CPU设计报告

吕文龙
曲狄

设计总结

- 使用Haskell作为硬件描述语言，设计实现了一个简单的CPU，支持Load、Store、基本算数操作、条件与无条件跳转
- Haskell代码被编译为可综合的verilog
- 汇编语言目前直接嵌入在Haskell中，作为一门EDSL(Embeded Domain Specific Language)使用
- 无cache，无pipeline。
- Processor通过JTAG UART与host PC通信。

实例——嵌入在Haskell中的汇编代码

```haskell
prog :: IRom
prog =  Arith Id iReg 0 r7
     :> Jump CR 2
     :> Jump UR (-2)
     :> Load (RImm 2)  r8  -- r8 := 2
     :> Arith Add pcreg r8 jmpreg  -- jmpreg := pcreg + 2
     :> Jump UA facIterAddr    -- call facIter
     :> Arith Id r8 0 oReg
     :> Jump UA 0 -- infinite loop
     :> EndProg
     :> Load  (RImm 1) r8      -- r8 = 1
     :> Arith Eq       r7 zeroreg r9
     :> Jump  CA       facIterRet
     :> Arith Mul      r8 r7 r8
     :> Arith Decr     r7 r7 r7
     :> Jump  UR       (-4)
     :> Jump  Back     0 -- return
     :> (repeat EndProg)
     where facIterAddr = 9
           facIterRet  = 15
```

一段从PIO读取数据，并
计算阶乘(factorial)的代码

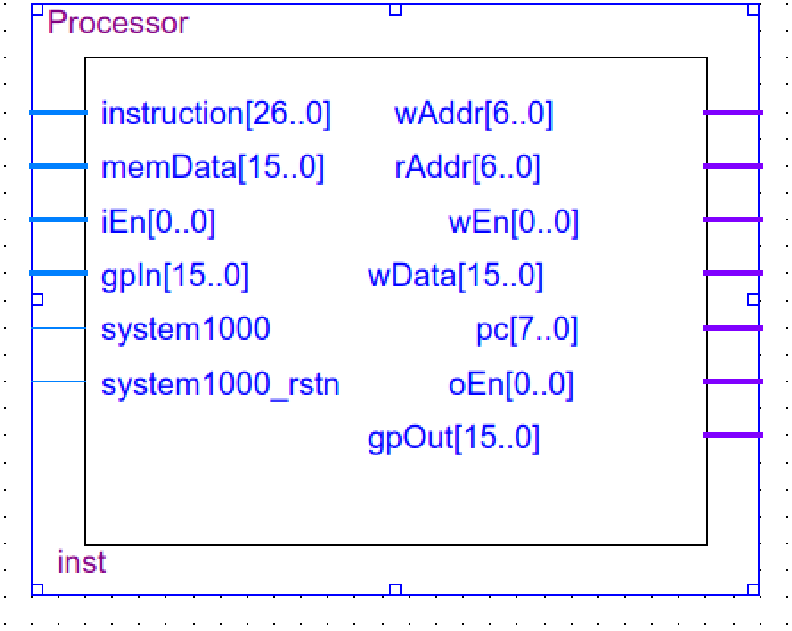
在DE2上，可以将16bit PI
接到switch上，16bit PO接到
LEDR上，进行测试

Demo——Hello World

```
:> Load (RImm 0b0000000011111111) r9 -- mask,  only the least 8 bit of data is valid
:> Arith And r7 r9 r7
:> Load (RImm 0b0000000000000000) r9 -- mask,  set address, write data
:> Arith Or r7 r9 oReg
:> Load (RImm 0b1111111111111111) r9 -- set address = 1, write control
:> Arith Id r9 0  oReg
:> Jump Back 0
:> (repeat EndProg)
```

```
helloWorld :: IRom
helloWorld =  Load (RImm 2) r8
           :> Load (RImm (trans 'H')) r7
           :> Arith Add pcreg r8 jmpreg
           :> Jump UA putCharLabel
           :> Load (RImm (trans 'e')) r7
           :> Arith Add pcreg r8 jmpreg
           :> Jump UA putCharLabel
           :> Load (RImm (trans 'l')) r7
           :> Arith Add pcreg r8 jmpreg
           :> Jump UA putCharLabel
           :> Load (RImm (trans 'l')) r7
           :> Arith Add pcreg r8 jmpreg
           :> Jump UA putCharLabel
           :> Load (RImm (trans 'o')) r7
           :> Arith Add pcreg r8 jmpreg
           :> Jump UA putCharLabel
           :> Load (RImm (trans ' ')) r7
```

Processor接口



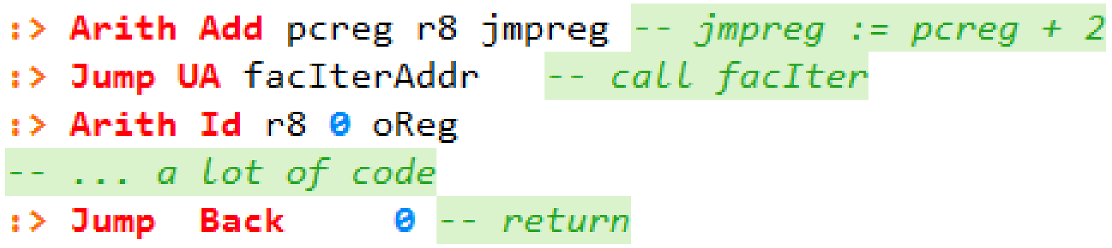
- Input
  - 16 bit General Purpose input + 输入使能
  - 指令输入
  - data read from memory

- Output
  - 16 bit General Purpose output + 输出使能
  - PC
  - wAddr, rAddr, wEn, wData

寄存器模型

- Word: 16bit
- 32个 Register, 其中 0~5号register约定有特殊用途
  - register 0: zeroreg，存储常数0，只读
  - register 1: jmpreg，存储函数返回地址
  - register 2: pcreg，存储当前PC，常用于计算函数返回地址，以存入jmpreg
  - register 3: iReg，存储PI的输入
  - register 4: oReg，存储PO的输出
  - register 5: iEn，存储PI的输入使能

pcreg与jmpreg用于函数调用



```
:> Arith Add pcreg r8 jmpreg -- jmpreg := pcreg + 2
:> Jump UA facIterAddr     -- call facIter
:> Arith Id r8 0 oReg
-- ... a lot of code
:> Jump  Back      0 -- return
```

指令集与处理器架构

- 基本算数指令
  - 算数运算
  - 逻辑运算
  - 条件判断
- 跳转指令
  - 条件与无条件跳转
  - 相对与绝对地址
  - 函数返回跳转
- Load 指令
  - 加载立即数(Alu运算不支持立即数操作)
  - 访存
- Store 指令
  - 存立即数
  - 存寄存器数据
- 堆栈操作指令
  - Push
  - Pop

```
data Instruction = Arith OpCode     RegIdx RegIdx RegIdx
                 | Jump  JmpCode    PC
                 | Load  LoadFrom   RegIdx
                 | Store StoreFrom DAddr
                 | Push  RegIdx
                 | Pop   RegIdx
                 | EndProg
                 deriving(Eq, Show)
```

算数运算指令

```haskell
data Instruction =
                 Arith OpCode    RegIdx RegIdx RegIdx
                 -- | Jump  JmpCode   PC
                 -- | Load  LoadFrom  RegIdx
                 -- | Store StoreFrom DAddr
                 -- | Push  RegIdx
                 -- | Pop   RegIdx
                 -- | EndProg
                 -- deriving(Eq, Show)

data OpCode     = Nop    | Id  | Incr | Decr | Neg | Not
                | Add    | Sub | Mul  | Div  | Mod
                | Eq     | Ne  | Lt   | Gt   | Le  | Ge
                | And    | Or  | Xor deriving(Eq, Show)
```

ALU

```
alu :: OpCode           -- operator
    -> (Word, Word)     -- two operands
    -> (Word, Bool)     -- result and Conditional test resutl
alu op (x, y) =  (opRet, cnd)
    where (opRet, cnd) = (app op x y, testBit opRet 0)
          app op x y   = case op of
            Nop   -> 0
            Id    -> x
            Incr  -> x + 1
            Decr  -> x - 1
            Neg   -> negate x
            Not   -> complement x
            Add   -> x + y
            Sub   -> x - y
            Mul   -> x * y
            Div   -> x `quot` y
            Mod   -> x `rem` y
            Eq    -> if x == y then 1 else 0
            Ne    -> if x /= y then 1 else 0
            Gt    -> if x > y  then 1 else 0
            Lt    -> if x < y  then 1 else 0
            Le    -> if x <= y then 1 else 0
            Ge    -> if x >= y then 1 else 0
            And   -> x .&. y
            Or    -> x .|. y
            Xor   -> x `xor` y
```

Id 指令可用用来实现
mov操作。
mov r1 r3可以用
Arith Id r1 r2 r3
来实现

跳转指令

```
data Instruction =
                 -- Arith OpCode    RegIdx RegIdx RegIdx
                 | Jump  JmpCode   PC
                 -- | Load  LoadFrom  RegIdx
                 -- | Store StoreFrom DAddr
                 -- | Push  RegIdx
                 -- | Pop   RegIdx
                 -- | EndProg
                 -- deriving(Eq, Show)

data JmpCode    = NoJmp -- no jump
                | UA    -- unconditional absolute jump
                | UR    -- unconditional relative jump
                | CA    -- conditional absolute jump
                | CR    -- conditional relative jump
                | Back  -- read jmpreg and jump back
                deriving(Eq, Show)
```

CPU的跳转模块

- 输入：
  - JumpCode，以及ALU中的逻辑运算结果
  - 当前PC，要跳转的PC(绝对地址或者偏移量)，jmpreg中的值
- 输出：
  - 下一时钟周期的PC

```haskell
updatePC :: (JmpCode, Bool) -- (jump code, cnd)
         -> (PC, PC, Word)  -- (current-pc, jump-addr, jmpreg)
         -> PC
updatePC (jmpCode, cnd) (pc, jmpNum, jumpRegV) = case jmpCode of
    NoJmp -> pc + 1
    UA    -> jmpNum
    UR    -> pc + jmpNum
    CA    -> if cnd then jmpNum else pc + 1
    CR    -> if cnd then pc + jmpNum else pc + 1
    Back  -> fromIntegral jumpRegV
```

Load指令

```
data Instruction =
                  -- Arith OpCode    RegIdx RegIdx RegIdx
                  -- | Jump  JmpCode   PC
                  | Load  LoadFrom  RegIdx
                  -- | Store StoreFrom DAddr
                  -- | Push  RegIdx
                  -- | Pop   RegIdx
                  -- | EndProg
                  -- deriving(Eq, Show)

data LoadFrom   = RAddr DAddr
                | RImm Word deriving(Eq, Show)

data LdCode     = NoLoad   -- 不在指令集中
                | LdImm    -- 是译码所得
                | LdAddr
                | LdAlu  deriving(Eq, Show)
```

CPU的Load模块

- 输入：
  - LdCode，
  - 寄存器号，
  - 译码所得的立即数以及ALU运算结果
- 功能：根据LdCode更新寄存器堆
- 从Memory加载数据要下一周期才到，因此不在load模块处理

```
load :: LdCode
     -> RegIdx
     -> (Word, Word)  -- (immediate-number, aluOut)
     -> Reg
     -> Reg
load ldCode toReg (imm, aluOut) regs = regs <~ (toReg, v)
    where v = case ldCode of
                NoLoad -> 0
                LdImm  -> imm
                LdAlu  -> aluOut
                LdAddr -> regs !! toReg -- memory-load is delayed
```

Store指令

- 向内存写数据，可用写立即数，也可以写寄存器中的值

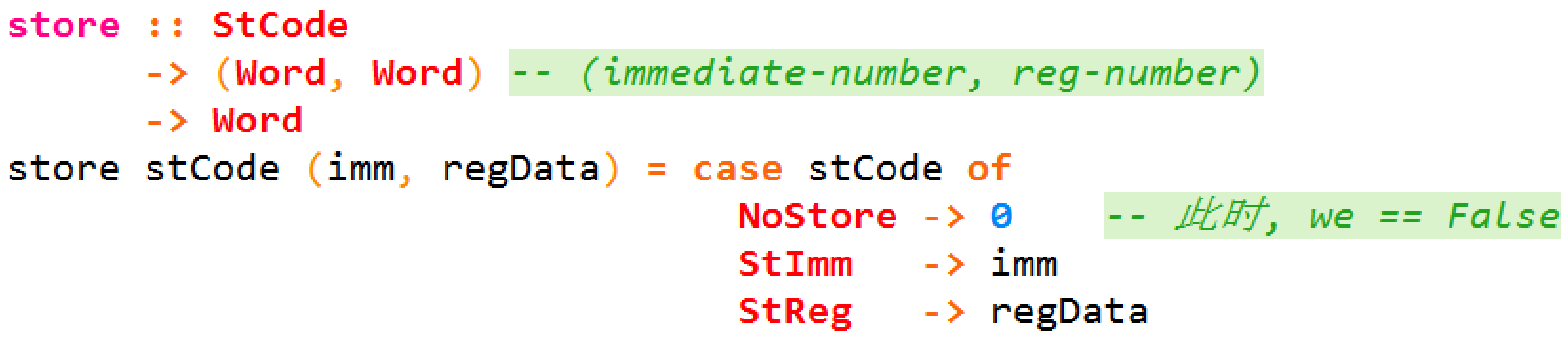
```
data Instruction =
                   -- Arith OpCode     RegIdx RegIdx RegIdx
                   -- | Jump   JmpCode    PC
                   -- | Load   LoadFrom   RegIdx
                   | Store StoreFrom DAddr
                   -- | Push   RegIdx
                   -- | Pop    RegIdx
                   -- | EndProg
                   -- deriving(Eq, Show)

data StoreFrom = MReg RegIdx
               | MImm Word deriving(Eq, Show)

data StCode    = NoStore
               | StImm
               | StReg            deriving(Eq, Show)
```

CPU中的Store模块

- 功能：根据StCode，给出要写入内存的数据



```haskell
store :: StCode
      -> (Word, Word) -- (immediate-number, reg-number)
      -> Word
store stCode (imm, regData) = case stCode of
                                NoStore -> 0     -- 此时, we == False
                                StImm   -> imm
                                StReg   -> regData
```

堆栈操作指令

- 内部状态有一个对程序员不可见的寄存器sp，指向栈顶，初始值为20，Push时sp加一，Pop时sp减一
- Push regN: 把 regN寄存器中的数据压栈
- Pop regN: 把栈顶的数据pop出并存入regN寄存器
- Push/Pop其实是特殊的Store/Load
- 可用于递归函数

```
data Instruction =
                    -- Arith OpCode     RegIdx RegIdx RegIdx
                    -- | Jump   JmpCode    PC
                    -- | Load   LoadFrom   RegIdx
                    -- | Store  StoreFrom  DAddr
                    | Push   RegIdx
                    | Pop    RegIdx
                    -- | EndProg
                    -- deriving(Eq, Show)
```

```
data SpCode     = None
                | Up
                | Down
                deriving(Eq, Show)
```

```
updateSp :: SpCode -> DAddr -> DAddr
updateSp None sp = sp
updateSp Up   sp = sp + 1
updateSp Down sp = sp - 1
```

PIO

- 可以像普通寄存器那样操作IO
- 每个时钟周期，processor的输入使能信号会存入iEn寄存器，通用输入信号会存入iReg寄存器
- 向oReg寄存器写数据，会把数据输出到通用输出，并使输出使能置一。

机器码

```haskell
data MachCode = MachCode {
    ldCode      :: LdCode
  , stCode      :: StCode
  , opCode      :: OpCode
  , jmpCode     :: JmpCode
  , spCode      :: SpCode
  , ldImm       :: Word
  , stImm       :: Word
  , fromReg0 :: RegIdx -- oprand 0
  , fromReg1 :: RegIdx -- oprand 1
  , toReg       :: RegIdx -- write back register
  , toAddr      :: DAddr  -- write address
  , fromAddr :: DAddr  -- read address
  , we          :: Bool
  , jmpNum      :: PC
  } deriving(Eq, Show)
```

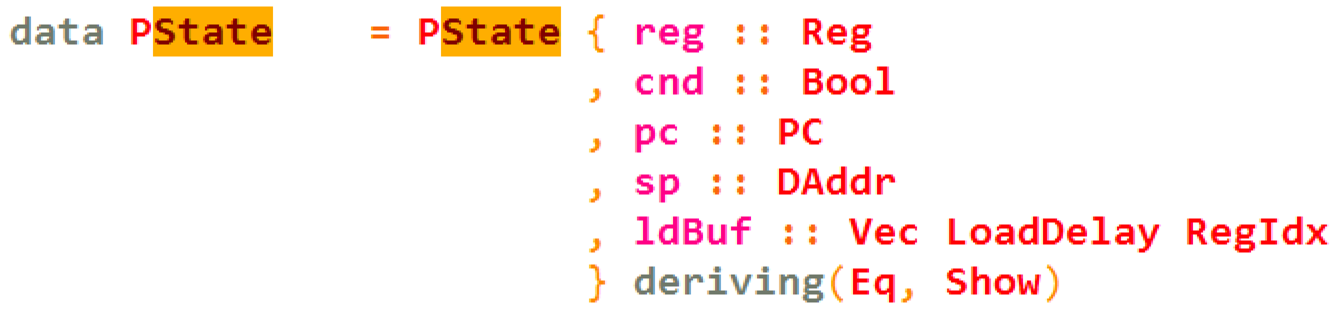
译码器

- 将Instruction翻译成MachCode

```haskell
decode :: DAddr          -- ^ stack pointer
       -> Instruction    -- ^ current instruction
       -> MachCode       -- ^ target machine code
decode sp instr = case instr of
    Arith op r0 r1 r2  -> def {ldCode = LdAlu, opCode = op, fromReg0 = r0, fromReg1 = r1, toReg = r2}
    Jump  jType jAddr  -> def {jmpCode = jType, jmpNum = jAddr}
    Load (RImm n) rid  -> def {ldCode = LdImm, ldImm = n, toReg = rid}
    Load (RAddr a) rid -> def {ldCode = LdAddr, fromAddr = a, toReg = rid}
    Store (MImm n) a   -> def {stCode = StImm, stImm = n, toAddr = a, we = True}
    Store (MReg r) a   -> def {stCode = StReg, fromReg0 = r, toAddr = a, we = True}
    Push r             -> def {stCode = StReg, fromReg0 = r, toAddr = sp + 1, spCode = Up, we = True}
    Pop r              -> def {ldCode = LdAddr, fromAddr = sp, toReg = r, spCode = Down}
    EndProg            -> def {jmpCode = UR, jmpNum = 0} -- forever loop here
```

内部状态

- 寄存器堆
- 上一条指令ALU的条件判断结果
- 当前PC
- 栈顶指针sp
- ldBuf：用于缓存从memory中读数据的寄存器



```
data PState     = PState { reg :: Reg
                         , cnd :: Bool
                         , pc :: PC
                         , sp :: DAddr
                         , ldBuf :: Vec LoadDelay RegIdx
                         } deriving(Eq, Show)
```

```haskell
esprockellMealy :: PState -> PIn -> (PState, POut)
esprockellMealy state (instr, memData, gpInEn, gpInput) = (state', out)
    where
        MachCode{..}   = decode sp instr
        PState{..}     = state
        (aluOut, aluCnd) = alu opCode (x, y)
        cnd' = if fromReg0 == iReg || opCode == Id then gpInEn else aluCnd
        state'  = PState { reg = reg', cnd = cnd', pc = pc', sp = sp', ldBuf = ldBuf'}
        out     = (toAddr, fromAddr, we, toMem, pc', gpOutEn, gpOut)
        gpOut   = reg' !! oReg
        gpOutEn = oEn bufLast toReg ldCode
        (x, y)  = (reg0 !! fromReg0, reg0 !! fromReg1)
        ldBuf'  = ldReg +>> ldBuf
        bufLast = last ldBuf
        ldReg
          | ldCode == LdAddr = toReg
          | otherwise        = 0
        reg0 = reg  <~ (bufLast, memData)
                    <~ (iReg, gpInput)
                    <~ (iEn, fromIntegral $ pack gpInEn)
                    <~ (zeroreg, 0)         -- r0
                    <~ (pcreg, fromIntegral pc) -- pc of next clock
        reg' = load ldCode toReg (ldImm, aluOut) reg0
        toMem  = store stCode (stImm, x)
        pc'    = updatePC (jmpCode, cnd) (pc, jmpNum, reg' !! jmpreg)
        sp'    = updateSp spCode sp
```

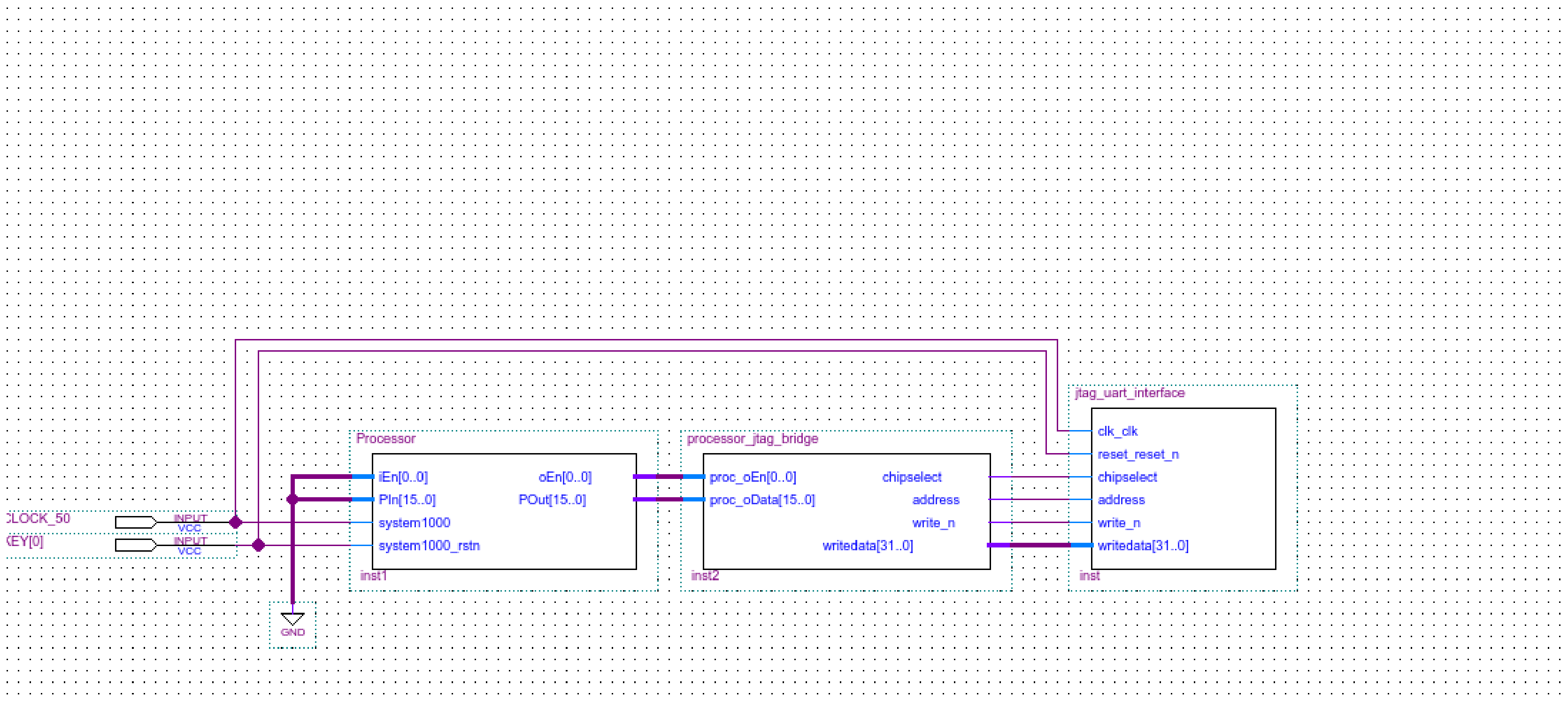
Memory与JTAG UART

- Memory：直接与指令ROM、数据RAM相连，使用了CLaSH内置的asyncRom、blockRam函数
- JTAG UART
  - 实例化qsys中生成的jtag_uart模块，通过赋值synthesis文件夹中的verilog代码
  - 根据avalon总线波形，手写了一个processor的POut到jtag_uart的接口
  - 输出使能做chipselect与write_n信号
  - POut前8位做writedata，第9位做address

```verilog
module processor_jtag_bridge(
    input wire[0:0] proc_oEn
    , input wire[15:0] proc_oData

    , output wire chipselect
    , output wire address
    , output wire write_n
    , output wire[31:0] writedata
);
assign chipselect = proc_oEn;
assign write_n    = ~proc_oEn;
assign address    = proc_oData[8];
assign writedata  = address == 0 ? {24'd0,proc_oData[7:0]} : 32'd2;
endmodule
```

Hello Word系统框图



TODO

- 汇编器
  - label的处理
  - 伪指令，比如mov指令，function call指令
- 中断处理，可以通过外部的控制电路来实现
- Write a simple compiler