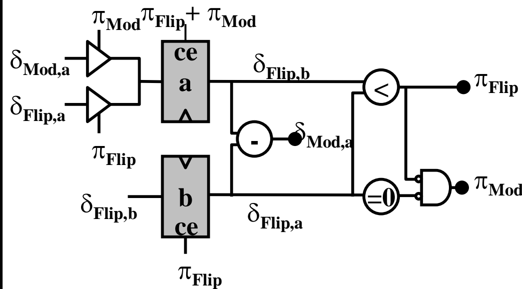
Modeling Computer Systems with Term Rewriting Systems

Krste Asanovic
Laboratory for Computer Science
Massachusetts Institute of Technology

http://www.csg.lcs.mit.edu/6.823

State-Centric Descriptions

Schematics

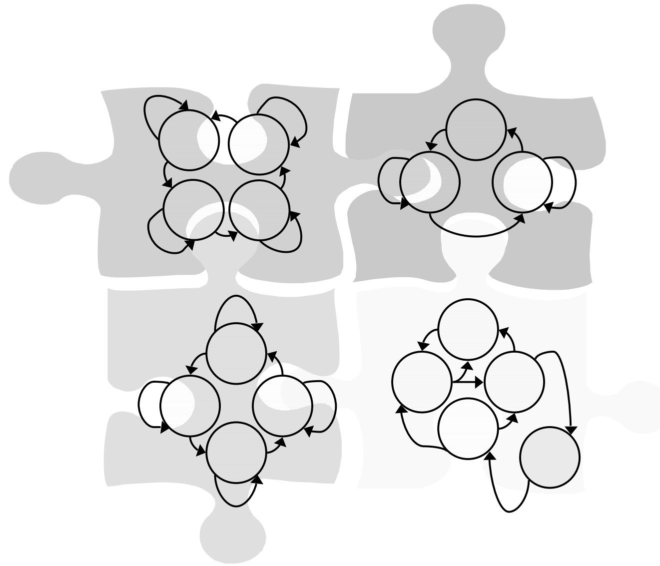


Hardware description languages

```
always @ (posedge Clk) begin
        if (a >= b) begin
                a <= a - b;
                b <= b;
        end else begin
                a <= b;
                b <= a;
        end
end
```

what does it describe?

State-centric Descriptions:
Cooperating Finite State Machines



Operation-Centric Descriptions

Microprocessor Manual

ADD rd, rs, rt

GPR[rd] ← GPR[rs] + GPR[rt]
PC ← PC + 4

TRS: A Novel "Language" to Describe Systems

System ≡ Structure + Behavior

hierarchically organized state (Structure) — *state transition rules* (Behavior)

Rule: **state$_1$ *if* predicate → state$_2$**

pattern on state — *action on state*

- ***Natural*** **for designers, especially to describe asynchronous behavior**
- ***Compact*** **descriptions**
- **Amenable to *verification***
- **Amenable to *synthesis***

GCD Example

Rules

Gcd(a, b) if b≠0 ⇒ Gcd(b, Rem(a, b))	*(Rule$_1$)*
Gcd(a, 0) ⇒ a	*(Rule$_2$)*
Rem(a, b) if a<b ⇒ a	*(Rule$_3$)*
Rem(a, b) if a≥b ⇒ Rem(a-b, b)	*(Rule$_4$)*

Execution:

Gcd(2,4) $\overset{R_1}{\Rightarrow}$ **Gcd(4,Rem(2,4))**
$\overset{R_3}{\Rightarrow}$ **Gcd(4,2)** $\overset{R_1}{\Rightarrow}$ **Gcd(2,Rem(4,2))**
$\overset{R_4}{\Rightarrow}$ **Gcd(2,Rem(2,2))** $\overset{R_4}{\Rightarrow}$ **Gcd(2,Rem(0,2))**
$\overset{R_3}{\Rightarrow}$ **Gcd(2,0)** $\overset{R_2}{\Rightarrow}$ **2**

Hardware description?

AX: A Minimalist RISC Instruction Set

Instruction	≡ r := Loadc(v)	*Load constant*
	|| r := Loadpc	*Load PC*
	|| r := Op(r_1, r_2)	*Arithmetic-Logic*
	|| Jz(r_c , r_a)	*Jump if $r_c = 0$*
	|| r := Load(r_a)	*Load*
	|| Store(r_a, r_v)	*Store*

'||' is a meta-linguistic symbol for identifying disjuncts

Programs are *reentrant* (not modifiable)

Abstract Data Types

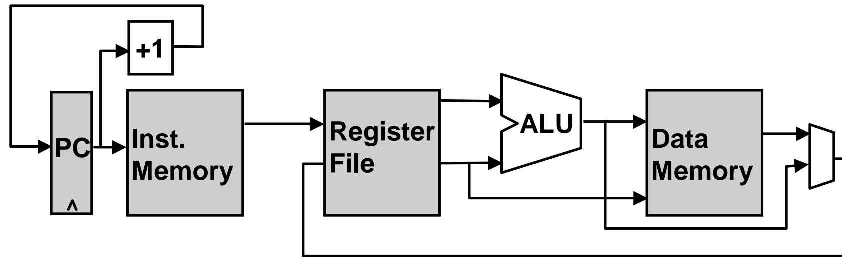
- **Arrays: if a is an array**
  - a[r] is the r'th entry of a
  - a[r:=v] sets the r'th entry of a to v

  Register Files, memories, etc. are represented using array data type

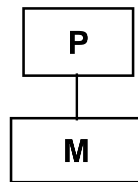
- **FIFO's: if q is a FIFO**
  - first(q) is the oldest entry in q
  - enq(x,q) enqueues a new entry x to q
  - deq(q) dequeues the oldest entry of q
  - clear(q) clears the contents of q

  Pipeline buffers are represented using FIFO data type

Non-pipelined AX Processor



P_B: Non-Pipelined Processor Model



S	≡	Sys(P, DM)
P	≡	Proc(PC, RF, IM)
PC	≡	Register of VAL
RF	≡	Array[rf-size] of VAL
DM	≡	Array[dm-size] of VAL
IM	≡	Array[im-size] of Instruction
VAL	≡	Bits[val-size]

Sys(Proc(pc, rf, im), dm)

Program counter — **Register file** — **Instruction memory** — **Data memory**

Programs are assumed to be non-modifiable

P_B: Non-pipelined Processor Rules

A simple rule can be given to describe the behavior of each instruction

Op rule

Proc(pc, rf, im) *if* **im[pc] = r:=Op(r_1, r_2)**
→ Proc(pc+1, rf[r:=v], im) *where* **v=<u>Op</u>(rf[r_1], rf[r_2])**

Jump rules

Proc(pc, rf, im) *if* **im[pc] = Jz(r_c , r_a) & rf[r_c]=0**
→ Proc(rf[r_a], rf, im)

Proc(pc, rf, im) *if* **im[pc] = Jz(r_c , r_a) & rf[r_c]≠0**
→ Proc(pc+1, rf, im)

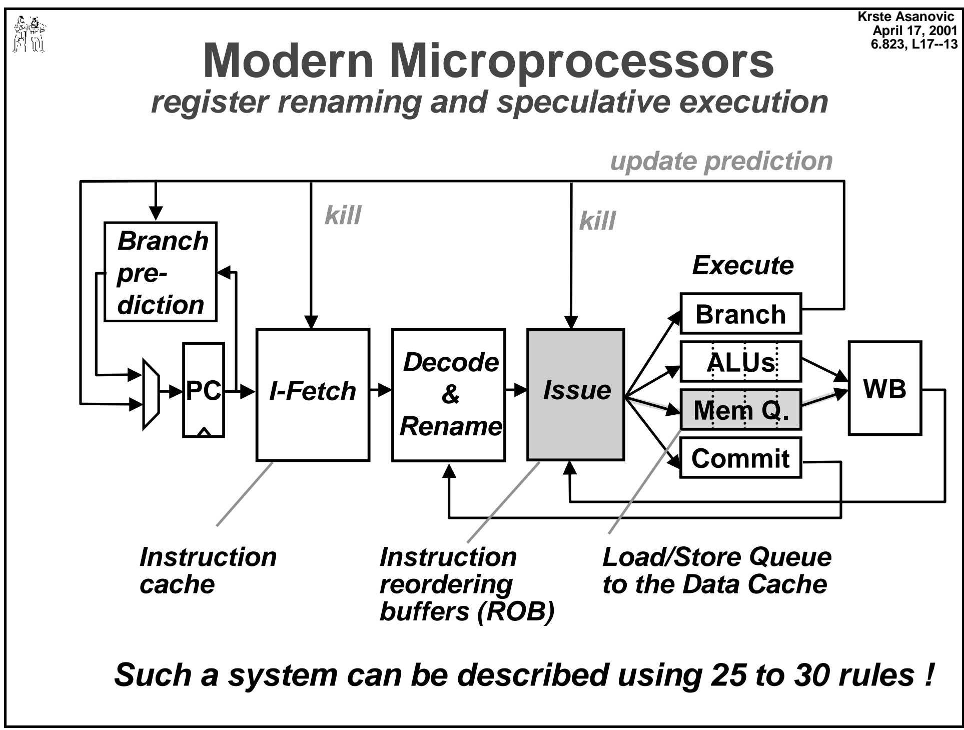
P_B : Load and Store Rules

Load rule

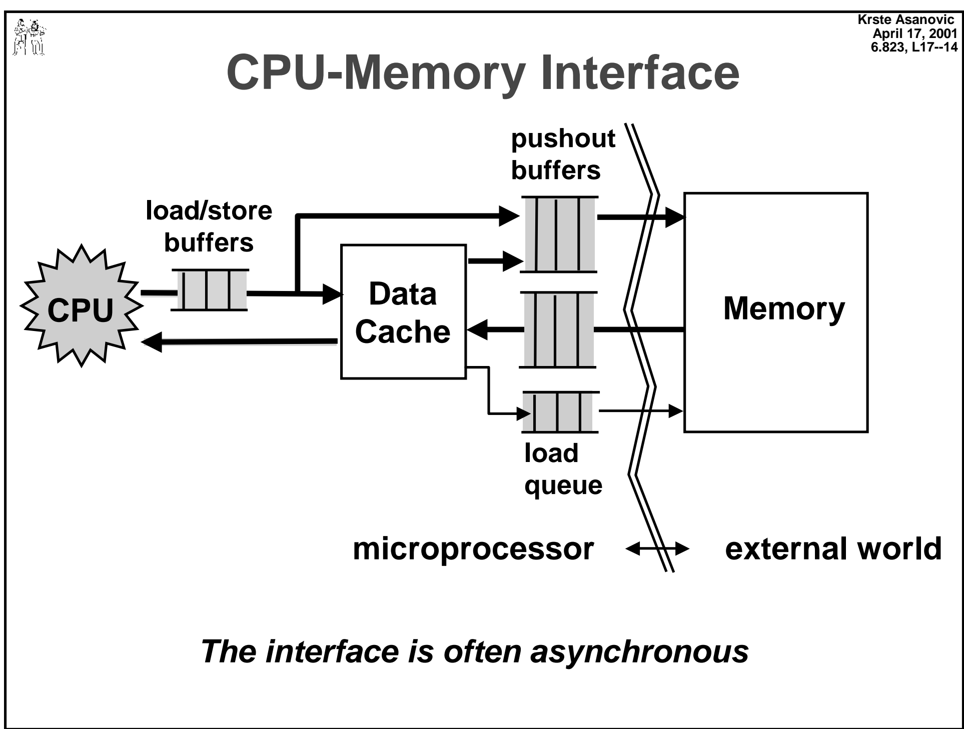
Sys(Proc(pc, rf, im), dm) *if* **im[pc] = r:=Load(r_a)**
→Sys(Proc(pc+1, rf[r:=dm[a]], im), dm)
where **a = rf[r_a]**

Store rule

Sys(Proc(pc, rf, im), dm) *if* **im[pc] = Store(r_a, r_v)**
→ Sys(Proc(pc+1, rf, im), dm[a:= rf[r_v]])
where **a = rf[r_a]**

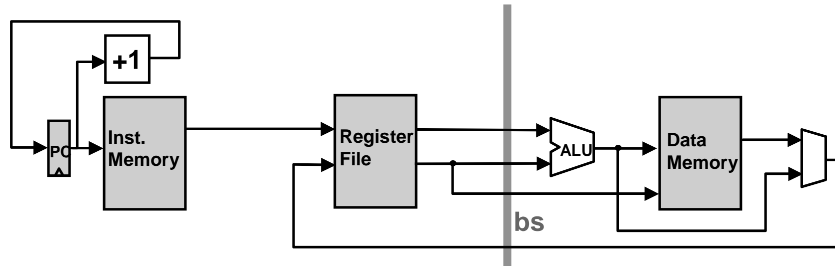


Krste Asanovic
April 17, 2001
6.823, L17--13
Modern Microprocessors
register renaming and speculative execution
update prediction
kill
kill
Branch pre-diction
Execute
Branch
ALUs
Mem Q.
Commit
PC
I-Fetch
Decode & Rename
Issue
WB
Instruction cache
Instruction reordering buffers (ROB)
Load/Store Queue to the Data Cache
Such a system can be described using 25 to 30 rules !



Krste Asanovic
April 17, 2001
6.823, L17--14
CPU-Memory Interface
pushout buffers
load/store buffers
CPU
Data Cache
Memory
load queue
microprocessor
external world
The interface is often asynchronous

Introducing a Pipeline Stage



All the state change implied by a rule conceptually happens atomically (*i.e.*, in one cycle)

Thus to pipeline one may want to split any rule that reads and writes the register file into multiple rules

⇒ *Introduce FIFO buffers (bs) to hold partially executed instructions*

Sys(Proc(pc, rf, bs, im), dm)

Splitting a Rule for Pipelining

Op rule

Proc(pc, rf, im) *if* im[pc] = $r{:=}Op(r_1, r_2)$
→ Proc(pc+1, rf[r:=v], im) *where* v=Op(rf[r_1], rf[r_2])

will take the following two steps:

Proc(pc, rf, bs, im) *if* **im[pc] = $r{:=}Op(r_1, r_2)$**
→ **Proc(pc+1, rf, bs;Itb(pc, r:=Op(rf[r_1], rf[r_2])), im)**

Proc(pc, rf, Itb(pc_1, r:=Op(v_1, v_2));bs, im)
→ **Proc(pc, rf[r:=v], bs, im)**
where **v= Op(v_1, v_2)**

Not quite correct!

";" represents ordered concatenation of elements

Op Rule for Two-Stage Pipeline

The Op-Fetch Rule:

Proc(pc, rf, bs, im) ***if*** **im[pc] = r:=Op(r_1, r_2)**
and **r1 ∉ Dest(bs)** ***and*** **r2 ∉ Dest(bs)**
→ Proc(pc+1, rf, bs;Itb(pc, r:=Op(rf[r_1], rf[r_2])), im)

The Op-Execute Rule:

Proc(pc, rf, Itb(pc_1, r:=Op(v_1, v_2));bs, im)
→ Proc(pc, rf[r:=v], bs, im)
where **v= <u>Op</u>(v_1, v_2)**

Let

Dest(inst) ≡ **destination register of instruction inst or instruction template**

Dest(bs) ≡ **the union of all destination registers of all instructions in buffer bs**

Splitting the Jump Rules

Jump rules

Proc(pc, rf, im) ***if*** **im[pc] = Jz(r_c , r_a) & rf[r_c]=0**
→ Proc(rf[r_a], rf, im)

Proc(pc, rf, im) ***if*** **im[pc] = Jz(r_c , r_a) & rf[r_c]≠0**
→ Proc(pc+1, rf, im)

are replaced by the following rules:

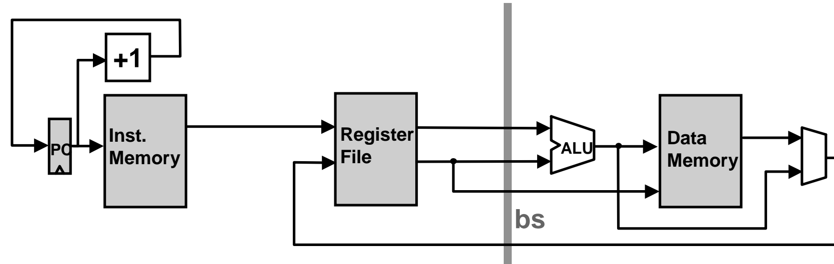
Proc(pc, rf, bs, im) ***if*** **im[pc] = Jz(r_c , r_a)**
and **r_c ∉ Dest(bs)** ***and*** **r_a ∉ Dest(bs)**
→ Proc(pc+1, rf, bs;Itb(pc, Jz(rf[r_c], rf[r_a])), im)

Proc(pc, rf, Itb(pc_1, Jz(v, npc));bs, im) ***if*** **v=0**
→ Proc(npc, rf, ε, im)

Proc(pc, rf, Itb(pc_1, Jz(v, npc));bs, im) ***if*** **v≠0**
→ Proc(pc, rf, bs, im)

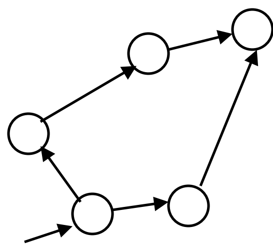
"ε" represents an empty FIFO

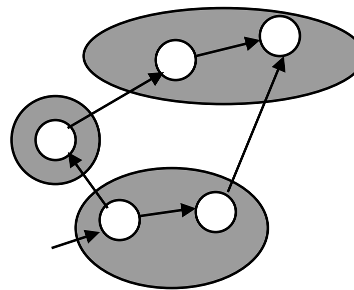
TRS Execution Model



- Model moves between valid states using any applicable rule $\Rightarrow$ ***non-deterministic FSM***
- Model can fire multiple times in first pipe stage before firing in second pipe stage
- Model only specifies allowable set of states, not a schedule for optimal execution
- Possible to develop hardware synthesis system that optimizes implementation to perform multiple state transitions per cycle

TRS Model ➔Implementation

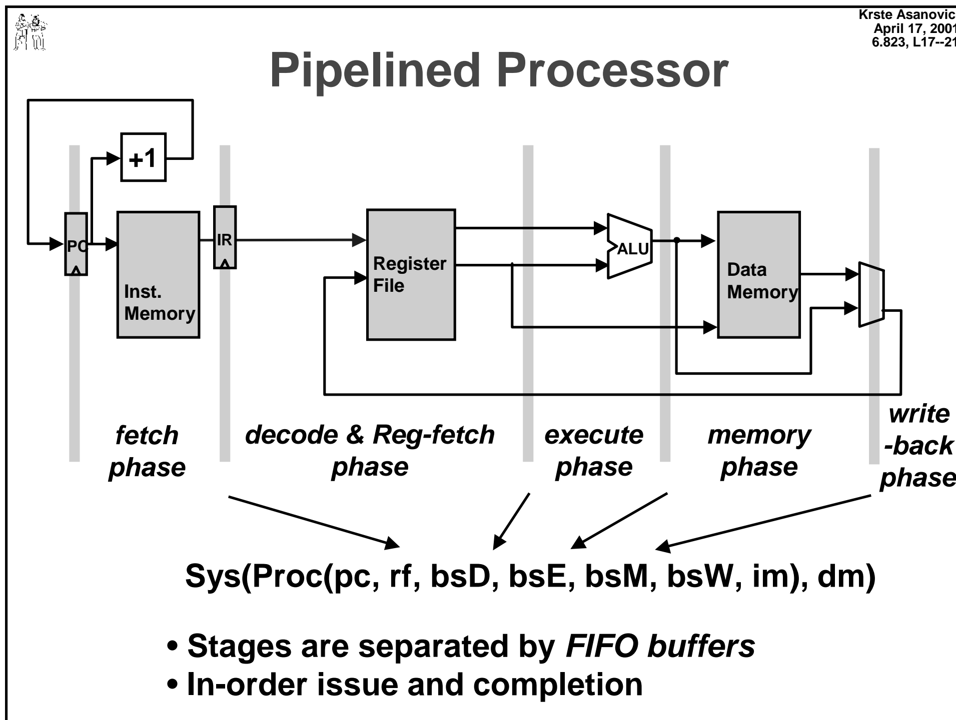


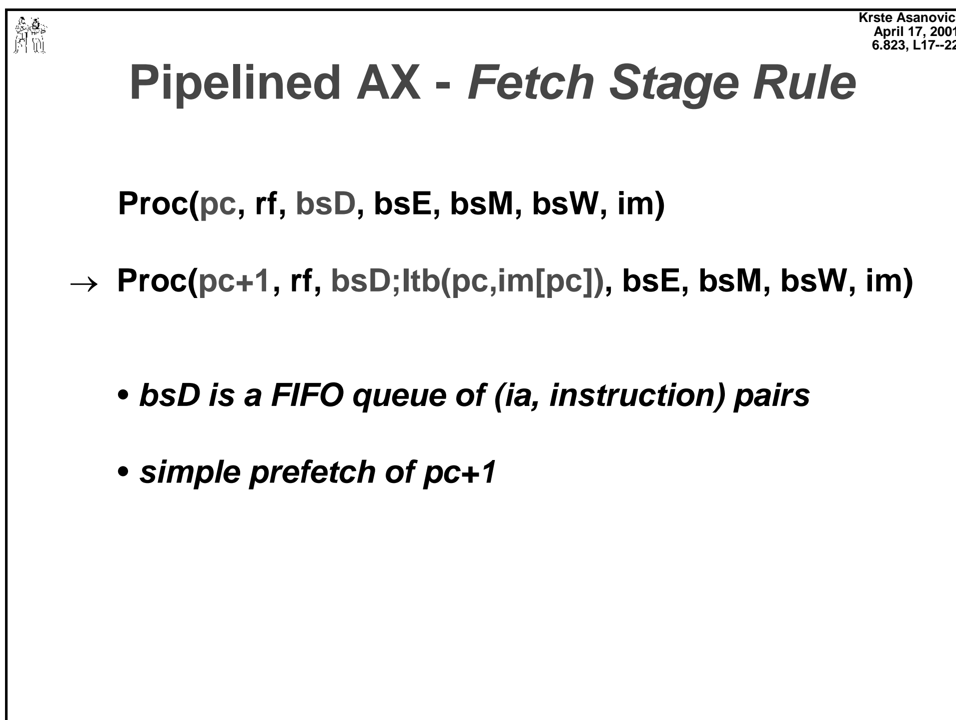


Define small atomic state changes in TRS

Perform multiple state changes in parallel in implementation

Concentrate on obtaining correct incremental state transitions in TRS model, then optimize to obtain fast parallel state machine


Krste Asanovic
April 17, 2001
6.823, L17--21
Pipelined Processor
+1
PC
IR
Inst.
Memory
Register
File
ALU
Data
Memory
fetch
phase
decode & Reg-fetch
phase
execute
phase
memory
phase
write
-back
phase
Sys(Proc(pc, rf, bsD, bsE, bsM, bsW, im), dm)
• Stages are separated by FIFO buffers
• In-order issue and completion


Krste Asanovic
April 17, 2001
6.823, L17--22
Pipelined AX - Fetch Stage Rule
Proc(pc, rf, bsD, bsE, bsM, bsW, im)
→ Proc(pc+1, rf, bsD;Itb(pc,im[pc]), bsE, bsM, bsW, im)
• bsD is a FIFO queue of (ia, instruction) pairs
• simple prefetch of pc+1

Hazard Detection

- ***Op decode rule***

Proc(pc, rf, Itb(ia, r:=Op(r1,r2));bsD, bsE, bsM, bsW, im)

if r1 $\notin$ Dest(bsE) *and* r2 $\notin$ Dest(bsE)
and r1 $\notin$ Dest(bsM) *and* r2 $\notin$ Dest(bsM)
and r1 $\notin$ Dest(bsW) *and* r2 $\notin$ Dest(bsW)

$\rightarrow$ **Proc(pc, rf, bsD, bsE;Itb(ia, r:=Op(rf[r1], rf[r2])), bsM, bsW, im)**

Let
Sources(inst) $\equiv$ source register(s) of instruction inst
Dests(a_1,a_2,...,a_n) $\equiv$ Dest(a_1) $\cup$ Dest(a_2) $\cup$ ... $\cup$ Dest(a_n)

NoHazard(inst, (bs_1, ... , bs_n)) $\equiv$
$\forall$s $\in$ Sources(inst), s $\notin$ Dests(bs_1, ... , bs_n)
Hazard(inst, (bs_1, ... , bs_n)) $\equiv$
$\exists$s $\in$ Sources(inst), s $\in$ Dests(bs , ... , bs)

Pipelined AX - *Decode Stage Rule*

bsE is a FIFO queue of (ia, instruction-template) pairs where a template is

r:= v || r:= Op(v_1,v_2) || r:= Load(a) || Jz(v_c,ia) || Store(a,v)

Proc(pc, rf, Itb(ia, $inst_1$);bsD, bsE, bsM, bsW, im)
if NoHazard($inst_1$,(bsE, bsM, bsW))
$\rightarrow$ **Proc(pc, rf, bsD, bsE;Itb(ia, it_1), bsM, bsW, im)**

it_1 is the template for $inst_1$, where the operands of $inst_1$ have been fetched from rf

Pipelined AX - *Execute Stage Rules-1*

No RAW hazards left

- ***Op execute rule***

  Proc(pc, rf, bsD, Itb(ia, r:=Op(v1,v2);bsE, bsM, bsW, im)

  → Proc(pc, rf, bsD, bsE, bsM;Itb(ia, r:=v), bsW, im)

  where **v=<u>Op</u>(v1,v2)**

- ***Jz execute rules***

  Proc(pc, rf, bsD, Itb(ia, r:=Jz(0,npc));bsE, bsM, bsW, im)

  → Proc(npc, rf, ε, ε, bsM, bsW, im)

  Proc(pc, rf, bsD, Itb(ia, r:=Jz(v,_));bsE, bsM, bsW, im)

  if **v ≠ 0**

  → Proc(pc, rf, bsD, bsE, bsM, bsW, im)

"_" signifies don't care in pattern match

Pipelined AX - *Execute Stage Rules-2*

- ***Copy execute rule***

  Proc(pc, rf, bsD, Itb(ia, it);bsE, bsM, bsW, im)

  if **it ≠ r:=Op(-,-)** ***or*** **it ≠ Jz(-,-)**

  → Proc(pc, rf, bsD, bsE, bsM;Itb(ia, it), bsW, im)

Pipelined AX - *Memory Stage Rules*

- ***Load memory rule***

  Sys(Proc(pc, rf, bsD, bsE, Itb(ia, r:=Load(a));bsM, bsW, im), dm)

  → Sys(Proc(pc, rf, bsD, bsE, bsM, bsW;Itb(ia, r:=v]), im), dm)   *where* v=dm[a]

- ***Store memory rule***

  Sys(Proc(pc, rf, bsD, bsE, Itb(ia, Store(a,v));bsM, bsW, im), dm)

  → Sys(Proc(pc, rf, bsD, bsE, bsM, bsW, im), dm[a:=v])

- ***Copy memory rule***

  Proc(pc, rf, bsD, bsE, Itb(ia, r:=v);bsM, bsW, im)

  → Proc(pc, rf, bsD, bsE, bsM, bsW;Itb(ia, r:=v), im)

Pipelined AX - *Writeback Stage Rule*

- ***Writeback rule***

  Proc(pc, rf, bsD, bsE, bsM, Itb(ia, r:=v));bsW, im)

  → Proc(pc, rf[r:=v], bsD, bsE, bsM, bsW, im)