ECE 154A Introduction to Computer Architecture
Fall 2012

Dmitri Strukov

Lecture 11-12-13
Pipelined design
MIPS Pipeline

Five stages, one step per stage
1. IF: Instruction fetch from memory
2. ID: Instruction decode & register read
3. EX: Execute operation or calculate address
4. MEM: Access memory operand
5. WB: Write result back to register
Why Pipeline? For Performance!

Once the pipeline is full, one instruction is completed every cycle, so CPI = 1.
Hazards

• Situations that prevent starting the next instruction in the next cycle
• Structure hazards
  – A required resource is busy
• Data hazard
  – Need to wait for previous instruction to complete its data read/write
• Control hazard
  – Deciding on control action depends on previous instruction
Structure Hazards

• Conflict for use of a resource
• In MIPS pipeline with a single memory
  – Load/store requires data access
  – Instruction fetch would have to *stall* for that cycle
    • Would cause a pipeline “bubble”
• Hence, pipelined datapaths require separate instruction/data memories
  – Or separate instruction/data caches
A Single Memory Would Be a Structural Hazard

Time (clock cycles)

Fix with separate instr and data memories (I$ and D$)
Data Hazards

- An instruction depends on completion of data access by a previous instruction
  
  - `add $s0, $t0, $t1`
  - `sub $t2, $s0, $t3`
Register Usage Can Cause Data Hazards

- Dependencies backward in time cause hazards

- add $1,
- sub $4,$1,$5
- and $6,$1,$7
- or $8,$1,$9
- xor $4,$1,$5

- Read before write data hazard
Register Usage Can Cause Data Hazards

- Dependencies backward in time cause hazards

- add $1,
- sub $4, $1, $5
- and $6, $1, $7
- or $8, $1, $9
- xor $4, $1, $5

- Read before write data hazard
Loads Can Cause Data Hazards

- Dependencies backward in time cause hazards

lw $1,4($2)
sub $4,$1,$5
and $6,$1,$7
or $8,$1,$9
xor $4,$1,$5

Load-use data hazard
How About Register File Access?

**Time (clock cycles)**

**Inst 1**
- `add $1, $2`
- IM
- Reg
- ALU
- DM
- Reg

**Inst 2**
- `add $2, $1`
- IM
- Reg
- ALU
- DM
- Reg

Fix register file access hazard by doing reads in the second half of the cycle and writes in the first half.

Clock edge that controls register writing:

Clock edge that controls loading of pipeline state registers:
One Way to “Fix” a Data Hazard

1. **add $1**
   - Stall
2. **sub $4,$1,$5**
   - Stall
3. **and $6,$1,$7**

Can fix data hazard by waiting – **stall** – but impacts CPI
Forwarding (aka Bypassing)

• Use result when it is computed
  – Don’t wait for it to be stored in a register
  – Requires extra connections in the datapath
Another Way to “Fix” a Data Hazard

Fix data hazards by forwarding results as soon as they are available to where they are needed.

- **add $1,**
- **sub $4,$1,$5**
- **and $6,$1,$7**
- **or $8,$1,$9**
- **xor $4,$1,$5**
Another Way to “Fix” a Data Hazard

Fix data hazards by forwarding results as soon as they are available to where they are needed.

- add $1,$
- sub $4,$1,$5
- and $6,$1,$7
- or $8,$1,$9
- xor $4,$1,$5
Forwarding Illustration

- **add** $1,$
- **sub** $4,$1,$5
- **and** $6,$7,$1

**EX forwarding**

**MEM forwarding**
Yet Another Complication!

- Another potential data hazard can occur when there is a conflict between the result of the WB stage instruction and the MEM stage instruction – which should be forwarded?

**Instruction Order**

- `add $1, $1, $2`
- `add $1, $1, $3`
- `add $1, $1, $4`
Yet Another Complication!

- Another potential data hazard can occur when there is a conflict between the result of the WB stage instruction and the MEM stage instruction – which should be forwarded?

```
<table>
<thead>
<tr>
<th>Instruction</th>
<th>ALU</th>
<th>DM</th>
<th>Reg</th>
</tr>
</thead>
<tbody>
<tr>
<td>add $1, $1, $2</td>
<td>IM</td>
<td>Reg</td>
<td>ALU</td>
</tr>
<tr>
<td>add $1, $1, $3</td>
<td>IM</td>
<td>Reg</td>
<td>ALU</td>
</tr>
<tr>
<td>add $1, $1, $4</td>
<td>IM</td>
<td>Reg</td>
<td>ALU</td>
</tr>
</tbody>
</table>
```
Load-Use Data Hazard

• Can’t always avoid stalls by forwarding
  – If value not computed when needed
  – Can’t forward backward in time!
Code Scheduling to Avoid Stalls

• Reorder code to avoid use of load result in the next instruction

• C code for $A = B + E; \ C = B + F$;

```
lw $t1, 0($t0)  # load $t1 with 0th element of array
lw $t2, 4($t0)   # load $t2 with 4th element of array
add $t3, $t1, $t2 # add $t1 and $t2, result in $t3
sw $t3, 12($t0)  # store $t3 at 12th address
```

```
lw $t1, 0($t0)  # load $t1 with 0th element of array
lw $t2, 4($t0)   # load $t2 with 4th element of array
lw $t4, 8($t0)   # load $t4 with 8th element of array
add $t3, $t1, $t2 # add $t1 and $t2, result in $t3
sw $t3, 12($t0)  # store $t3 at 12th address
```

```
lw $t1, 0($t0)  # load $t1 with 0th element of array
lw $t2, 4($t0)   # load $t2 with 4th element of array
lw $t4, 8($t0)   # load $t4 with 8th element of array
add $t3, $t1, $t2 # add $t1 and $t2, result in $t3
add $t5, $t1, $t4 # add $t1 and $t4, result in $t5
sw $t5, 16($t0)  # store $t5 at 16th address
```

13 cycles

11 cycles
MIPS Pipeline Control Path Modifications

- All control signals can be determined during Decode
  - and held in the state registers between pipeline stages
Pipeline Control

- IF Stage: read Instr Memory (always asserted) and write PC (on System Clock)
- ID Stage: no optional control signals to set

<table>
<thead>
<tr>
<th></th>
<th>EX Stage</th>
<th>MEM Stage</th>
<th>WB Stage</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Reg Dst</td>
<td>ALU Op1</td>
<td>ALU Op0</td>
</tr>
<tr>
<td>R</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>lw</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>sw</td>
<td>X</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>beq</td>
<td>X</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
Data Forwarding Control Conditions

1. EX Forward Unit:
   if (EX/MEM.RegWrite
   and (EX/MEM.RegisterRd != 0)
   and (EX/MEM.RegisterRd = ID/EX.RegisterRs))
       ForwardA = 10
   if (EX/MEM.RegWrite
   and (EX/MEM.RegisterRd != 0)
   and (EX/MEM.RegisterRd = ID/EX.RegisterRt))
       ForwardB = 10

2. MEM Forward Unit:
   if (MEM/WB.RegWrite
   and (MEM/WB.RegisterRd != 0)
   and (EX/MEM.RegisterRd != ID/EX.RegisterRs)
   and (MEM/WB.RegisterRd = ID/EX.RegisterRs))
       ForwardA = 01
   if (MEM/WB.RegWrite
   and (MEM/WB.RegisterRd != 0)
   and (EX/MEM.RegisterRd != ID/EX.RegisterRt)
   and (MEM/WB.RegisterRd = ID/EX.RegisterRt))
       ForwardB = 01

Forwards the result from the previous instr. to either input of the ALU
Forwards the result from the previous or second previous instr. to either input of the ALU
Datapath with Forwarding Hardware

Instruction Memory
- Read Address

Register File
- Read Addr 1
- Read Addr 2
- Write Addr
- Read Data 1
- Read Data 2
- Write Data

Sign Extend
- 16
- 32

ALU ctrl

Forward Unit

Address Read Data

Branch

Data Memory
- Address
- Read Data
- Write Data

Add

Shift left 2

ALU

EX/MEM

Control

ID/EX

IF/ID

Add

4

Add

PCSrc

Write Addr
Register

Write Data

Instruction Memory

Forwarding Hardware

Datapath
Datapath with Forwarding Hardware
Load-use Hazard Detection Unit

• Need a Hazard detection Unit in the ID stage that inserts a stall between the load and its use

1. ID Hazard detection Unit:
   if (ID/EX.MemRead
   and ((ID/EX.RegisterRt = IF/ID.RegisterRs)
   or  (ID/EX.RegisterRt = IF/ID.RegisterRt)))
   stall the pipeline

- The first line tests to see if the instruction now in the EX stage is a \texttt{lw}; the next two lines check to see if the destination register of the \texttt{lw} matches either source register of the instruction in the ID stage (the load-use instruction)

- After this one cycle stall, the forwarding logic can handle the remaining data hazards
Hazard/Stall Hardware

• Along with the Hazard Unit, we have to implement the stall

• Prevent the instructions in the IF and ID stages from progressing down the pipeline – done by preventing the PC register and the IF/ID pipeline register from changing
  – Hazard detection Unit controls the writing of the PC (PC.write) and IF/ID (IF/ID.write) registers

• Insert a “bubble” between the lw instruction (in the EX stage) and the load-use instruction (in the ID stage) (i.e., insert a noop in the execution stream)
  – Set the control bits in the EX, MEM, and WB control fields of the ID/EX pipeline register to 0 (noop). The Hazard Unit controls the mux that chooses between the real control values and the 0’s.

• Let the lw instruction and the instructions after it in the pipeline (before it in the code) proceed normally down the pipeline
Adding the Hazard/Stall Hardware

[Diagram of a computer processor lifecycle with labels for different stages like IF/ID, ID/EX, EX/MEM, MEM/WM, and control units like Add, Control, Hazard Unit, etc.]
Adding the Hazard/Stall Hardware

- **Instruction Memory**
  - Read Address
  - Add

- **Hazard Unit**
  - Control
  - ID/EX
  - ID/EX.MemRead

- **PCSrc**

- **Add**
  - 4 Shift left 2

- **IF/ID**

- **MEM/WB**
  - Branch
  - MEM/WE

- **Data Memory**
  - Address
  - Read Data
  - Write Data

- **Control Unit**
  - Forward Unit

- **Register**
  - Read Addr 1
  - Read Addr 2
  - Write Addr
  - Write Data

- **File**
  - Read Data 1
  - Read Data 2

- **ALU cntrl**
  - ALU

- **Forward Unit**

- **ID/EX. RegisterRt**

- **Shift left 2**

- **Write Data**
Control Hazards

• Branch determines flow of control
  – Fetching next instruction depends on branch outcome
  – Pipeline can’t always fetch correct instruction
    • Still working on ID stage of branch

• In MIPS pipeline
  – Need to compare registers and compute target early in the pipeline
  – Add hardware to do it in ID stage
Control Hazards

• When the flow of instruction addresses is not sequential (i.e., PC = PC + 4); incurred by change of flow instructions
  – Unconditional branches (j, jal, jr)
  – Conditional branches (beq, bne)
  – Exceptions

• Possible approaches
  – Stall (impacts CPI)
  – Move decision point as early in the pipeline as possible, thereby reducing the number of stall cycles
  – Delay decision (requires compiler support)
  – Predict and hope for the best!

• Control hazards occur less frequently than data hazards, but there is nothing as effective against control hazards as forwarding is for data hazards
Jumps Incur One Stall

- Jumps not decoded until ID, so one flush is needed
  - To flush, set IF.Flush to zero the instruction field of the IF/ID pipeline register (turning it into a noop)

- Fortunately, jumps are very infrequent – only 3% of the SPECint instruction mix
Supporting ID Stage Jumps

IF/ID
- Add
- Branch
- Control
- Shift left 2
- PCSrc

ID/EX
- Add
- Branch
- EX/MEM
- Forward Unit
- MEM/WB

Instruction Memory
- Read Address
- Write Addr
- Write Data

Register File
- Read Addr 1
- Read Addr 2
- Write Data

PC+4[31-28]

ALU cntrl
- ALU
- Read Data
- Sign Extend
- Write Data

Data Memory
- Address
- Read Data
Branch Instr’s Cause Control Hazards

- Dependencies backward in time cause hazards
One Way to “Fix” a Branch Control Hazard

Fix branch hazard by waiting – flush – but affects CPI
Another Way to “Fix” a Branch Control Hazard

• Move branch decision hardware back to as early in the pipeline as possible – i.e., during the decode cycle
Reducing the Delay of Branches

• Move the branch decision hardware back to the EX stage
  – Reduces the number of stall (flush) cycles to two
  – Adds an and gate and a 2x1 mux to the EX timing path
• Add hardware to compute the branch target address and evaluate the branch decision to the ID stage
  – Reduces the number of stall (flush) cycles to one
    (like with jumps)
    • But now need to add forwarding hardware in ID stage
  – Computing branch target address can be done in parallel with RegFile read (done for all instructions – only used when needed)
  – Comparing the registers can’t be done until after RegFile read, so comparing and updating the PC adds a mux, a comparator, and an and gate to the ID timing path
• For deeper pipelines, branch decision points can be even later in the pipeline, incurring more stalls
ID Branch Forwarding Issues

- MEM/WB “forwarding” is taken care of by the normal RegFile write before read operation

- Need to forward from the EX/MEM pipeline stage to the ID comparison hardware for cases like

```c
if (IDcontrol.Branch and (EX/MEM.RegisterRd != 0) and (EX/MEM.RegisterRd = IF/ID.RegisterRs))
    ForwardC = 1
if (IDcontrol.Branch and (EX/MEM.RegisterRd != 0) and (EX/MEM.RegisterRd = IF/ID.RegisterRt))
    ForwardD = 1
```

Forwards the result from the second previous instr. to either input of the compare
ID Branch Forwarding Issues, con’t

- If the instruction immediately before the branch produces one of the branch source operands, then a stall needs to be inserted (between the `beq` and `add1`) since the EX stage ALU operation is occurring at the same time as the ID stage branch compare operation.
  
  - “Bounce” the `beq` (in ID) and `next_seq_instr` (in IF) in place (ID Hazard Unit deasserts PC.Write and IF/ID.Write)
  - Insert a stall between the `add` in the EX stage and the `beq` in the ID stage by zeroing the control bits going into the ID/EX pipeline register (done by the ID Hazard Unit)

- If the branch is found to be taken, then flush the instruction currently in IF (`IF.Flush`)
Supporting ID Stage Branches
Delayed Branches

• If the branch hardware has been moved to the ID stage, then we can eliminate all branch stalls with *delayed branches* which are defined as always executing the next sequential instruction after the branch instruction – the branch takes effect *after* that next instruction
  – MIPS compiler moves an instruction to immediately after the branch that is not affected by the branch (a *safe* instruction) thereby *hiding* the branch delay

- With deeper pipelines, the branch delay grows requiring more than one delay slot
  - Delayed branches have lost popularity compared to more expensive but more flexible (dynamic) hardware branch prediction
  - Growth in available transistors has made hardware branch prediction relatively cheaper
Scheduling Branch Delay Slots

A. From before branch

```
add $1,$2,$3
if $2=0$ then

delay slot
```

becomes

```
if $2=0$ then
add $1,$2,$3
```

B. From branch target

```
sub $4,$5,$6

add $1,$2,$3
if $1=0$ then

delay slot
```

becomes

```
add $1,$2,$3
if $1=0$ then
sub $4,$5,$6
```

C. From fall through

```
add $1,$2,$3
if $1=0$ then

delay slot
```

becomes

```
sub $4,$5,$6
```

• A is the best choice, fills delay slot and reduces IC
• In B and C, the sub instruction may need to be copied, increasing IC
• In B and C, must be okay to execute sub when branch fails
Static Branch Prediction

- Resolve branch hazards by assuming a given outcome and proceeding without waiting to see the actual branch outcome

1. **Predict not taken** – always predict branches will *not be taken*, continue to fetch from the sequential instruction stream, only when branch *is* taken does the pipeline stall
   - If taken, flush instructions *after* the branch (earlier in the pipeline)
     - in IF, ID, and EX stages if branch logic in MEM – **three stalls**
     - In IF and ID stages if branch logic in EX – **two stalls**
     - in IF stage if branch logic in ID – **one stall**
   - ensure that those flushed instructions haven’t changed the machine state – automatic in the MIPS pipeline since machine state changing operations are at the tail end of the pipeline (MemWrite (in MEM) or RegWrite (in WB))
   - restart the pipeline at the branch destination
Flushing with Misprediction (Not Taken)

- To flush the IF stage instruction, assert IF. Flush to zero the instruction field of the IF/ID pipeline register (transforming it into a `noop`)
Flushing with Misprediction (Not Taken)

- To flush the IF stage instruction, assert \texttt{IF.Flush} to zero the instruction field of the IF/ID pipeline register (transforming it into a \texttt{noop})
Branching Structures

• Predict not taken works well for “top of the loop” branching structures

  ● But such loops have jumps at the bottom of the loop to return to the top of the loop – and incur the jump stall overhead

  ![Loop: beq $1,$2,Out](loop-beq.png)
  1st loop instr
  2nd loop instr
  1nd loop instr
  Loop
  j
  Out: fall out instr

osi Predict not taken doesn’t work well for “bottom of the loop” branching structures

  ![Loop: bne $1,$2,Loop](loop-bne.png)
  1st loop instr
  2nd loop instr
  last loop instr
  bne $1,$2,Loop
  fall out instr
Static Branch Prediction, con’t

- Resolve branch hazards by assuming a given outcome and proceeding

2. Predict taken – predict branches will always be taken
   - Predict taken *always* incurs one stall cycle (if branch destination hardware has been moved to the ID stage)
   - Is there a way to “cache” the address of the branch target instruction??

- As the branch penalty increases (for deeper pipelines), a simple static prediction scheme will hurt performance. With more hardware, it is possible to try to predict branch behavior dynamically during program execution

3. Dynamic branch prediction – predict branches at run-time using *run-time* information
Dynamic Branch Prediction

- A branch prediction buffer (aka branch history table (BHT)) in the IF stage addressed by the lower bits of the PC, contains bit(s) passed to the ID stage through the IF/ID pipeline register that tells whether the branch was taken the last time it was execute
  - Prediction bit may predict incorrectly (may be a wrong prediction for this branch this iteration or may be from a different branch with the same low order PC bits) but the doesn’t affect correctness, just performance
    - Branch decision occurs in the ID stage after determining that the fetched instruction is a branch and checking the prediction bit(s)
  - If the prediction is wrong, flush the incorrect instruction(s) in pipeline, restart the pipeline with the right instruction, and invert the prediction bit(s)
    - A 4096 bit BHT varies from 1% misprediction (nasa7, tomcatv) to 18% (eqntott)
Branch Target Buffer

- The BHT predicts *when* a branch is taken, but does not tell *where* its taken to!
  - A branch target buffer (BTB) in the IF stage caches the branch target address, but we also need to fetch the next sequential instruction. The prediction bit in IF/ID selects which “next” instruction will be loaded into IF/ID at the next clock edge.
    - Would need a two read port instruction memory

- Or the BTB can cache the branch taken instruction while the instruction memory is fetching the next sequential instruction

- If the prediction is correct, stalls can be avoided no matter which direction they go
1-bit Prediction Accuracy

- A 1-bit predictor will be incorrect twice when not taken

- Assume predict_bit = 0 to start (indicating branch not taken) and loop control is at the bottom of the loop code

1. First time through the loop, the predictor mispredicts the branch since the branch is taken back to the top of the loop; invert prediction bit (predict_bit = 1)

2. As long as branch is taken (looping), prediction is correct

3. Exiting the loop, the predictor again mispredicts the branch since this time the branch is not taken falling out of the loop; invert prediction bit (predict_bit = 0)

- For 10 times through the loop we have a 80% prediction accuracy for a branch that is taken 90% of the time
2-bit Predictors

- A 2-bit scheme can give 90% accuracy since a prediction must be wrong twice before the prediction bit is changed.

Loop:
1st loop instr
2nd loop instr
...
last loop instr
bne $1,$2,Loop
fall out instr
2-bit Predictors

- A 2-bit scheme can give 90% accuracy since a prediction must be wrong twice before the prediction bit is changed.

**right 9 times**

Loop: 1\(^{st}\) loop instr
2\(^{nd}\) loop instr
.. 
.. 
last loop instr
bne $1,$2,Loop
fall out instr

BHT also stores the initial FSM state.
Dealing with Exceptions

• Exceptions (aka interrupts) are just another form of control hazard. Exceptions arise from
  – R-type arithmetic overflow
  – Trying to execute an undefined instruction
  – An I/O device request
  – An OS service request (e.g., a page fault, TLB exception)
  – A hardware malfunction

• The pipeline has to stop executing the offending instruction in midstream, let all prior instructions complete, flush all following instructions, set a register to show the cause of the exception, save the address of the offending instruction, and then jump to a prearranged address (the address of the exception handler code)

• The software (OS) looks at the cause of the exception and “deals” with it
Two Types of Exceptions

• Interrupts – asynchronous to program execution
  – caused by external events
  – may be handled between instructions, so can let the instructions currently active in the pipeline complete before passing control to the OS interrupt handler
  – simply suspend and resume user program

• Traps (Exception) – synchronous to program execution
  – caused by internal events
  – condition must be remedied by the trap handler for that instruction, so much stop the offending instruction midstream in the pipeline and pass control to the OS trap handler
  – the offending instruction may be retried (or simulated by the OS) and the program may continue or it may be aborted
Where in the Pipeline Exceptions Occur

- Arithmetic overflow
- Undefined instruction
- TLB or page fault
- I/O service request
- Hardware malfunction
Where in the Pipeline Exceptions Occur

<table>
<thead>
<tr>
<th>Exception</th>
<th>Stage(s)?</th>
<th>Synchronous?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arithmetic overflow</td>
<td>EX</td>
<td>yes</td>
</tr>
<tr>
<td>Undefined instruction</td>
<td>ID</td>
<td>yes</td>
</tr>
<tr>
<td>TLB or page fault</td>
<td>IF, MEM</td>
<td>yes</td>
</tr>
<tr>
<td>I/O service request</td>
<td>any</td>
<td>no</td>
</tr>
<tr>
<td>Hardware malfunction</td>
<td>any</td>
<td>no</td>
</tr>
</tbody>
</table>

- Beware that multiple exceptions can occur simultaneously in a *single* clock cycle.
Multiple Simultaneous Exceptions

- Hardware sorts the exceptions so that the earliest instruction is the one interrupted first.
Multiple Simultaneous Exceptions

- Hardware sorts the exceptions so that the earliest instruction is the one interrupted first.
Additions to MIPS to Handle Exceptions (Fig 6.42)

• Cause register (records exceptions) – hardware to record in Cause the exceptions and a signal to control writes to it (CauseWrite)

• EPC register (records the addresses of the offending instructions) – hardware to record in EPC the address of the offending instruction and a signal to control writes to it (EPCWrite)
  – Exception software must match exception to instruction

• A way to load the PC with the address of the exception handler
  – Expand the PC input mux where the new input is hardwired to the exception handler address - (e.g., 8000 0180$_{\text{hex}}$ for arithmetic overflow)

• A way to flush offending instruction and the ones that follow it
Datapath with Controls for Exceptions

- **Instruction Memory:**
  - Read Address

- **Data Memory:**
  - Read Data Address
  - Write Data

- **ALU:**
  - Read Data
  - Write Data

- **RegFile:**
  - Read Addr 1
  - Read Addr 2
  - Read Data 1
  - Write Addr
  - ReadData 2
  - Write Data

- **Forward Unit:**
  - Sign Extend
  - 16
  - 32

- **Control Unit:**
  - Add
  - Shift left 2

- **Hazard Unit:**
  - ID.Flush
  - Branch

- **ID/EX Unit:**
  - PCSrc

- **EX/MEM Unit:**
  - EX.Flush
  - ID.Flush

- **MEM/WB Unit:**
  - Write Addr
  - Read Data 2

- **Branch Unit:**
  - 8000 0180<sub>hex</sub>

- **Cause Unit:**
  - EPC

- **Forward Unit 2:**
  - Sign Extend

- **ALU cntrl:**
  - Forward Unit

Diagram shows flow of data and control signals through various units.
Other Sample Pipeline Alternatives

- **ARM7**
  
  ![ARM7 Diagram]
  
  PC update IM access  
  decode reg access  
  ALU op DM access  
  shift/rotate commit result (write back)

- **XScale**
  
  ![XScale Diagram]
  
  PC update BTB access start IM access  
  decode reg 1 access  
  shift/rotate reg 2 access  
  start DM access exception  
  DM write reg write