PC
=========
```
unsigned add(unsigned a, unsigned b);

model pcu${bit_width}{
  port{
    clock clock;
    in load, reset, hold, data_in[$w_1:0];
    out data_out[$w_1:0];
  }
  storage{
    register reg[$w_1:0];
  }
  default_control{
    load  = '0';
    reset = '0';
    hold  = '1';
  }

  /** no operation*/
  function nop : idle{
    control{
      in load, reset, hold;
    }
    protocol{
      [load == 0 && reset == 0 && hold == 1]{
      }
    }
  }

  /** reset */
  function reset : reset{
    assignment{
      reg = 0;
    }
    control{
      in reset;
    }
    protocol{
      [reset == 1]{
         store reg;
      }
    }
  }

  /** increment */
  function inc{
    assignment{
      reg = add(reg, $inc_step);
    }
  }

  /** write : set program counter value */
  function write{
    input{
      bit_vector data_in;
    }
    assignment{
      reg = data_in;
    }
    control{
      in bit load;
    }
    protocol{
      [load = '1' && hold data_in]{
         store reg;
      }
    }
  }

  /** read : read program counter value */
  function read{
    output{
      bit_vector data_out;
    }
  }

  priority{ ( reset > ( inc | write ) ), read}
}
```

IR
=======
```
/** ${bit_width}-bit register */
model reg${bit_width}{
  port{
    clock clock;
    in    reset, enb;
    in    data_in${range};
    out   data_out${range};
  }
  storage{
    register reg${range};
  }
  default_control{
    reset = 0;
    enb   = 0;
  }

  /** no operation */
  function nop : idle{
    control{
      in reset, enb;
    }
    protocol{
      [reset == 0 && enb == 0]{
      }
    }
  }

  /** reset */
  function reset: reset{
    assignment{
      reg = 0;
    }
    control{
      in reset;
    }protocol{
      [reset == 1]{
         store reg;
      }
    }
  }

  /** write */
  function write{
    input{
      bit_vector data_in;
    }
    assignment{
      reg = data_in;
    }
    control{
      in enb;
    }
    protocol{
      [enb == 1 && hold data_in]{
         store reg;
      }
    }
  }

  /** read */
  function read{
    input{
    }
    output{
      bit_vector data_out = reg;
    }
  }

  priority{ ( reset > ( nop | write )), read }
}
```

IMAU
===========
```
/** ${bit_width}-bit instruction memory access unit */
model imau${bit_width}{
  port{
    in  addr[${a_range}];
    out addr_bus[${a_range}];
    in  data_bus[${b_range}];
    out data[${b_range}];
  }

  /** read */
  function read{
    input{
      bit_vector addr;
    }
    output{
      bit_vector data = data_bus;
    }
    protocol{
      valid data;
    }
  }
}
```

DMAU
===========
```
/** ${bit_width}-bit data memory access unit */
model dmau${bit_width}{
  port{
    in    reset;
    in    req, rw, ${acmode_str}${ext_str}addr[${a_range}],
data_in[${b_range}];
    out   ${aerr_str}req_bus, w_mode_bus$wmode_str,
addr_bus[${a_range}];
    inout data_bus[${b_range}];
    in    ack_bus;
    out   ack, data_out[${b_range}];
  }
  default_control{
    reset   = 0;
    req     = 0;
  }
  /** no operation */
  function nop : idle{
    control{
      in reset, req;
    }
    protocol{
      [reset ==0 && req == 0]{
      }
    }
  }

  /** reset */
  function reset : reset{
    assignment{
      reg = 0;
    }
    control{
      in reset;
    }
    protocol{
      [reset == 0]{
         store reg;
      }
    }
  }

FHM_DL_FUNC
    }
if ($acmode_num != 0){
    {
        print <<FHM_DL_FUNC
  /** load ${bit_width} bits */
  function ld_${bit_width}{
    input{
      bit_vector addr;
    }
    output{
      bit_vector data_out = data_bus;
      bit addr_err = addr_err(addr, ac_mode);
    }
    control{
      in  reset, req, rw, ac_mode;
      out ack;
    }
    protocol{
      repeat [req == 1 && rw == 0 && ac_mode == ${b_str}]
until (ack == 1 || reset == 1);
      if (ack == 1){
         valid data_out;
         valid addr_err;
      }
    }
  }

FHM_DL_FUNC
    }
    for ($i=$wmode_1,$w_b=$bit_width-$access; $i>0; $i--
,$w_b=$w_b-$access){
        $str = $a_t_comma . &to_comp($i-1, $acmode_num) .
$a_t_comma;
        {
            print <<FHM_DL_FUNC
  /** load ${w_b} bits */
  function ld_${w_b}{
    input{
      bit_vector addr;
    }
    output{
      bit_vector data_out = data_bus;
      bit addr_err = addr_err(addr, ac_mode);
    }
    control{
      in  reset, req, rw, ac_mode, ext_mode;
      out ack;
    }
    protocol{
      repeat [req == 1 && rw == 0 && ac_mode == $str &&
ext_mode == 1] until (ack == 1 || reset == 1);
      if (ack == 1){
         valid data_out;
         valid addr_err;
      }
    }
  }

  /** load ${w_b} bits (unsigned) */
  function ldu_${w_b}{
    input{
      bit_vector addr;
    }
    output{
      bit_vector data_out = data_bus;
      bit addr_err = addr_err(addr, ac_mode);
    }
    control{
      in  reset, req, rw, ac_mode, ext_mode;
      out ack;
    }
    protocol{
      repeat [req == 1 && rw == 0 && ac_mode == $str &&
ext_mode == 0] until (ack == 1 || reset == 1);
      if (ack == 1){
         valid data_out;
         valid addr_err;
      }
    }
  }

FHM_DL_FUNC
        }
    }
    {
        print <<FHM_DL_FUNC
  /** store ${bit_width} bits */
  function s_${bit_width}{
    input{
      bit_vector addr;
      bit_vector data_in;
    }
    output{
      bit addr_err = addr_err(addr, ac_mode);
    }
    control{
      in  reset, req, rw, ac_mode;
      out ack;
    }
    protocol{
      repeat [req == 1 && rw == 1 && ac_mode == ${b_str}]
until (ack == 1 || reset == 1);
      if (ack == 1){
         valid addr_err;
      }
    }
  }

FHM_DL_FUNC
    }
    for ($i=$wmode_1,$w_b=$bit_width-$access; $i>0; $i--
,$w_b=$w_b-$access){
        $str = $a_t_comma . &to_comp($i-1, $acmode_num) .
$a_t_comma;
        {
            print <<FHM_DL_FUNC
  /** store ${w_b} bits */
  function s_${w_b}{
    input{
      bit_vector addr;
      bit_vector data_in;
    }
    output{
      bit addr_err = addr_err(addr, ac_mode);
    }
    control{
      in  reset, req, rw, ac_mode;
      out ack;
    }
    protocol{
      repeat [req == 1 && rw == 1 && ac_mode == $str] until
(ack == 1 || reset == 1);
      if (ack == 1){
         valid addr_err;
      }
    }
  }

FHM_DL_FUNC
        }
    }
    {
        print <<FHM_DL_FUNC
  /** load : same as ld_${bit_width} */
  function load{
    input{
      bit_vector addr;
    }
    output{
      bit_vector data_out = data_bus;
      bit addr_err = addr_err(addr, ac_mode);
    }
    control{
      in  reset, req, rw, ac_mode;
      out ack;
    }
    protocol{
      repeat [req == 1 && rw == 0 && ac_mode == ${b_str}]
until (ack == 1 || reset == 1);
      if (ack == 1){
         valid data_out;
         valid addr_err;
      }
    }
  }

  /** read : same as ld_${bit_width} */
  function read{
    input{
      bit_vector addr;
    }
    output{
      bit_vector data_out = data_bus;
      bit addr_err = addr_err(addr, ac_mode);
    }
    control{
      in  reset, req, rw, ac_mode;
      out ack;
    }
    protocol{
      repeat [req == 1 && rw == 0 && ac_mode == ${b_str}]
until (ack == 1 || reset == 1);
      if (ack == 1){
         valid data_out;
         valid addr_err;
      }
    }
  }

  /** lh : same as ld_${b_h} */
  function lh{
    input{
      bit_vector addr;
    }
    output{
      bit_vector data_out = data_bus;
      bit addr_err = addr_err(addr, ac_mode);
    }
    control{
      in  reset, req, rw, ac_mode, ext_mode;
      out ack;
    }
    protocol{
      repeat [req == 1 && rw == 0 && ac_mode == ${b_h_str}
&& ext_mode = 1] until (ack == 1 || reset == 1);
      if (ack == 1){
         valid data_out;
         valid addr_err;
      }
    }
  }

  /** lhu : same as ldu_${b_h} */
  function lhu{
    input{
      bit_vector addr;
    }
    output{
      bit_vector data_out = data_bus;
      bit addr_err = addr_err(addr, ac_mode);
    }
    control{
      in  reset, req, rw, ac_mode, ext_mode;
      out ack;
    }
    protocol{
      repeat [req == 1 && rw == 0 && ac_mode == ${b_h_str}
&& ext_mode = 0] until (ack == 1 || reset == 1);
      if (ack == 1){
         valid data_out;
         valid addr_err;
      }
    }
  }

  /** lb : same as ld_${b_b} */
  function lb{
    input{
      bit_vector addr;
    }
    output{
      bit_vector data_out = data_bus;
      bit addr_err = addr_err(addr, ac_mode);
    }
    control{
      in  reset, req, rw, ac_mode, ext_mode;
      out ack;
    }
    protocol{
      repeat [req == 1 && rw == 0 && ac_mode == ${b_b_str}
&& ext_mode = 1] until (ack == 1 || reset == 1);
      if (ack == 1){
         valid data_out;
         valid addr_err;
      }
    }
  }

  /** lbu : same as ldu_${b_b} */
  function lbu{
    input{
      bit_vector addr;
    }
    output{
      bit_vector data_out = data_bus;
      bit addr_err = addr_err(addr, ac_mode);
    }
    control{
      in  reset, req, rw, ac_mode, ext_mode;
      out ack;
    }
    protocol{
      repeat [req == 1 && rw == 0 && ac_mode == ${b_b_str}
&& ext_mode = 0] until (ack == 1 || reset == 1);
      if (ack == 1){
         valid data_out;
         valid addr_err;
      }
    }
  }

  /** store : same as s_${bit_width} */
  function store{
    input{
      bit_vector addr;
      bit_vector data_in;
    }
    output{
      bit addr_err = addr_err(addr, ac_mode);
    }
    control{
      in  reset, req, rw, ac_mode;
      out ack;
    }
    protocol{
      repeat [req == 1 && rw == 1 && ac_mode == ${b_str}]
until (ack == 1 || reset == 1);
      if (ack == 1){
         valid addr_err;
      }
    }
  }

  /** write : same as s_${bit_width} */
  function write{
    input{
      bit_vector addr;
      bit_vector data_in;
    }
    output{
      bit addr_err = addr_err(addr, ac_mode);
    }
    control{
      in  reset, req, rw, ac_mode;
      out ack;
    }
    protocol{
      repeat [req == 1 && rw == 1 && ac_mode == ${b_str}]
until (ack == 1 || reset == 1);
      if (ack == 1){
         valid addr_err;
      }
    }
  }

  /** sh : same as s_${b_h} */
  function sh{
    input{
      bit_vector addr;
      bit_vector data_in;
    }
    output{
      bit addr_err = addr_err(addr, ac_mode);
    }
    control{
      in  reset, req, rw, ac_mode;
      out ack;
    }
    protocol{
      repeat [req == 1 && rw == 1 && ac_mode == ${b_h_str}]
until (ack == 1 || reset == 1);
      if (ack == 1){
         valid addr_err;
      }
    }
  }

  /** sb : same as s_${b_b} */
  function sb{
    input{
      bit_vector addr;
      bit_vector data_in;
    }
    output{
      bit addr_err = addr_err(addr, ac_mode);
    }
    control{
      in  reset, req, rw, ac_mode;
      out ack;
    }
    protocol{
      repeat [req == 1 && rw == 1 && ac_mode == ${b_b_str}]
until (ack == 1 || reset == 1);
      if (ack == 1){
         valid addr_err;
      }
    }
  }
```

```
}
FHM_DL_FUNC
    }
}
else{
    {
         print <<FHM_DL_FUNC
  /** load : load data */
  function load{
    input{
      bit_vector addr;
    }
    output{
      bit_vector data_out = data_bus;
    }
    control{
      in  reset, req, rw;
      out ack;
    }
    protocol{
      repeat [req == 1 && rw == 0] until (ack == 1 || reset
== 1);
      if (ack == 1){
         valid data_out;
      }
    }
  }

  /** read : same as load */
  function read{
    input{
      bit_vector addr;
    }
    output{
      bit_vector data_out = data_bus;
    }
    control{
      in  reset, req, rw;
      out ack;
    }
    protocol{
      repeat [req == 1 && rw == 0] until (ack == 1 || reset
== 1);
      if (ack == 1){
         valid data_out;
      }
    }
  }

  /** store : store data */
  function store{
    input{
      bit_vector addr;
      bit_vector data_in;
    }
    control{
      in  reset, req, rw;
      out ack;
    }
    protocol{
      repeat [req == 1 && rw == 1] until (ack == 1 || reset
== 1);
    }
  }

  /** write : same as store */
  function write{
    input{
      bit_vector addr;
      bit_vector data_in;
    }
    control{
      in  reset, req, rw;
      out ack;
    }
    protocol{
      repeat [req == 1 && rw == 1] until (ack == 1 || reset
== 1);
    }
  }
}
```

```
GPR
===========
/** $bit_width-bit registerfile with $n_reg registers,
$n_read read port, $n_write $n_write port */
model regfile${bit_width}_${n_reg}_${n_read}_${n_write}{
  port{
    clock clock;
    in reset;
FHM_DL_MODEL
}
print "    in  ";
for ($i=0; $i<=$n_write-1; $i++){
    if ($i == $n_write - 1){print "w_enb$i;\n";}
    else{print "w_enb$i, ";}
}
print "    in  ";
for ($i=0; $i<=$n_write-1; $i++){
    if ($i == $n_write - 1){print
"w_sel${i}[$n_sel_1:0];\n";}
    else{print "w_sel${i}[$n_sel_1:0], ";}
}
print "    in  ";
for ($i=0; $i<=$n_write-1; $i++){
    if ($i == $n_write - 1){print
"data_in${i}[$w_1:0];\n";}
    else{print "data_in${i}[$w_1:0], ";}
}
print "    in  ";
for ($i=0; $i<=$n_read-1; $i++){
    if ($i == $n_read - 1){print
"r_sel${i}[$n_sel_1:0];\n";}
    else{print "r_sel${i}[$n_sel_1:0], ";}
}
print "    out ";
for ($i=0; $i<=$n_read-1; $i++){
    if ($i == $n_read - 1){print
"data_out${i}[$w_1:0];\n";}
    else{print "data_out${i}[$w_1:0], ";}
}
print "  }\n\n";
{
  print <<FHM_DL_NOP1
  /** no operation */
  function nop : idle{
    control{
      in reset;
FHM_DL_NOP1
}
for ($i=0; $i<=$n_write-1; $i++){
    print "      in w_enb$i;\n";
}
{
    print <<FHM_DL_NOP2
    }
    protocol{
FHM_DL_NOP2
}
print "      [reset == '0' && ";
for ($i=0; $i<=$n_write-1; $i++){
    if ($i == $n_write - 1){print "w_enb$i == '0']{\n";}
    else{print "w_enb$i == '0' && ";}
}
{
    print <<FHM_DL_NOP_RESET
    }
  }

  /** reset */
  function reset : reset{
    control{
      in reset;
    }
    protocol{
      [reset == '1']{
      }
    }
  }

FHM_DL_NOP_RESET
}
for ($i=0; $i<=$n_write-1; $i++){
    {
        print <<FHM_DL_WRITE1
  /** write$i */
  function write${i}{
    input{
      bit_vector data_in$i;
    }
    assignment{
FHM_DL_WRITE1
    }
    for ($j=0; $j<=$n_reg-1; $j++){print "     reg$j =
data_in$i;\n";}
    {
        print <<FHM_DL_WRITE2
    }
    control{
      in w_enb$i;
      in w_sel$i;
    }
    protocol{
FHM_DL_WRITE2
    }
    for ($j=0; $j<=$n_reg-1; $j++){
         $j2 = &to_comp($j, $n_sel);
         print "      [w_enb$i == '1' && w_sel$i == \"$j2\"
&& hold data_in$i]{\n";
         print "        store reg$j;\n       }\n";
    }
    print "    }\n  }\n\n";
}

for ($i=0; $i<=$n_read-1; $i++){
    print <<FHM_DL_READ
  /** read$i */
  function read${i}{
    input{
      bit_vector r_sel$i;
    }
    output{
      bit_vector data_out$i;
    }
  }

FHM_DL_READ
}
print "  priority{ ( reset > ( nop | ";
for ($i=0; $i<=$n_write-1; $i++){
    if ($i == $n_write - 1){print "write$i ) ), ";}
    else{print "write$i | ";}
}
for ($i=0; $i<=$n_read-1; $i++){
    if ($i == $n_read - 1){print "read$i }\n}\n";}
    else{print "read$i, ";}
}


ALU0
==========
unsigned   addu(twoscomp a, twoscomp b);
unsigned   subu(twoscomp a, twoscomp b);
unsigned   add(twoscomp a, twoscomp b);
unsigned   sub(twoscomp a, twoscomp b);
unsigned   and(twoscomp a, twoscomp b);
unsigned   or(twoscomp a, twoscomp b);
unsigned   xor(twoscomp a, twoscomp b);
unsigned   nor(twoscomp a, twoscomp b);
unsigned   cmpu(twoscomp a, twoscomp b);
unsigned   cmp(twoscomp a, twoscomp b);
unsigned   cmpzu(twoscomp a);
unsigned   cmpz(twoscomp a);
signed     inc(twoscomp a);
unsigned   incu(twoscomp a);
unsigned   dec(twoscomp a);
unsigned   cdec(twoscomp a);
unsigned   caddu(twoscomp a, twoscomp b);
signed     cadd(twoscomp a, twoscomp b);
unsigned   csubu(twoscomp a, twoscomp b);
signed     csub(twoscomp a, twoscomp b);
bit_vector alu_flag(bit_vector mode, bit_vector a,
bit_vector b, bit cin);

/** ${bit_width}-th alu */
model alu${bit_width}{
  port{
    in  a[${bit_width_1}:0], b[${bit_width_1}:0], cin,
mode[4:0];
    out result[${bit_width_1}:0], flag[3:0];
  }

   /** C is '1' when (carry-occurred or not-bollowed) and
unsigned-mode else '0'
       V is '1' when overflowed and signed-mode else '0'
       S is equal to MSB of result
       Z is '1' when result = 0 else '0' */

  /** caddu : unsigned clip add, flag(3):C, flag(2):Z,
flag(1):S, flag(0)=0 */
  function caddu{
    input{
      unsigned a, b;
    }
    output{
      unsigned   result = caddu(a, b);
      bit_vector flag = alu_flag(mode, a, b);
    }
    control{
      in mode;
      in cin;
    }
    protocol{
      [mode == "00101" && cin == '0']{
        valid result;
        valid flag;
      }
    }
  }

  /** cadd : signed clip add, flag(3):C, flag(2):Z,
flag(1):S, flag(0):V */
  function cadd{
    input{
      unsigned a, b;
    }
    output{
      unsigned   result = cadd(a, b);
      bit_vector flag = alu_flag(mode, a, b);
    }
    control{
      in mode;
      in cin;
    }
    protocol{
      [mode == "01101" && cin == '0']{
        valid result;
        valid flag;
      }
    }
  }

  /** csubu : unsigned clip subtract, flag(3):C, flag(2):Z,
flag(1):S, flag(0)=0 */
  function csubu{
    input{
      unsigned a, b;
    }
    output{
      unsigned   result = csubu(a, b);
      bit_vector flag = alu_flag(mode, a, b);
    }
    control{
      in mode;
      in cin;
    }
    protocol{
      [mode == "00110" && cin == '1']{
        valid result;
        valid flag;
      }
    }
  }

  /** csub : signed clip subtract, flag(3):C, flag(2):Z,
flag(1):S, flag(0):V */
  function csub{
    input{
      unsigned a, b;
    }
    output{
      unsigned   result = csub(a, b);
      bit_vector flag = alu_flag(mode, a, b);
    }
    control{
      in mode;
      in cin;
    }
    protocol{
      [mode == "01110" && cin == '1']{
        valid result;
        valid flag;
      }
    }
  }

  /** addu : unsigned add, flag(3):C, flag(2):Z, flag(1):S,
flag(0)=0 */
  function addu{
    input{
      unsigned a, b;
    }
    output{
      unsigned   result = addu(a, b);
      bit_vector flag = alu_flag(mode, a, b);
    }
    control{
      in mode;
      in cin;
    }
    protocol{
      [mode == "00001" && cin == '0']{
        valid result;
        valid flag;
      }
    }
  }

  /** subu : unsigned sub, flag(3):C, flag(2):Z, flag(1):S,
flag(0)=0 */
  function subu{
    input{
      unsigned a, b;
    }
    output{
      unsigned   result = subu(a, b, cin);
      bit_vector flag = alu_flag(mode, a, b, cin);
    }
    control{
      in mode;
      in cin;
    }
    protocol{
      [mode == "00010" && cin == '1']{
        valid result;
        valid flag;
      }
    }
  }

  /** add : signed add, flag(3):C, flag(2):Z, flag(1):S,
flag(0):V */
  function add{
    input{
      unsigned a, b;
    }
    output{
      unsigned   result = add(a, b);
      bit_vector flag = alu_flag(mode, a, b, cin);
    }
    control{
      in mode;
      in cin;
    }
    protocol{
      [mode == "01001" && cin == '0']{
        valid result;
        valid flag;
      }
    }
  }

  /** sub : signed sub, flag(3):C, flag(2):Z, flag(1):S,
flag(0):V */
  function sub{
    input{
      unsigned a, b;
    }
    output{
      unsigned   result = sub(a, b);
      bit_vector flag = alu_flag(mode, a, b, cin);
    }
    control{
      in mode;
      in cin;
    }
    protocol{
      [mode == "01010" && cin == '1']{
        valid result;
        valid flag;
      }
    }
  }

  /** and : and, flag(3)=0, flag(2):Z, flag(1):S, flag(0)=0
*/
  function and{
    input{
      unsigned a, b;
    }
    output{
      unsigned   result = and(a, b);
      bit_vector flag = alu_flag(mode, a, b, cin);
    }
    control{
      in mode;
    }
    protocol{
      [mode == "10010"]{
        valid result;
        valid flag;
      }
    }
  }

  /** or : or, flag(3)=0, flag(2):Z, flag(1):S, flag(0)=0
*/
  function or{
    input{
      unsigned a, b;
    }
    output{
      unsigned   result = or(a, b);
      bit_vector flag = alu_flag(mode, a, b, cin);
    }
    control{
      in mode;
    }
    protocol{
      [mode == "10000"]{
        valid result;
        valid flag;
      }
    }
  }

  /** xor : xor, flag(3)=0, flag(2):Z, flag(1):S, flag(0)=0
*/
  function xor{
    input{
      unsigned a, b;
    }
    output{
      unsigned   result = xor(a, b);
      bit_vector flag = alu_flag(mode, a, b, cin);
    }
    control{
      in mode;
    }
    protocol{
      [mode == "10001"]{
        valid result;
        valid flag;
      }
      }
    }

  /** nor : nor, flag(3)=0, flag(2):Z, flag(1):S, flag(0)=0
*/
  function nor{
    input{
      unsigned a, b;
    }
    output{
      unsigned   result = nor(a, b);
      bit_vector flag = alu_flag(mode, a, b, cin);
    }
    control{
      in mode;
    }
    protocol{
      [mode == "11000"]{
        valid result;
        valid flag;
      }
    }
  }

  /** cmpu : unsigned comp, flag(3):C, flag(2):Z,
flag(1):S, flag(0)=0 */
  function cmpu{
    input{
      unsigned a, b;
    }
    output{
      unsigned   result = cmpu(a, b);
      bit_vector flag = alu_flag(mode, a, b, cin);
    }
    control{
      in mode;
      in cin;
    }
    protocol{
      [mode == "00010" && cin = '1']{
        valid result;
        valid flag;
      }
    }
  }

  /** cmp : signed comp, flag(3):C, flag(2):Z, flag(1):S,
flag(0):V */
  function cmp{
    input{
      unsigned a, b;
    }
    output{
      unsigned   result = cmp(a, b);
      bit_vector flag = alu_flag(mode, a, b, cin);
    }
    control{
      in mode;
      in cin;
    }
    protocol{
      [mode == "01010" && cin = '1']{
        valid result;
        valid flag;
      }
    }
  }

  /** cmpzu : unsigned comp with zero, flag(3):C,
flag(2):Z, flag(1):S, flag(0):0 */
  function cmpzu{
    input{
      unsigned a;
    }
    output{
      unsigned   result = cmpzu(a);
      bit_vector flag = alu_flag(mode, a, b, cin);
    }
    control{
      in mode;
      in cin;
    }
    protocol{
      [mode == "00000" && cin = '0']{
        valid result;
        valid flag;
      }
    }
  }

  /** cmpz : signed comp with zero, flag(3):C, flag(2):Z,
flag(1):S, flag(0):0 */
  function cmpz{
    input{
      unsigned a;
    }
    output{
      unsigned   result = cmpz(a);
      bit_vector flag = alu_flag(mode, a, b, cin);
    }
    control{
      in mode;
      in cin;
    }
    protocol{
      [mode == "01000" && cin = '0']{
         valid result;
         valid flag;
      }
    }
  }

  /** inc : inc, flag(3):C, flag(2):Z, flag(1):S, flag(0):V
*/
  function inc{
    input{
      unsigned a;
    }
    output{
      unsigned   result = inc(a);
      bit_vector flag = alu_flag(mode, a, b, cin);
    }
    control{
      in mode;
      in cin;
    }
    protocol{
      [mode == "01000" && cin = '1']{
         valid result;
         valid flag;
      }
    }
  }

  /** incu : unsigned inc, flag(3):C, flag(2):Z, flag(1):S,
flag(0):0 */
  function incu{
    input{
      unsigned a;
    }
    output{
      unsigned   result = incu(a);
      bit_vector flag = alu_flag(mode, a, b, cin);
    }
    control{
      in mode;
      in cin;
    }
    protocol{
      [mode == "00000" && cin = '1']{
         valid result;
         valid flag;
      }
    }
  }

  /** dec : unsigned dec, flag(3):C, flag(2):Z, flag(1):S,
flag(0):0 */
  function dec{
    input{
      unsigned a;
    }
    output{
      unsigned   result = dec(a);
      bit_vector flag = alu_flag(mode, a, b, cin);
    }
    control{
      in mode;
      in cin;
    }
    protocol{
      [mode == "00011" && cin = '0']{
         valid result;
         valid flag;
      }
    }
  }

  /** cdec : unsigned dec(clip), flag(3):C, flag(2):Z,
flag(1):S, flag(0):0 */
  function cdec{
    input{
      unsigned a;
    }
    output{
      unsigned   result = cdec(a);
      bit_vector flag = alu_flag(mode, a, b, cin);
    }
    control{
      in mode;
      in cin;
    }
    protocol{
      [mode == "00111" && cin = '0']{
         valid result;
         valid flag;
      }
    }
```

```
  }
}

EXT0
======
unsigned extz(unsigned data_in);
unsigned exts(unsigned data_in);

/** ${bit_width}-bit extender : ${bit_width}-bit to
${bit_width_out}-bit */
model extender${bit_width}_${bit_width_out}{
  port{
    in data_in[$w:0], mode;
    out data_out[$w2:0];
  }

  /** zero : zero extention */
  function zero{
    input{
      unsigned data_in;
    }
    output{
      unsigned data_out = extz(a);
    }
    control{
      in mode;
    }
    protocol{
      [mode == '0']{
         valid data_out;
      }
    }
  }

  /** sign : sign extention */
  function sign{
    input{
      unsigned data_in;
    }
    output{
      unsigned data_out = exts(a);
    }
    control{
      in mode;
    }
    protocol{
      [mode == '1']{
         valid data_out;
      }
    }
  }
}

MUL0
=======

DIV0
=======

SFT0
=======
unsigned shift(unsigned data_in, unsigned mode);\n\n";
/** ${bit_width}-bit shifter : ${info} */
model shifter_var{
  port{
    in  data_in[${bit_width_1}:0], mode,
ctrl[${ctrl_width_1}:0];
    out data_out[${bit_width_1}:0];
  }

  /** shift left logical */
  function sll{
    input{
      unsigned data_in;
      unsigned ctrl;
    }
    output{
      unsigned data_out = ${func}(${func_par});
    }
    control{
      unsigned mode;
    }
    protocol{
      [mode == "00"]{
         valid data_out;
      }
    }
  }

  /** shift left arithmetic */
  function sla{
    input{
      unsigned data_in;
      unsigned ctrl;
    }
    output{
      unsigned data_out = ${func}(${func_par});
    }
    control{
      unsigned mode;
    }
    protocol{
      [mode == "01"]{
        valid data_out;
      }
    }
  }

  /** shift right logical */
  function srl{
    input{
      unsigned data_in;
      unsigned ctrl;
    }
    output{
      unsigned data_out = ${func}(${func_par});
    }
    control{
      unsigned mode;
    }
    protocol{
      [mode == "10"]{
        valid data_out;
      }
    }
  }

  /** shift right arithmetic */
  function sra{
    input{
      unsigned data_in;
      unsigned ctrl;
    }
    output{
      unsigned data_out = ${func}(${func_par});
    }
    control{
      unsigned mode;
    }
    protocol{
      [mode == "11"]{
        valid data_out;
      }
    }
  }

EXT1
=======
```