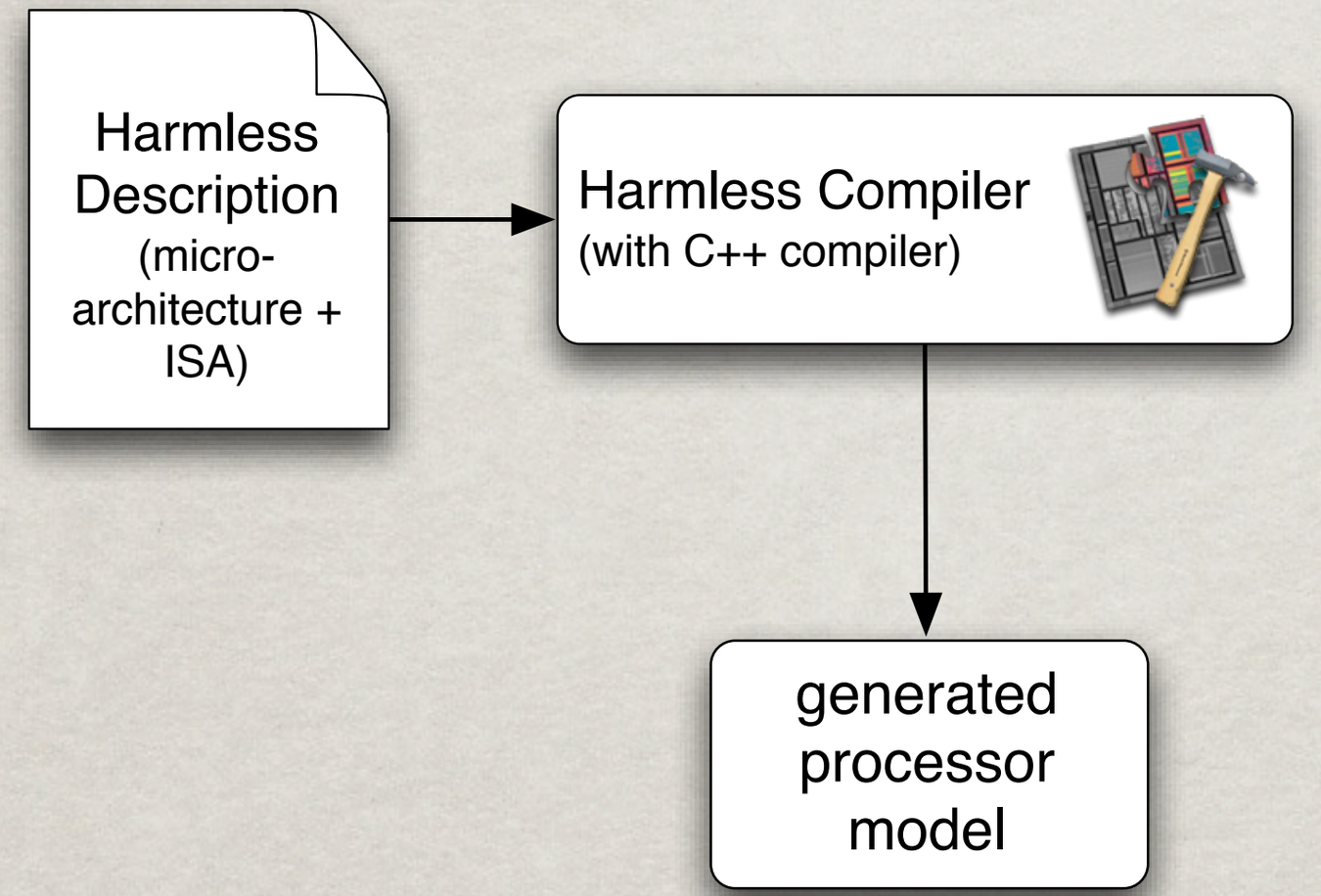
Modélisation Mémoire

Réunion 29/06/2011

Mik

Harmless architecture



Harmless architecture

Harmless Description
(micro-architecture + ISA)

Harmless Compiler
(with C++ compiler)

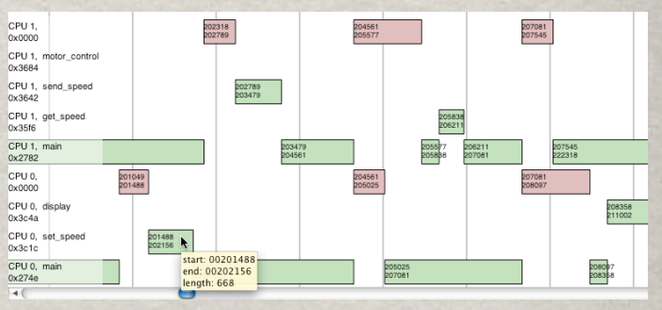
binary code of application (.elf, .H86, .s19, ..)

0100110100101001010
1011011101010110101
1001110101011010100

generated processor model

results

Task analysis
Stack analysis
Temporal measures
"classical" simulation
...



Harmless architecture

Harmless Description
(micro-architecture + ISA)

Harmless Compiler
(with C++ compiler)

binary code of application (.elf, .H86, .s19, ..)

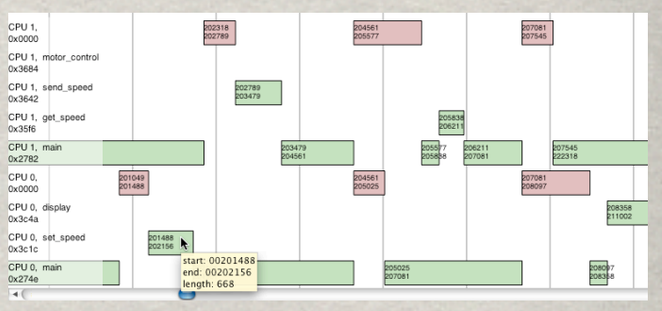
0100110100101001010
1011011101010110100
1001110101011010100

generated processor model

user commands

results

Task analysis
Stack analysis
Temporal measures
"classical" simulation
...

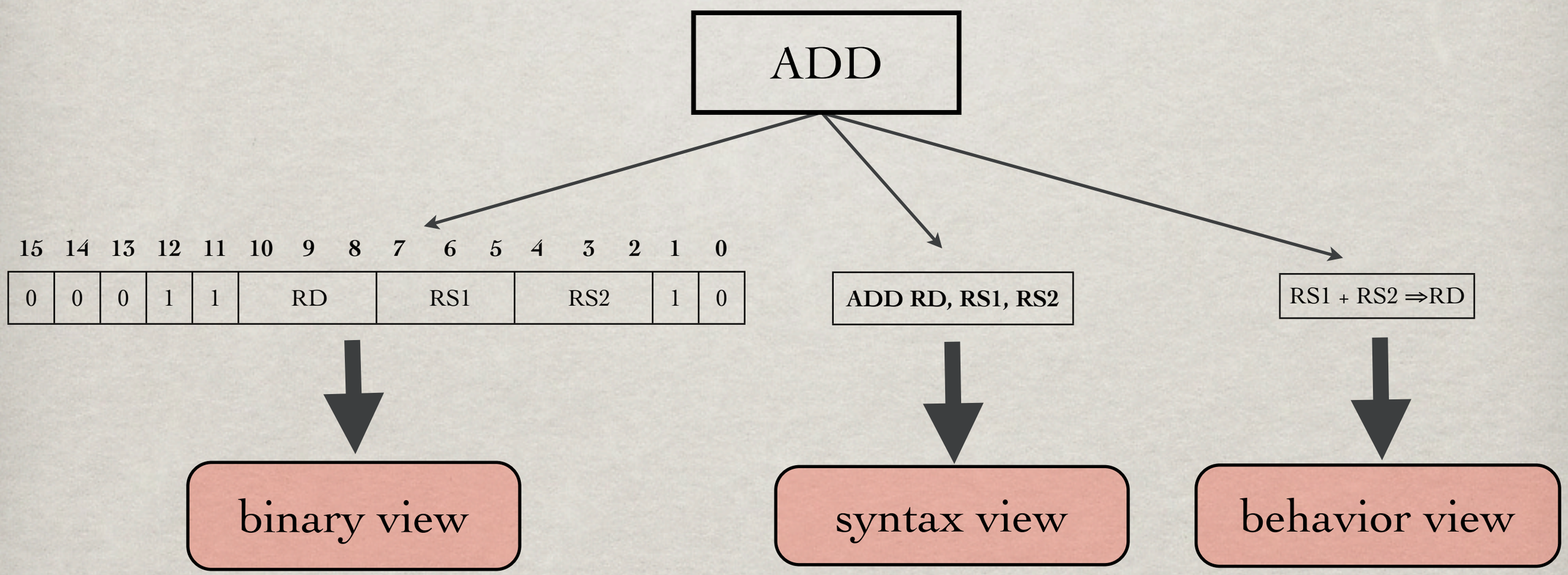


Plan

- **Bref rappel de la description du jeu d'instruction**
- Modélisation fonctionnelle de la mémoire
- description de la µ-architecture (nouvelle mouture)
- pistes pour l'intégration de la hiérarchie mémoire

Instruction Set Architecture

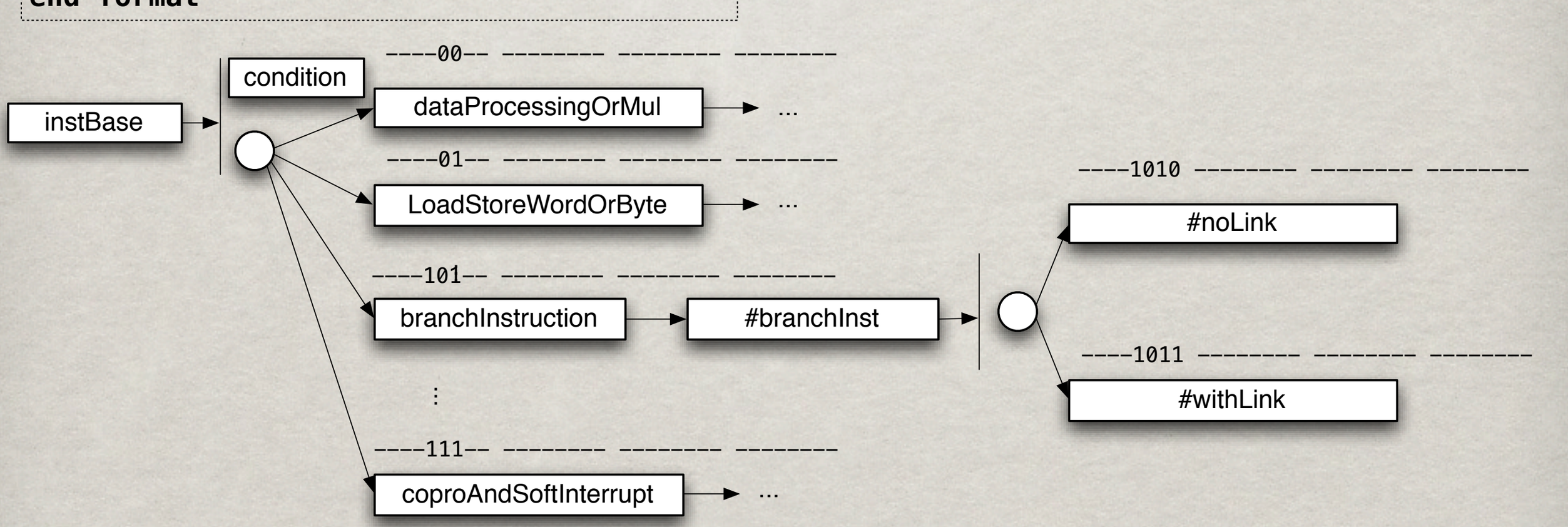
- Instructions are split in 3 views



Binary View

```
format instBase
  condition
  select slice{27..25}
    case \m00- is dataProcessingOrMul
    case \m01- is LoadStoreWordOrByte
    case \b100 is LoadStoreMultiple
    case \b101 is branchInstruction
    case \b111 is coproAndSoftInterrupt
  end select
end format
```

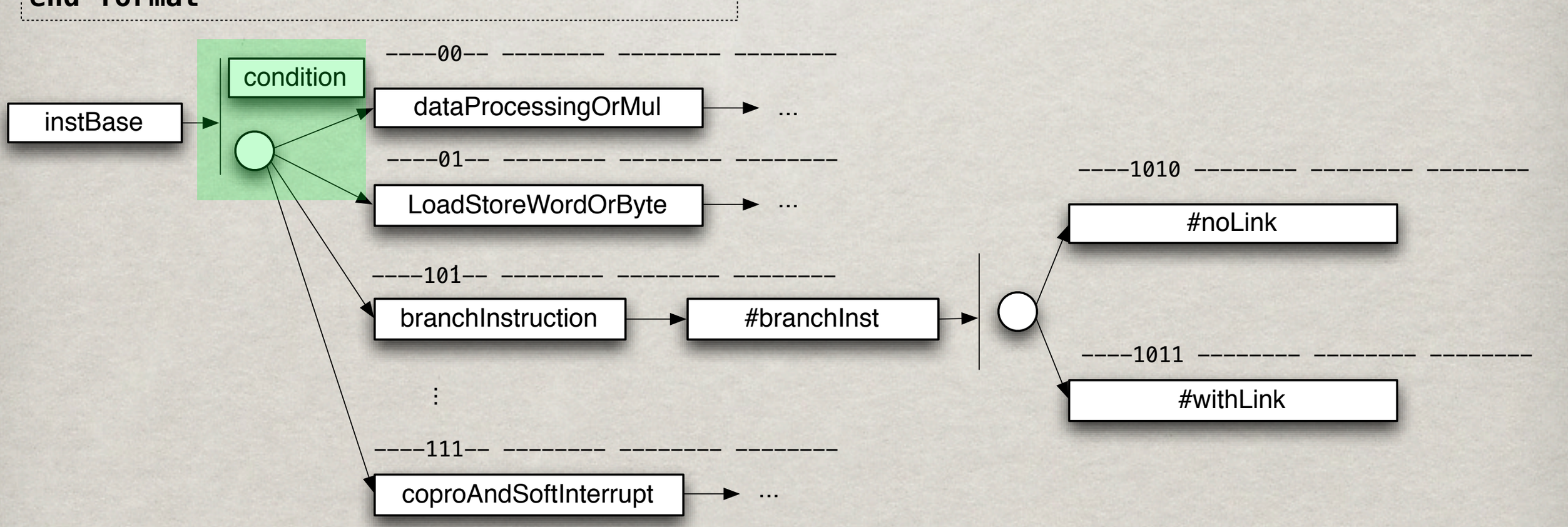
```
format branchInstruction #branchInst
  select slice{24}
    case 0 is #noLink
    case 1 is #withLink
  end select
  offset := signed slice {23..0}
end format
```



Binary View

```
format instBase
  condition
  select slice{27..25}
    case \m00- is dataProcessingOrMul
    case \m01- is LoadStoreWordOrByte
    case \b100 is LoadStoreMultiple
    case \b101 is branchInstruction
    case \b111 is coproAndSoftInterrupt
  end select
end format
```

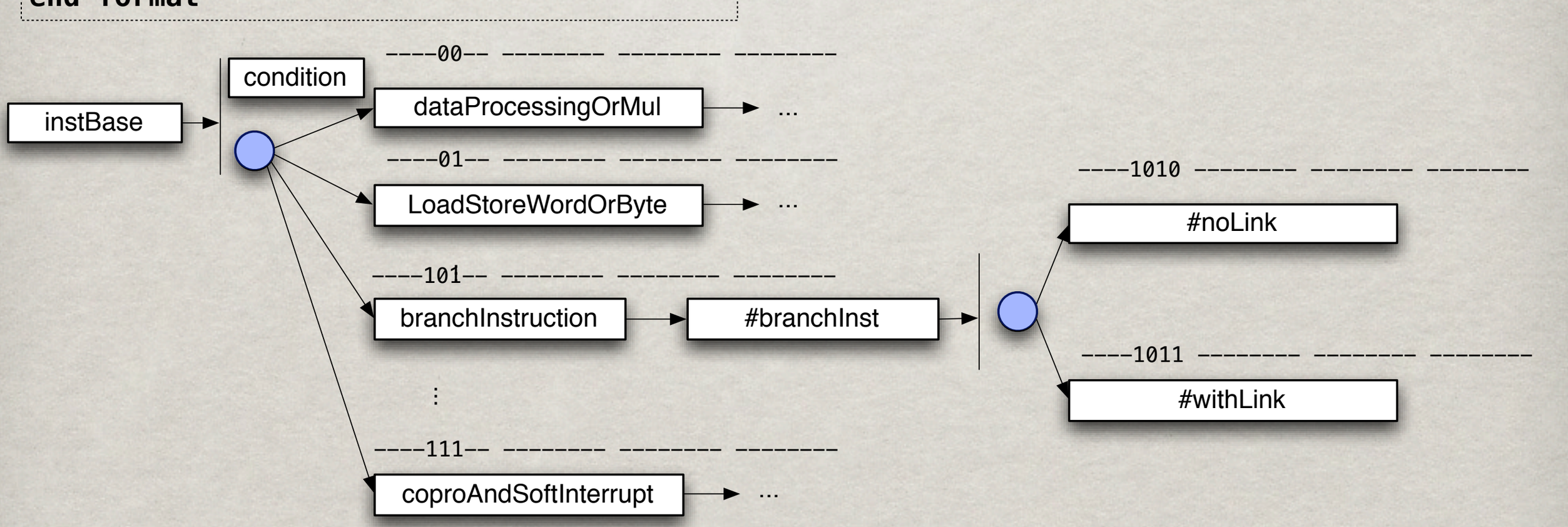
```
format branchInstruction #branchInst
  select slice{24}
    case 0 is #noLink
    case 1 is #withLink
  end select
  offset := signed slice {23..0}
end format
```



Binary View

```
format instBase
  condition
  select slice{27..25}
    case \m00- is dataProcessingOrMul
    case \m01- is LoadStoreWordOrByte
    case \b100 is LoadStoreMultiple
    case \b101 is branchInstruction
    case \b111 is coproAndSoftInterrupt
  end select
end format
```

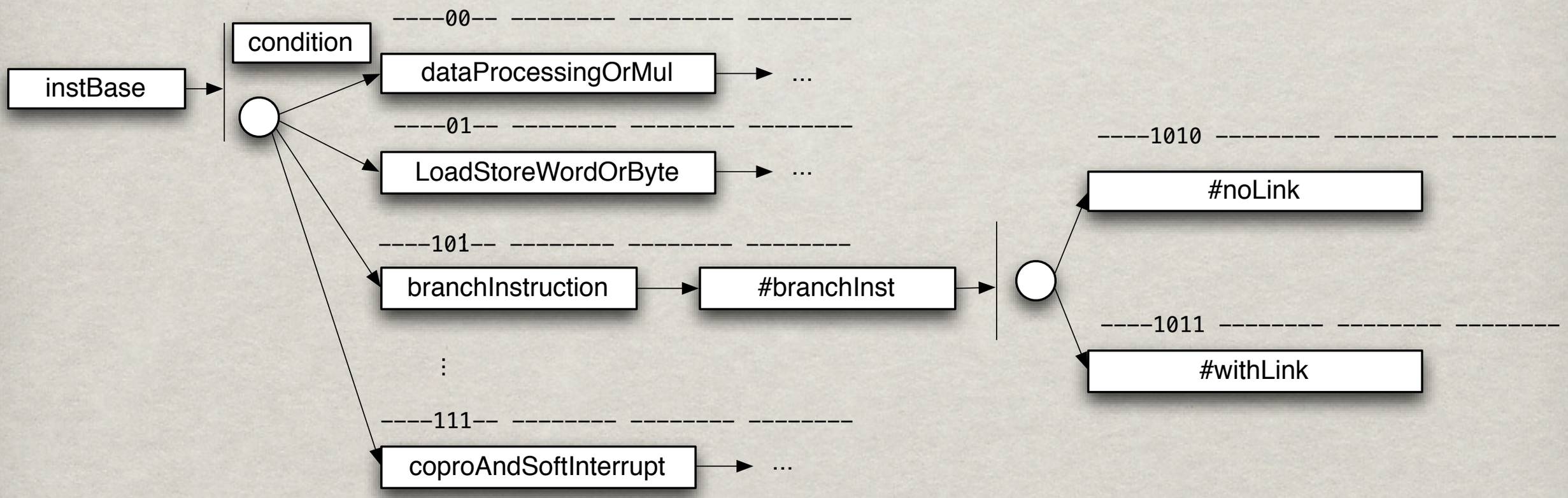
```
format branchInstruction #branchInst
  select slice{24}
    case 0 is #noLink
    case 1 is #withLink
  end select
  offset := signed slice {23..0}
end format
```



Binary View

```
format instBase
  condition
  select slice{27..25}
    case \m00- is dataProcessingOrMul
    case \m01- is LoadStoreWordOrByte
    case \b100 is LoadStoreMultiple
    case \b101 is branchInstruction
    case \b111 is coproAndSoftInterrupt
  end select
end format
```

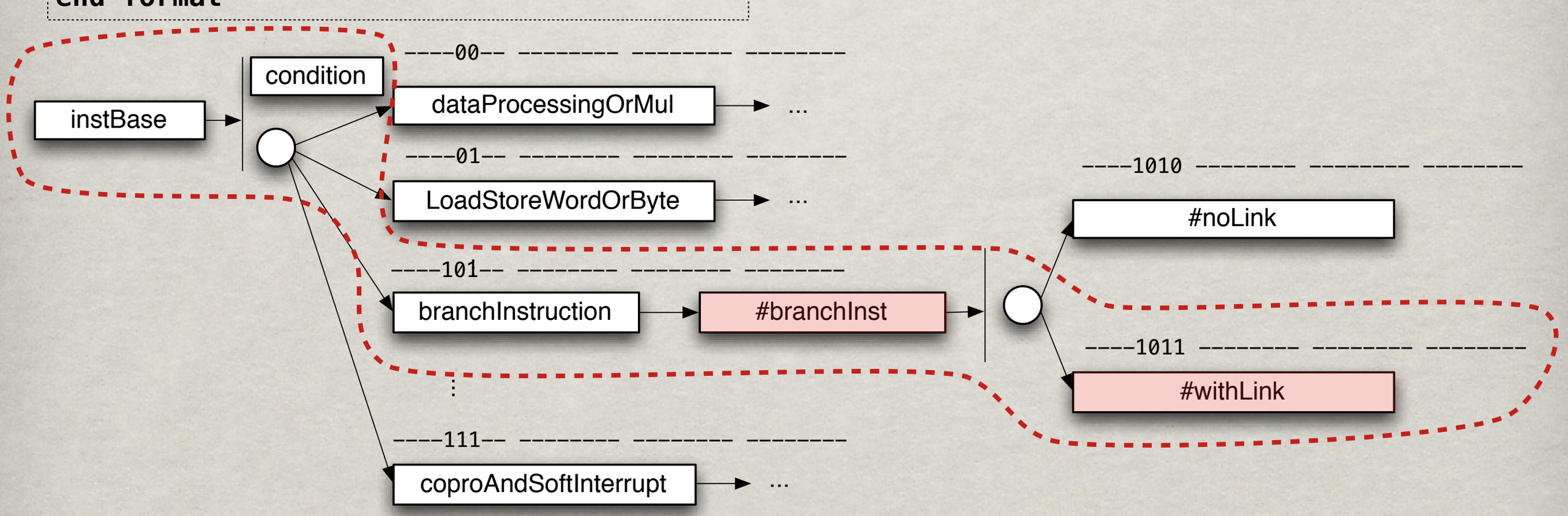
```
format branchInstruction #branchInst
  select slice{24}
    case 0 is #noLink
    case 1 is #withLink
  end select
  offset := signed slice {23..0}
end format
```



Binary View

```
format instBase
  condition
  select slice{27..25}
    case \m00- is dataProcessingOrMul
    case \m01- is LoadStoreWordOrByte
    case \b100 is LoadStoreMultiple
    case \b101 is branchInstruction
    case \b111 is coproAndSoftInterrupt
  end select
end format
```

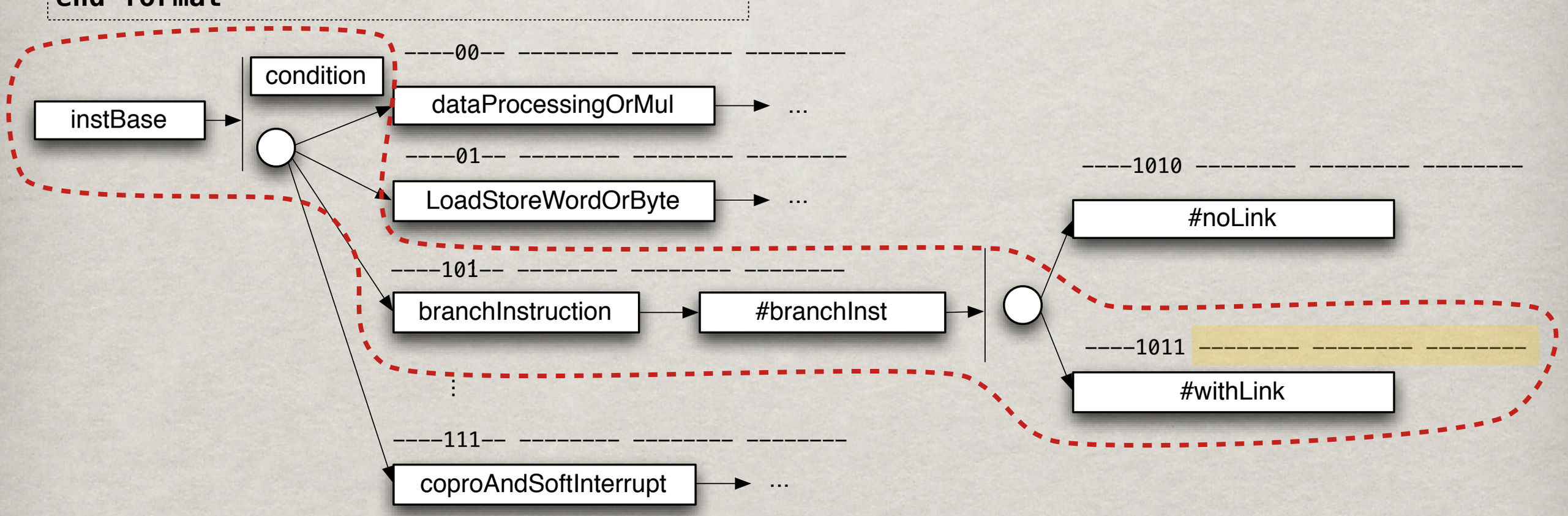
```
format branchInstruction #branchInst
  select slice{24}
    case 0 is #noLink
    case 1 is #withLink
  end select
  offset := signed slice {23..0}
end format
```



Binary View

```
format instBase
  condition
  select slice{27..25}
    case \m00- is dataProcessingOrMul
    case \m01- is LoadStoreWordOrByte
    case \b100 is LoadStoreMultiple
    case \b101 is branchInstruction
    case \b111 is coproAndSoftInterrupt
  end select
end format
```

```
format branchInstruction #branchInst
  select slice{24}
    case 0 is #noLink
    case 1 is #withLink
  end select
  offset := signed slice {23..0}
end format
```



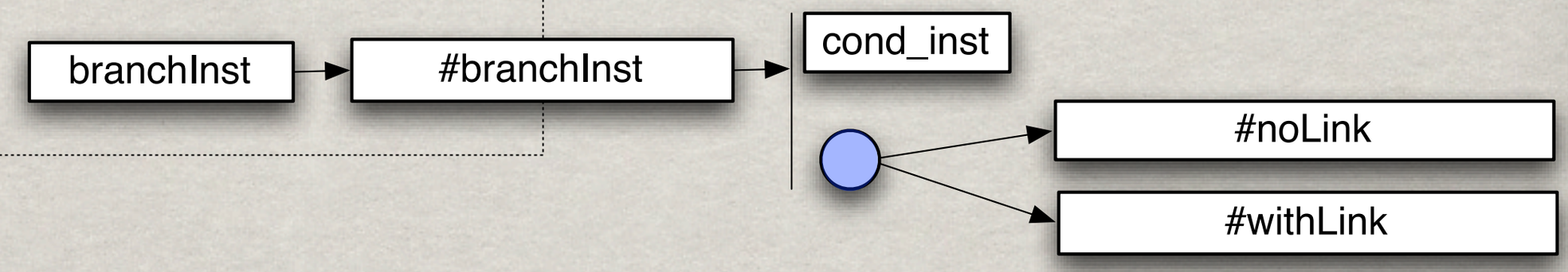
Behavior View

```
behavior branchInst #branchInst
  field s24 offset
  u1 condPassed
  cond_inst(condPassed)

  select
    case #withLink
      do
        if condPassed then
          SRU.writeR32(14,PC)
          SRU.PC_addOffset(offset)
        end if
      end do
    case #noLink
      do
        if condPassed then
          SRU.PC_addOffset(offset)
        end if
      end do
  end select
end behavior
```

Instruction signature is the **set** of tags of a branch

#branchInst #withLink



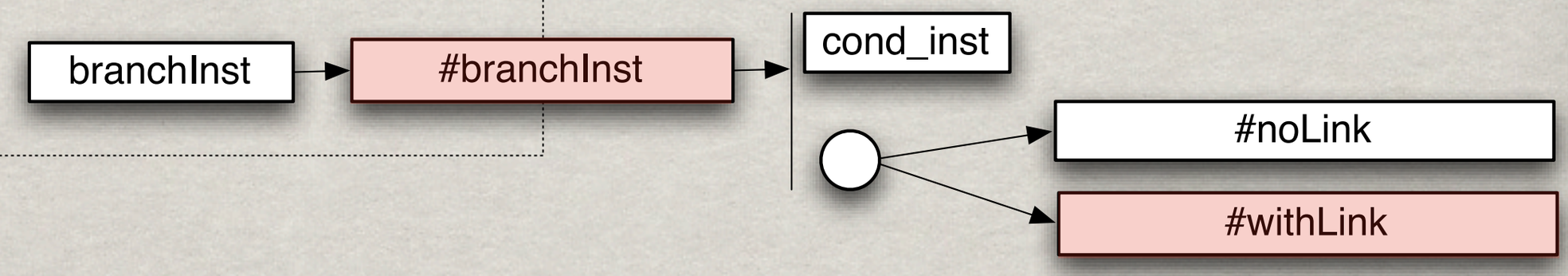
Behavior View

```
behavior branchInst #branchInst
  field s24 offset
  u1 condPassed
  cond_inst(condPassed)

  select
    case #withLink
      do
        if condPassed then
          SRU.writeR32(14,PC)
          SRU.PC_addOffset(offset)
        end if
      end do
    case #noLink
      do
        if condPassed then
          SRU.PC_addOffset(offset)
        end if
      end do
  end select
end behavior
```

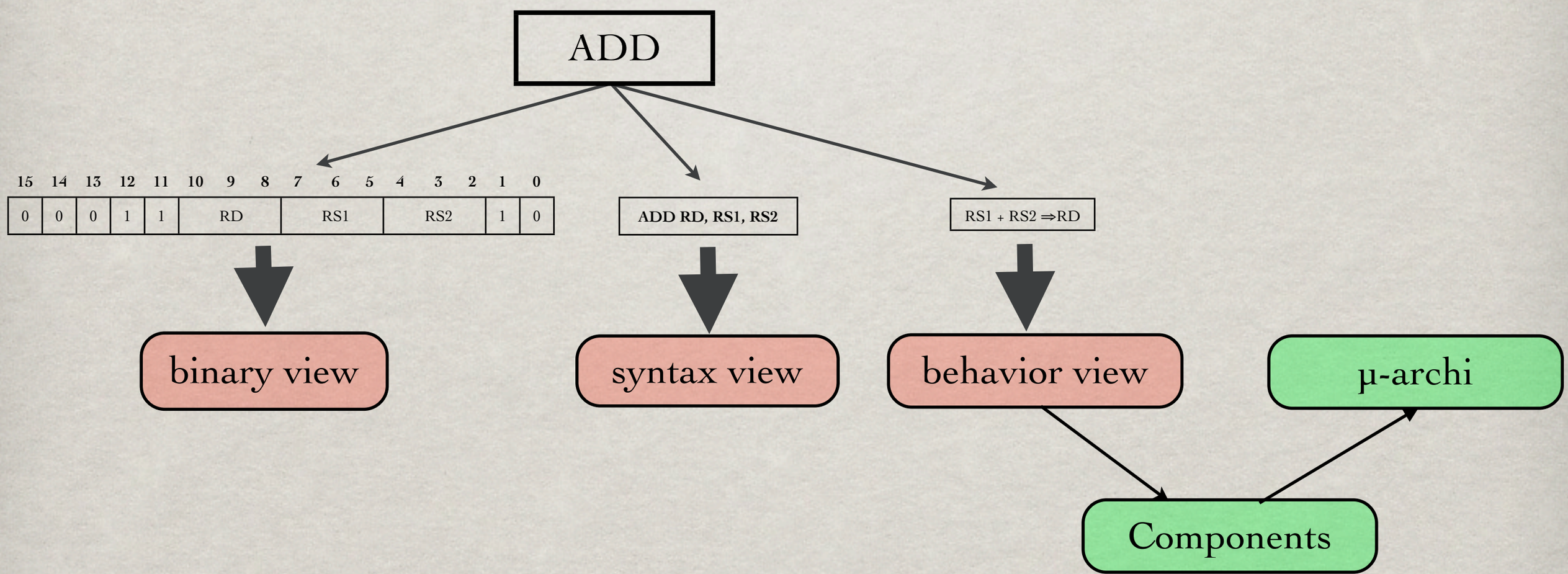
Instruction signature is the **set** of tags of a branch

#branchInst #withLink



Components

- Objectif: interface entre l'ISA et la description de la micro-architecture.



Plan

- Bref rappel de la description du jeu d'instruction
- **Modélisation fonctionnelle de la mémoire**
- description de la µ-architecture (nouvelle mouture)
- pistes pour l'intégration de la hiérarchie mémoire

Functional description of memory

- Defined inside a component

component description

```
component mem {
  program memory  ram{
    width   := 32
    address := 0..32mb
    type    := RAM
  }
}
```

usage inside the behavior view

```
u32 mem.ram.read32(u32 address)
u16 mem.ram.read16(u32 address)
u8  mem.ram.read8( u32 address)

void mem.ram.write32(u32 address, u32 value)
void mem.ram.write16(u32 address, u16 value)
void mem.ram.write8( u32 address, u8 value )

u32 mem.read32(u32 address)
u16 mem.read16(u32 address)
u8  mem.read8( u32 address)

void mem.write32(u32 address, u32 value)
void mem.write16(u32 address, u16 value)
void mem.write8( u32 address, u8 value )
```

Functional description of memory

- Defined inside a component

component description

```
component mem {
  program memory  ram{
    width   := 32
    address := 0..32mb
    type    := RAM
  }
}
```

usage inside the behavior view

```
u32 mem.ram.read32(u32 address)
u16 mem.ram.read16(u32 address)
u8  mem.ram.read8( u32 address)

void mem.ram.write32(u32 address, u32 value)
void mem.ram.write16(u32 address, u16 value)
void mem.ram.write8( u32 address, u8 value )

u32 mem.read32(u32 address)
u16 mem.read16(u32 address)
u8  mem.read8( u32 address)

void mem.write32(u32 address, u32 value)
void mem.write16(u32 address, u16 value)
void mem.write8( u32 address, u8 value )
```

Functional description of memory

- Defined inside a component

component description

```
component mem {
  program memory  ram{
    width   := 32
    address := 0..32mb
    type    := RAM
  }
}
```

usage inside the behavior view

```
u32 mem.ram.read32(u32 address)
u16 mem.ram.read16(u32 address)
u8  mem.ram.read8( u32 address)

void mem.ram.write32(u32 address, u32 value)
void mem.ram.write16(u32 address, u16 value)
void mem.ram.write8( u32 address, u8 value )

u32 mem.read32(u32 address)
u16 mem.read16(u32 address)
u8  mem.read8( u32 address)

void mem.write32(u32 address, u32 value)
void mem.write16(u32 address, u16 value)
void mem.write8( u32 address, u8 value )
```

Functional description of memory

```
component sram {
  memory ram {
    width   := 16  -- get 16 bits / access
    address := \x0..\x10FF
    type    := RAM

    GPR {
      width   := 16  -- get 16 bits / access
      address := 0..31
      stride  := 1
      type    := register
    }
    register u16 X maps to \x1a
    register u16 Y maps to \x1c
    register u16 Z maps to \x1e

    sfr { -- IN/OUT instructions.
      address := 0..\x3F
      type    := register
    } maps to \x20


    register u8  SPH  maps to \x5e -- stack H
    register u8  SPL  maps to \x5d -- stack L
    register u16 SP   maps to \x5d -- stack

    register u8 CCR maps to \x5f {
      C := slice{0} -- carry flag
      Z := slice{1} -- zero flag
      N := slice{2} -- neg flag
      V := slice{3} -- overflow flag
      S := slice{4} -- sign bit
      H := slice{5} -- half carry flag
      T := slice{6} -- Bit copy storage
      I := slice{7} -- global interrupt flag
    }
  }

  void push(u8 val) {     -- post decrement
    sram.write8(SP, val)
    SP := SP-1
  }

  u8 pop() {
    u8 result
    SP := (u16)(SP+1)
    result := sram.read8(SP)
    return result
  }
}
```

Functional description of memory

```
component sram {
  memory ram {
    width   := 16  -- get 16 bits / access
    address := \x0..\x10FF
    type    := RAM

    GPR {
      width   := 16  -- get 16 bits / access
      address := 0..31
      stride  := 1
      type    := register
    }
    register u16 X maps to \x1a
    register u16 Y maps to \x1c
    register u16 Z maps to \x1e

    sfr { -- IN/OUT instructions.
      address := 0..\x3F
      type    := register
    } maps to \x20


    register u8  SPH  maps to \x5e -- stack H
    register u8  SPL  maps to \x5d -- stack L
    register u16 SP   maps to \x5d -- stack
```

```
register u8 CCR maps to \x5f {
  C := slice{0} -- carry flag
}
}
}

void push(u8 val) {      -- post decrement
  sram.write8(SP, val)
  SP := SP-1
}

u8 pop() {
  u8 result
  SP := (u16)(SP+1)
  result := sram.read8(SP)
  return result
}
}
```

define a memory chunk inside another memory chunk

```
u16 sram.ram.GPR.read16(u32 address)
u16 sram.ram.SFR.read16(u32 address)
```

Functional description of memory

```
component sram {
  memory ram {
    width   := 16  -- get 16 bits / access
    address := \x0..\x10FF
    type    := RAM

    GPR {
      width   := 16  -- get 16 bits / access
      address := 0..31
      stride  := 1
      type    := register
    }
    register u16 X maps to \x1a
    register u16 Y maps to \x1c
    register u16 Z maps to \x1e

    sfr { -- IN/OUT instructions.
      address := 0..\x3F
      type    := register
    } maps to \x20


    register u8  SPH  maps to \x5e -- stack H
    register u8  SPL  maps to \x5d -- stack L
    register u16 SP   maps to \x5d -- stack
```

```
register u8 CCR maps to \x5f {
  C := slice{0} -- carry flag
```

define a memory chunk inside another memory chunk

```
u16 sram.ram.GPR.read16(u32 address)
u16 sram.ram.SFR.read16(u32 address)
```

```
  }
}

void push(u8 val) {      -- post decrement
  sram.write8(SP, val)
  SP := SP-1
}
```

```
u16 sram.ram.SFR.read16(address)
<=>
u16 sram.ram.read16(address + \x20)
```

```
  result := sram.read8(SP)
  return result
}
}
```

Functional description of memory

Registers mapped into memory. Even with slices.

```
com                                          {
  m
    address := 0..31
    stride  := 1
    type    := register
  }
  register u16 X maps to \x1a
  register u16 Y maps to \x1c
  register u16 Z maps to \x1e

  sfr { -- IN/OUT instructions.
    address := 0..\x3F
    type    := register
  } maps to \x20


  register u8  SPH  maps to \x5e -- stack H
  register u8  SPL  maps to \x5d -- stack L
  register u16 SP   maps to \x5d -- stack
```

```
  register u8 CCR maps to \x5f {
    C := slice{0} -- carry flag
    Z := slice{1} -- zero flag
    N := slice{2} -- neg flag
    V := slice{3} -- overflow flag
    S := slice{4} -- sign bit
    H := slice{5} -- half carry flag
    T := slice{6} -- Bit copy storage
    I := slice{7} -- global interrupt flag
  }
}

  void push(u8 val) {      -- post decrement
    sram.write8(SP, val)
    SP := SP-1
  }

  u8 pop() {
    u8 result
    SP := (u16)(SP+1)
    result := sram.read8(SP)
    return result
  }
}
```

Functional description of memory

```
component sram {
  memory ram {
    width   := 16  -- get 16 bits / access
    address := \x0..\x10FF
    type    := RAM

    GPR {
      width   := 16  -- get 16 bits / access
      address := 0..31
      stride  := 1
      type    := register
    }
    register u16 X maps to \x1a
    register u16 Y maps to \x1c
    r
    s
    }

    register u8  SPH  maps to \x5e -- stack H
    register u8  SPL  maps to \x5d -- stack L
    register u16 SP   maps to \x5d -- stack
```

user methods for specific behavior

```
    register u8 CCR maps to \x5f {
      C := slice{0} -- carry flag
      Z := slice{1} -- zero flag
      N := slice{2} -- neg flag
      V := slice{3} -- overflow flag
      S := slice{4} -- sign bit
      H := slice{5} -- half carry flag
      T := slice{6} -- Bit copy storage
      I := slice{7} -- global interrupt flag
    }
}
```

```
void push(u8 val) {     -- post decrement
  sram.write8(SP, val)
  SP := SP-1
}

u8 pop() {
  u8 result
  SP := (u16)(SP+1)
  result := sram.read8(SP)
  return result
}
}
```

Functional description of memory

- Ok to describe the HCS12 memory mapping:
  - Various Data Page pointers
  - Segmentation

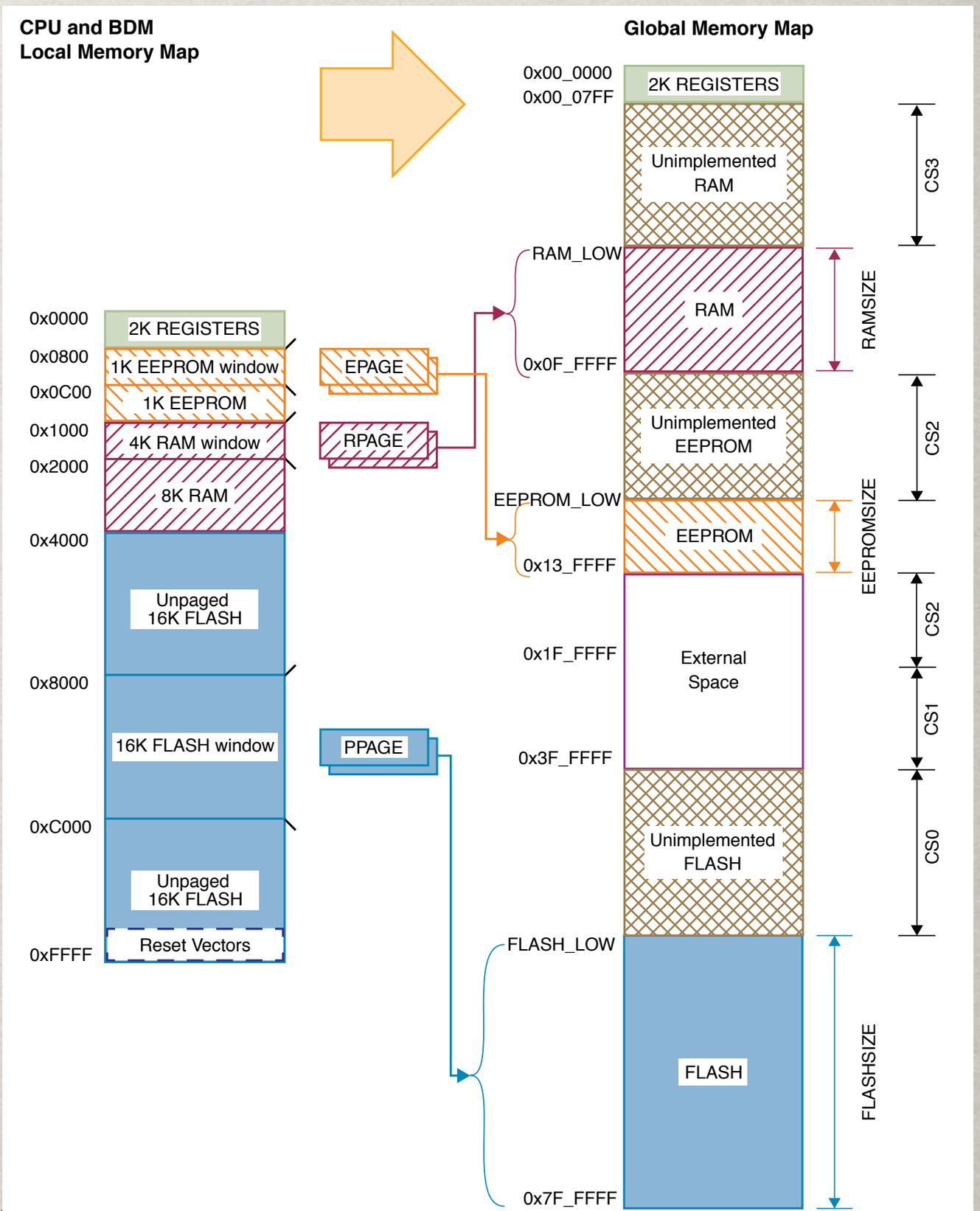


Figure 1-2. S12X CPU & BDM Global Address Mapping

Functional description of memory

- Ok to describe the HCS12 memory mapping:
  - Various Data Page pointers
  - Segmentation

- Drawback
  - static approach (changing the memory map leads to generate another simulator)

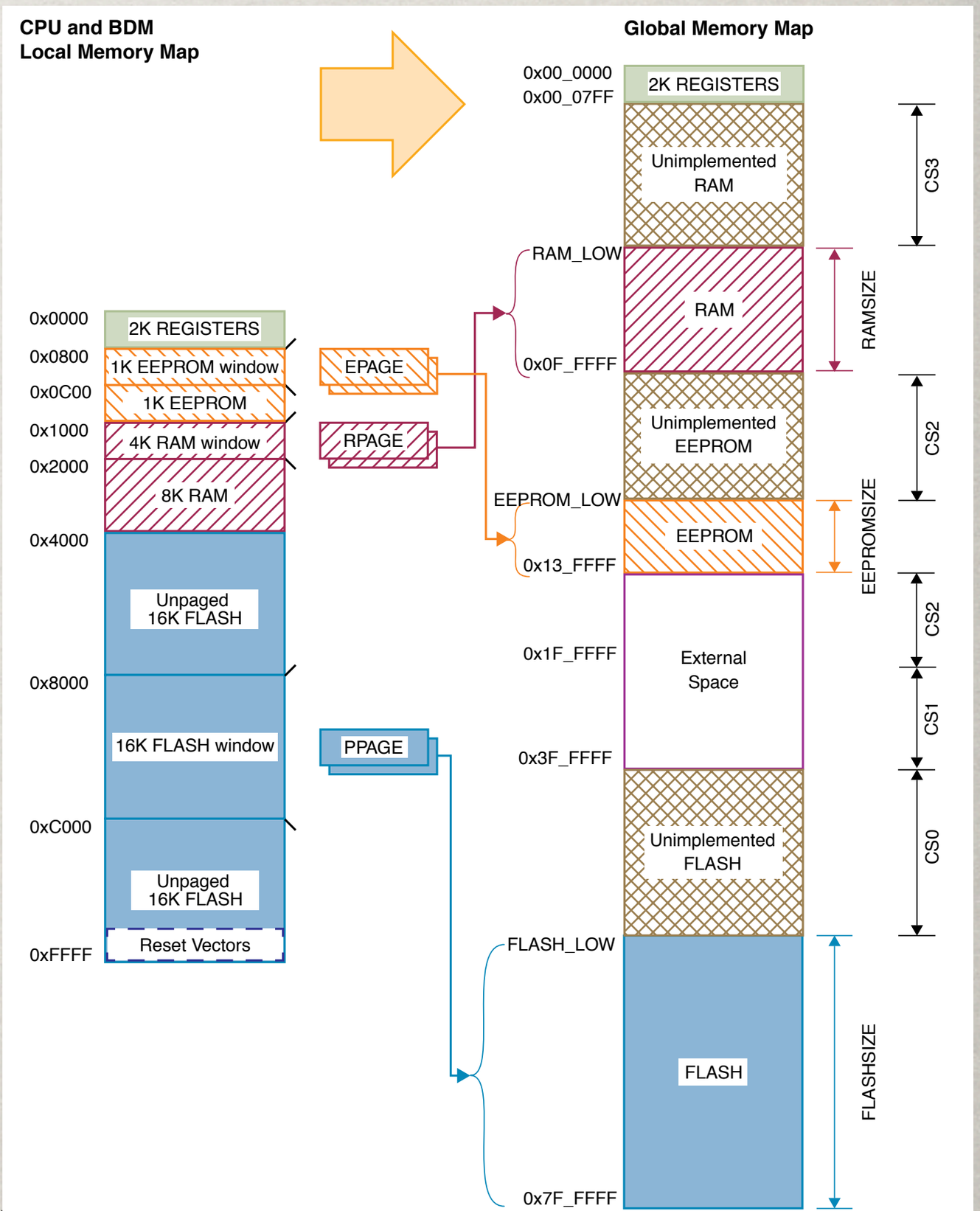


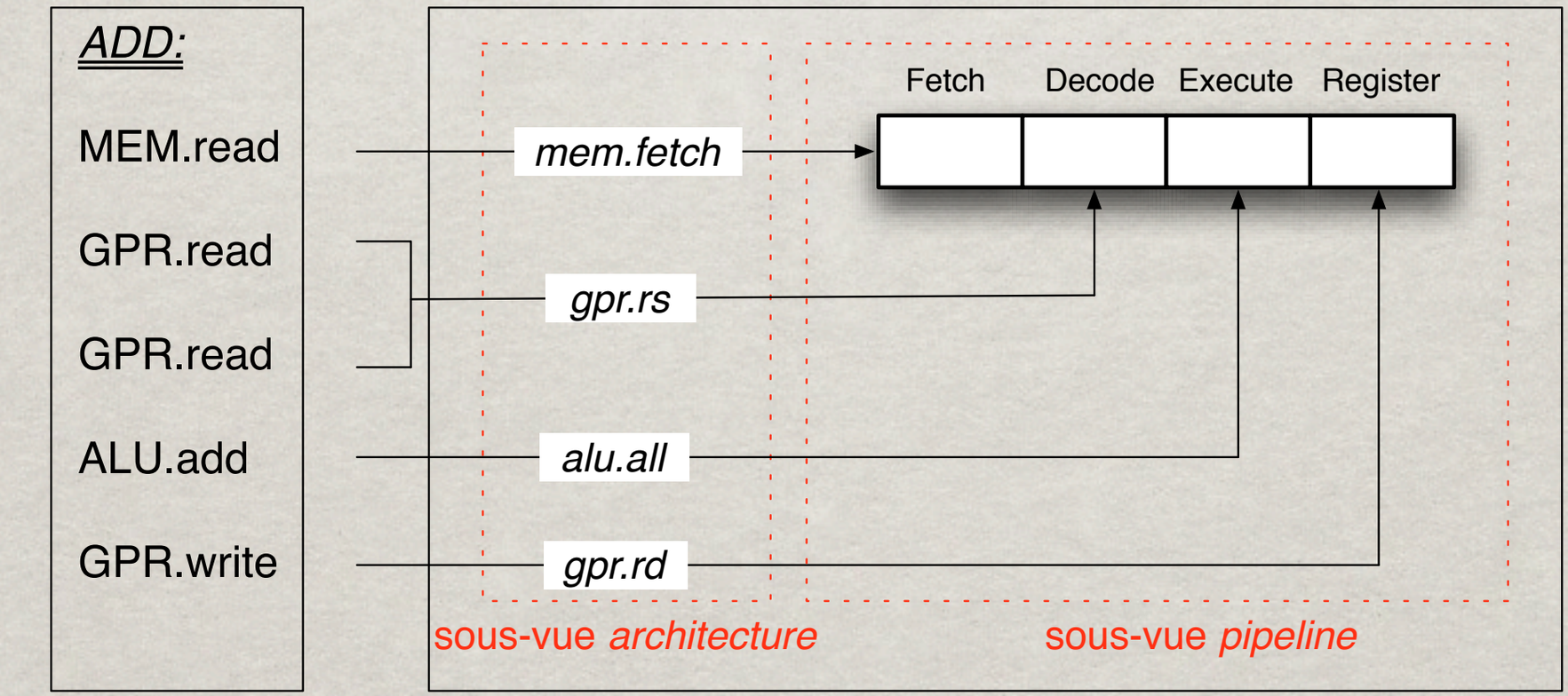
Figure 1-2. S12X CPU & BDM Global Address Mapping

Plan

- Bref rappel de la description du jeu d'instruction
- Modélisation fonctionnelle de la mémoire
- **description de la µ-architecture (nouvelle mouture)**
- pistes pour l'intégration de la hiérarchie mémoire

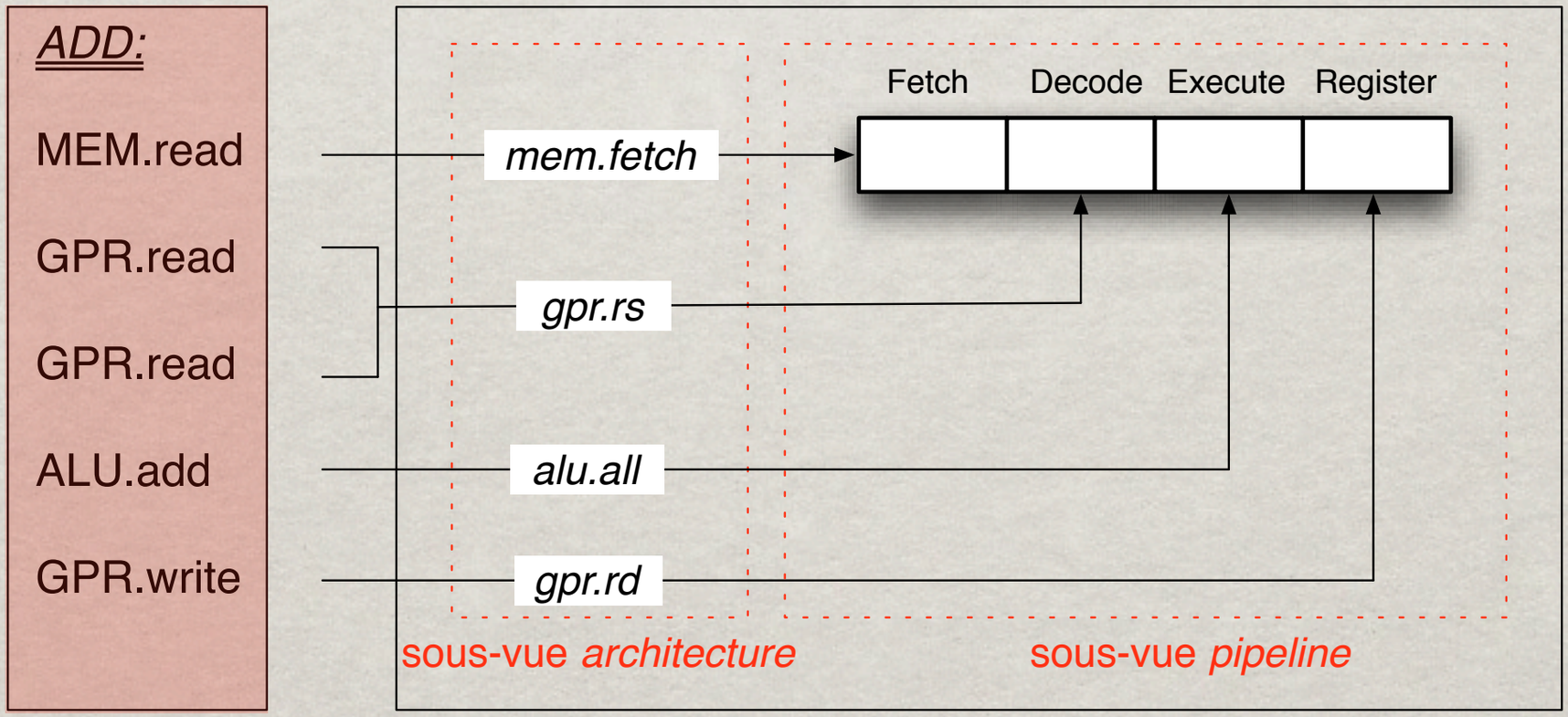
DESCRIPTION DE LA μ-ARCHI

- The ISA description call a component
  - There is no notion of **instance,** *i.e.* The add instruction uses the ALU, whenever there is 1 or more ALU in the processor.
  - **Defines what is needed to generate the ISS** (including functional memory)



DESCRIPTION DE LA μ-ARCHI

- The ISA description call a component
  - There is no notion of **instance,** *i.e.* The add instruction uses the ALU, whenever there is 1 or more ALU in the processor.
  - **Defines what is needed to generate the ISS** (including functional memory)

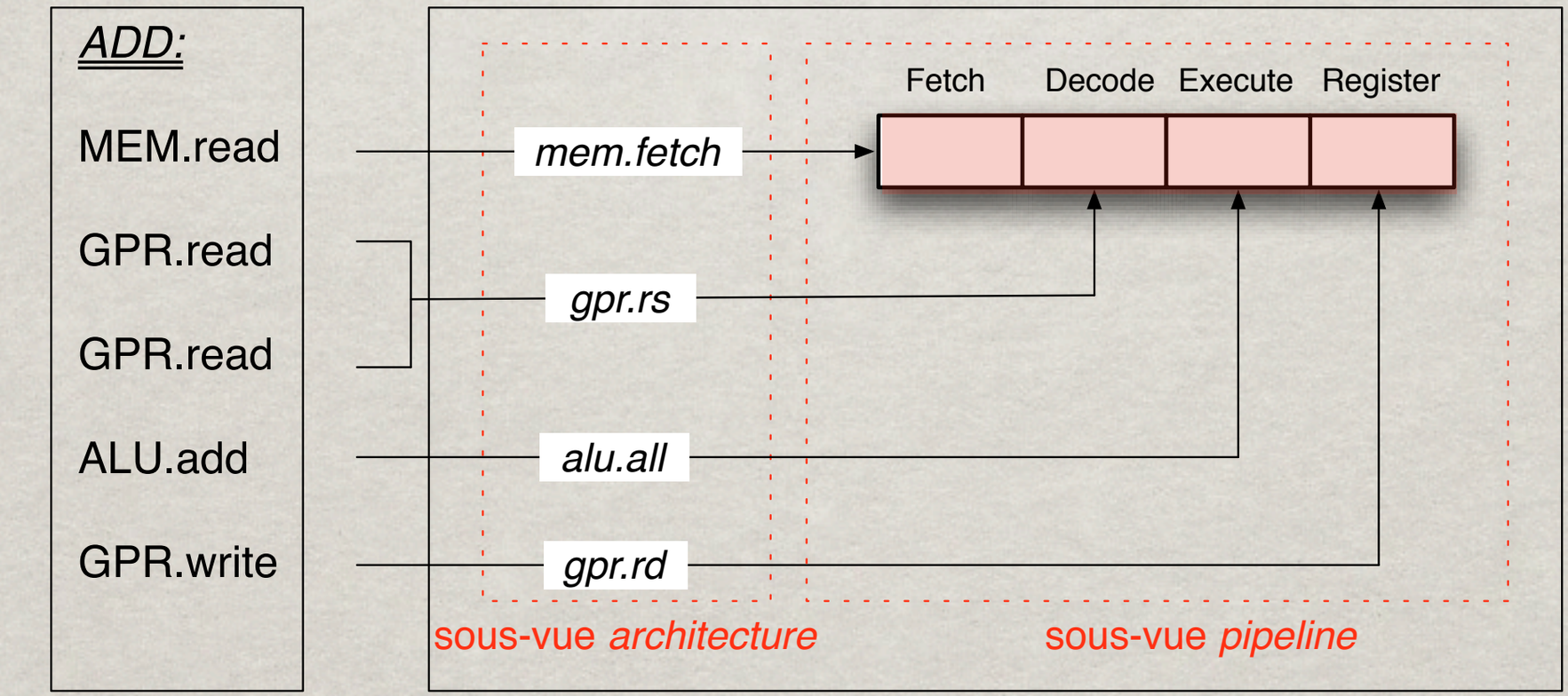


Vue sémantique

Vue micro-architecture

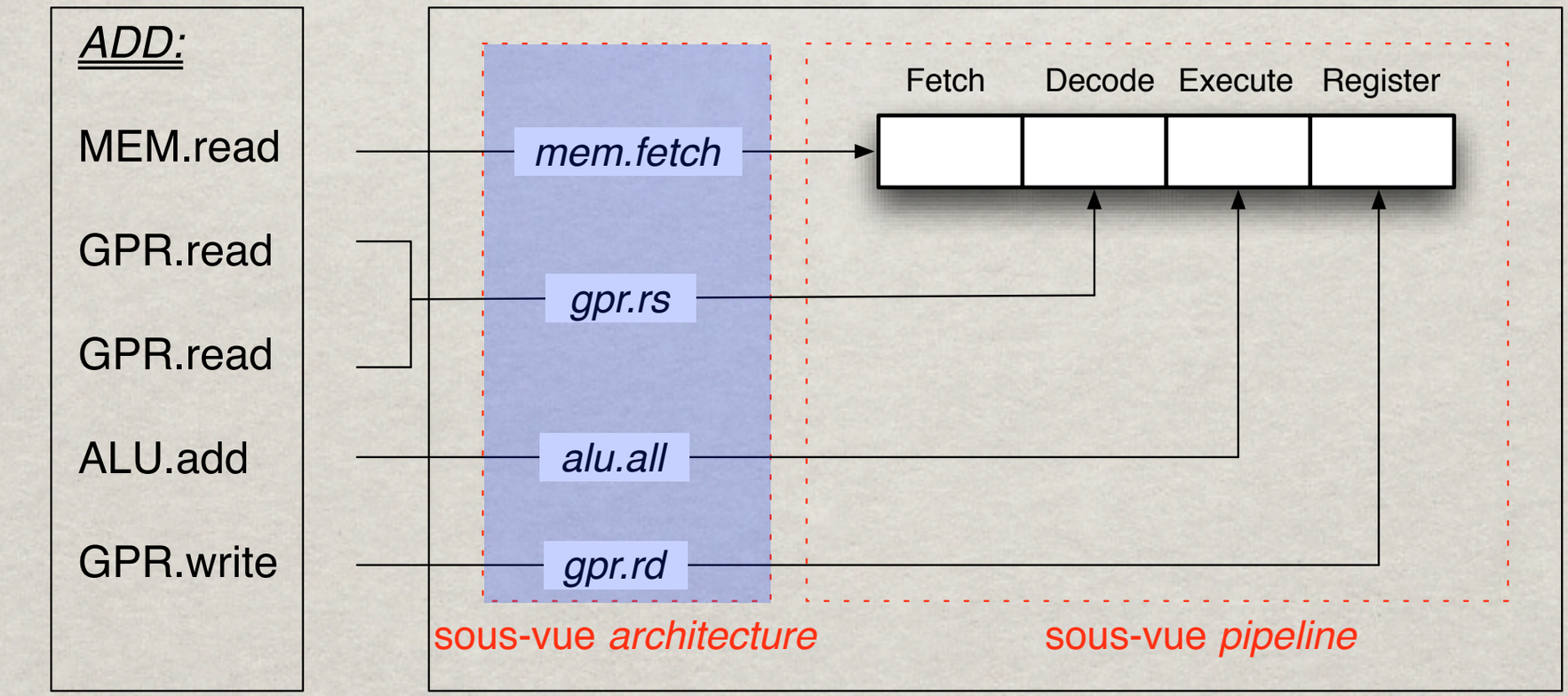
DESCRIPTION DE LA μ-ARCHI

- The ISA description call a component
  - There is no notion of **instance,** *i.e.* The add instruction uses the ALU, whenever there is 1 or more ALU in the processor.
  - **Defines what is needed to generate the ISS** (including functional memory)



DESCRIPTION DE LA μ-ARCHI

- The ISA description call a component
  - There is no notion of **instance,** *i.e.* The add instruction uses the ALU, whenever there is 1 or more ALU in the processor.
  - **Defines what is needed to generate the ISS** (including functional memory)



DESCRIPTION DE LA µ-ARCHI

- The DEVICE is the interface between the component and the pipeline
  - There is a notion of **instance** here.

```
component Integer_Unit {
 void updateStatus(u33 result){
  s32 tmp := (s32)(result{31..0})
  u32 testu34 := 0

   if tmp = 0s then
    CR.CR0 := 2 -- EQ
   elseif tmp > 0s then
    CR.CR0 := 4 -- GT
   else
    CR.CR0 := 8 -- LT
   end if
   CR.CR0{0} := XER.SO
 }
 ...
}
```

```
architecture Generic {
  device RegDev : SRU {
    read is GPR_read8 | GPR_read16 | GPR_read32 | spr_read
    write is GPR_write8 | GPR_write16 | GPR_write32 | spr_write
    port rs : read (3)
    port rd : write (2)
  }

  device IntDev : Integer_Unit {
    port all
  }

  device IntWithoutDiv : Integer_Unit {
    port all :except div_ov_signed,
                     div_ov_signed_withUpdateStatus,
                     div_ov_unsigned,
                     div_ov_unsigned_withUpdateStatus
  }
```

DESCRIPTION DE LA μ-ARCHI

- The DEVICE is the interface between the component and the pipeline
  - There is a notion of **instance** here.

```
component Integer_Unit {
 void updateStatus(u33 result){
  s32 tmp := (s32)(result{31..0})
  u32 testu34 := 0

   if tmp = 0s then
    CR.CR0 := 2 -- EQ
   elseif tmp > 0s then
    CR.CR0 := 4 -- GT
   else
    CR.CR0 := 8 -- LT
   end if
   CR.CR0{0} := XER.SO
 }
 ...
}
```

```
architecture Generic {
  device RegDev : SRU {
    read is GPR_read8 | GPR_read16 | GPR_read32 | spr_read
    write is GPR_write8 | GPR_write16 | GPR_write32 | spr_write
    port rs : read (3)
    port rd : write (2)
  }

  device IntDev : Integer_Unit {
    port all
  }

  device IntWithoutDiv : Integer_Unit {
    port all :except div_ov_signed,
                     div_ov_signed_withUpdateStatus,
                     div_ov_unsigned,
                     div_ov_unsigned_withUpdateStatus
```

2 integer units

Plan

- Bref rappel de la description du jeu d'instruction
- Modélisation fonctionnelle de la mémoire
- description de la µ-architecture (nouvelle mouture)
- **pistes pour l'intégration de la hiérarchie mémoire**

HIÉRARCHIE MÉMOIRE
Key points

- The memory hierarchy is split in two parts:
  - The **functional description**: The way the cache is organized
    - cache replacement policy;
    - data structure organization;
    - .. algorithmic parts!
  - The **timing description**: The way the time get in the party
    - how many cycles related to 1 cache miss?
    - how long to write to a memory location? (partially in // with the execution)
- We **keep the actual memory organization** (inside components)
  - use of actions
  - same description for the ISS generation

multicore pb?

HIÉRARCHIE MÉMOIRE
FUNCTIONAL DESCRIPTION

- The functional description should:
  - ...describe the functional behavior of the cache
  - allow the interaction with the ISA (cache lock, prefetch, config)
- We use the component approach, with some extensions:
  - tabular access
  - structures (like the `struct` of C)

HIÉRARCHIE MÉMOIRE
Timing description

- The timing description should give timings of hardware components
- Current approach is based on a textual description of timed automata:
  - Allow the use of Uppaal for formal verifications.
  - compatible with other internals models: allow compression.

HIÉRARCHIE MÉMOIRE
Example

- Example ICache ARM9TDMI:
  - 16KB
  - 512 lines of 32 bits (8 words)
  - 64-way associative (8 segs)
  - allocate-on-read-miss
  - Random or round robin policy
  - locking capability
- modèle mémoire RAM:
  - burst for next 3 accesses