register
    variable cc : conditionCode   -- flags n, c, z, v

    -- TEMPORARY USE VARIABLES
    variable pc_inc : word        -- incremented PC (R[15])
    variable foundPCPlus8 : bit   -- unavoidable detail to work around
                                  -- pipeline latency
    variable take : bit           -- take this instruction

    -- REGISTER I/O VARIABLES
    variable regNo : regNo        -- register no. for reads and writes
    variable regRead : word       -- output of register bank
    variable regWrite : word      -- input of register bank

    -- EXECUTE I/O VARIABLES
    variable aluResult, aluLeft, aluRight : word
    variable shiftIn : word
    variable shift : shift
    variable aluOpcode : opcode
    variable setFlags : bit

    -- RIGHT-HAND SIDE
    variable rhsType : rhsType

    -- DATA MEMORY ACCESSES
    variable mar : word

    -- INITIALISATION
    shared InitialiseCC is begin cc := {0,0,0,0} end

    shared Initialise is begin
        R[15] := 0 || InitialiseCC ()
    end

    -- PROGRAM COUNTER MANIPULATION
    shared WriteBackPC is begin R[15] := pc_inc end
    shared IncrementPC is begin pc_inc := (R[15] + 4 as word) end
    shared ReadPCPlus8 is begin
        if not foundPCPlus8 then
            IncrementPC (); foundPCPlus8 := 1
        end
    end

    -- INSTRUCTION FETCH
    shared FetchInstruction is begin
        a <- {R[15], read} ||
        di -> ir
    end

    -- REGISTER BANK
    shared RegisterRead is begin
        case regNo of
            -- Reading PC requires more care
            15 then ReadPCPlus8 (); regRead := pc_inc
        else regRead := R[regNo]
        end
    end

    shared RegisterWrite is begin
        R[regNo] := regWrite
    end

    -- CONDITION CODE CHECK
    shared CheckInstructionCondition is
    local channel take_c : bit
    begin
        case (ir as inst).condition of
          EQ, NE then take_c <- cc.z
        | CS, CC then take_c <- cc.c
        | MI, PL then take_c <- cc.n
        | VS, VC then take_c <- cc.v
        | HI, LS  then take_c <- cc.c or (not cc.z)
        | GE, LT then take_c <- cc.n = cc.v
        | GT, LE then take_c <- (cc.n = cc.v) or (not cc.z)
        | AL, NV then take_c <- 1
        end ||
        select take_c then
            take := take_c /= InstructionConditionInvertBit (ir)
        end
    end

    -- EXECUTION
    shared DoShift is
    local
        channel i, o : word
        channel command : shift
    begin
        shift (command, i, o) ||
        o -> aluRight ||
        i <- shiftIn ||
        command <- shift
    end

    shared DoALU is
    local
        channel l, r, o : word
        channel ci : bit
        channel opcode : opcode
        channel cco : conditionCode
        variable cc_new : conditionCode
    begin
        alu (opcode, l, r, ci, o, cco) ||
        opcode <- aluOpcode ||
        l <- aluLeft ||
        r <- aluRight ||
        ci <- cc.c ||
        o -> aluResult ||
        select cco then
            if setFlags then cc_new := cco end
        end ;
        if setFlags then cc := cc_new end
    end
```

```
    -- RIGHT-HAND SIDE
    shared FetchRn is begin
        regNo := (ir as inst).Rn;
        RegisterRead ();
        aluLeft := regRead
    end

    shared FetchRHS is begin
        case rhsType of
          imm8RHS then shiftIn :=
            (((ir as inst).rhs as immRHS).immediate as word)
        | imm12RHS then shiftIn := ((ir as inst).rhs as word)
        else
            regNo := ((ir as inst).rhs as immShiftRHS).Rm;
              RegisterRead ();
              shiftIn := regRead
        end ;
        case rhsType of
          imm8RHS then
            local type rotate is record _ : bit; rotate : 4 bits end
            begin
                shift := {ROR, (rotate {0, ((ir as inst).rhs as
                    immRHS).rotateDist} as 5 bits)};
                DoShift ()
            end
        | imm12RHS then aluRight := shiftIn
        | immShiftRHS then
            shift := ((ir as inst).rhs as immShiftRHS).shift;
            DoShift ()
        | regShiftRHS then
            regNo := ((ir as inst).rhs as regShiftRHS).Rm;
            RegisterRead ();
            shift := {((ir as inst).rhs as regShiftRHS).shiftType,
                (#regRead[0..4] as 5 bits)};
            DoShift ()
        end
    end

    -- DATA MEMORY ACCESSES
    shared MemoryRead is begin a <- {mar, read} || di -> regWrite end
    shared MemoryWrite is begin a <- {mar, write} || do <- regRead end

    -- DECODE AND EXECUTE
    shared DecodeInstruction is
    local
        shared RHS_ALU_writeBack is begin
              FetchRHS () ||
            aluOpcode := ((ir as inst).options as
                dataOpOptions).opcode ||
            setFlags := ((ir as inst).options as dataOpOptions).S;
            regNo := (ir as inst).Rd ||
              [ DoALU (); regWrite := aluResult ];
            RegisterWrite ()
        end

        shared LoadStore is
            shared WriteBackEA is begin
                regNo := (ir as inst).Rn ||
                regWrite := aluResult;
                RegisterWrite ()
            end
        begin
            FetchRHS () ||
            if ((ir as inst).options as loadStoreOptions).U then
                -- add/sub
                aluOpcode := ADD
            else
                aluOpcode := SUB
            end ||
            setFlags := 0;
            regNo := (ir as inst).Rd ||
            if ((ir as inst).options as loadStoreOptions).P then
                -- pre
                DoALU ();
                mar := aluResult
            else
                mar := aluLeft
            end;
            if ((ir as inst).options as loadStoreOptions).L then
                -- load
                MemoryRead ();
                if ((ir as inst).options as loadStoreOptions).B then
                    -- byte
                    regWrite := ((regWrite as array 4 of byte)
                        [(#mar[0..1] as 2 bits)] as word)
                end;
                RegisterWrite ()
            else -- store
                RegisterRead ();
                MemoryWrite ()
            end;
            if not ((ir as inst).options as loadStoreOptions).P then
                -- post
                DoALU (); WriteBackEA ()
                -- FIXME, writeback
            else if ((ir as inst).options as loadStoreOptions).W then
                -- pre & W
                WriteBackEA ()
            end end
        end
    begin
        case (ir as inst).instType of
          dataOpReg then
            FetchRn () ||
            if ((ir as inst).rhs as immShiftRHS).regNimm
            then rhsType := regShiftRHS
            else rhsType := immShiftRHS
            end ;
            RHS_ALU_writeBack ()
        | dataOpImm then
            FetchRn () ||
            rhsType := imm8RHS;
            RHS_ALU_writeBack ()
        | loadStoreImm then
            FetchRn () ||
            rhsType := imm12RHS;
            LoadStore ()
        | loadStoreReg then
            FetchRn () ||
            rhsType := immShiftRHS;
            LoadStore ()
        | loadStoreMult then
            continue -- INSERT CODE HERE
                     -- (after removing the 'continue')
        | branch then
            if (ir as instBranch).L then -- BL
                R[14] := R[15]
            end;
            ReadPCPlus8 ();
            aluLeft := pc_inc ||
            aluRight := BranchOffset (ir) ||
            aluOpcode := ADD ||
```

```
            setFlags := 0 ;
            DoALU ();
            R[15] := aluResult
        | coprocessor then continue
        | swi then continue
        end
    end
begin
    Initialise ();
    loop
        FetchInstruction () || foundPCPlus8 := 0;
        IncrementPC ();
        WriteBackPC () || CheckInstructionCondition ();
        if take then
            DecodeInstruction ()
        end
    end
end
```