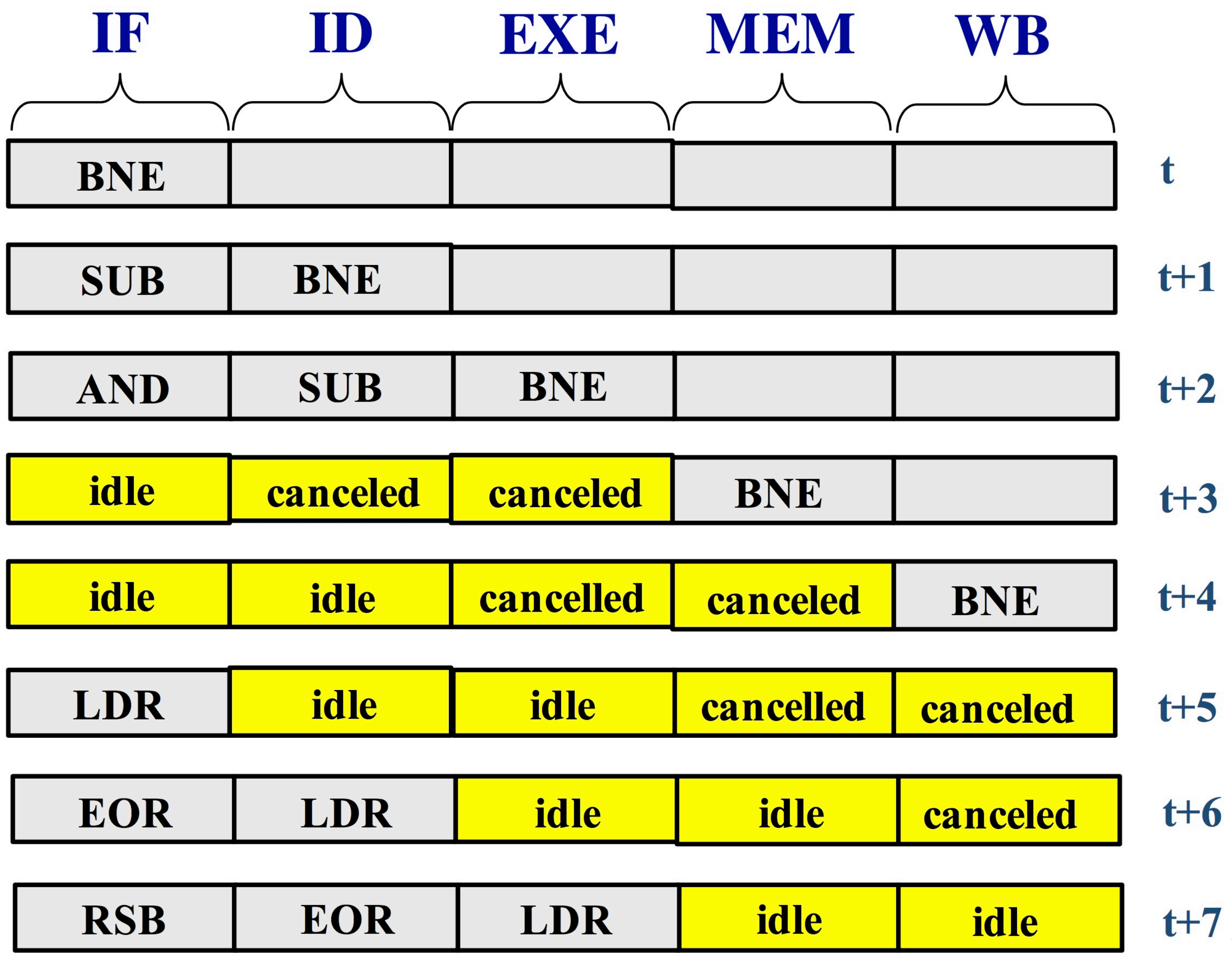
Pipelined Instruction Execution



	IF	ID	EXE	MEM	WB
t	BNE				
t+1	SUB	BNE			
t+2	AND	SUB	BNE		
t+3	idle	canceled	canceled	BNE	
t+4	idle	idle	cancelled	canceled	BNE
t+5	LDR	idle	idle	cancelled	canceled
t+6	EOR	LDR	idle	idle	canceled
t+7	RSB	EOR	LDR	idle	idle

In this Lesson

- Principles of pipelined instruction execution
- Pipeline hazards
- Solutions for hazards
  - *Data forwarding*
  - *Loop Buffer*
  - *Delayed branches*
  - *Branch prediction*
  - *Multiple prefetching*
- Performance with pipelined instruction execution units

Pipelined Instruction Execution
Part I: Pipelines and Hazards

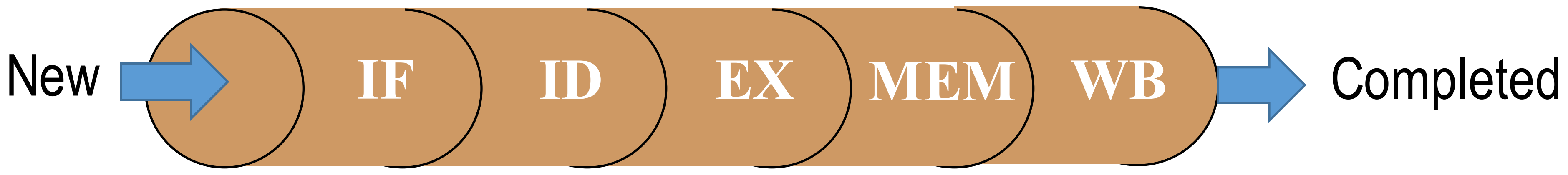
IF	ID	EXE	MEM	WB	
BNE					t
SUB	BNE				t+1
AND	SUB	BNE			t+2
idle	canceled	canceled	BNE		t+3
idle	idle	cancelled	canceled	BNE	t+4
LDR	idle	idle	cancelled	canceled	t+5
EOR	LDR	idle	idle	canceled	t+6
RSB	EOR	LDR	idle	idle	t+7

Execution Without Pipeline Stages

- The execution unit takes one instruction at a time.
- If an instruction takes **n** cycles to execute, **m** instructions take **n** x **m** cycles to complete.

Pipelining

- Pipelining is executing instructions by stages like a production line



- Ideally, the instructions pass from one stage to the other on every cycle and are executed at a rate of one per cycle (**m** instructions take **m** cycles to complete)

Pipeline of Five Stages

- ***Instruction Fetch* (IF)**
  - *Brings de instruction from memory into the CPU.*
- ***Instruction decode/register fetch* (ID)**
  - *Decodes the instruction,*
  - *Reads the content of the registers specified as operands into temporary registers*
  - *Sign-extends the immediate values,*
  - *Determines the target address of branch instructions*
- ***Execution/effective address* (EX)**
  - *Load/stores – calculates the effective addresss*
  - *Arithmetic/logic – executes the operation*

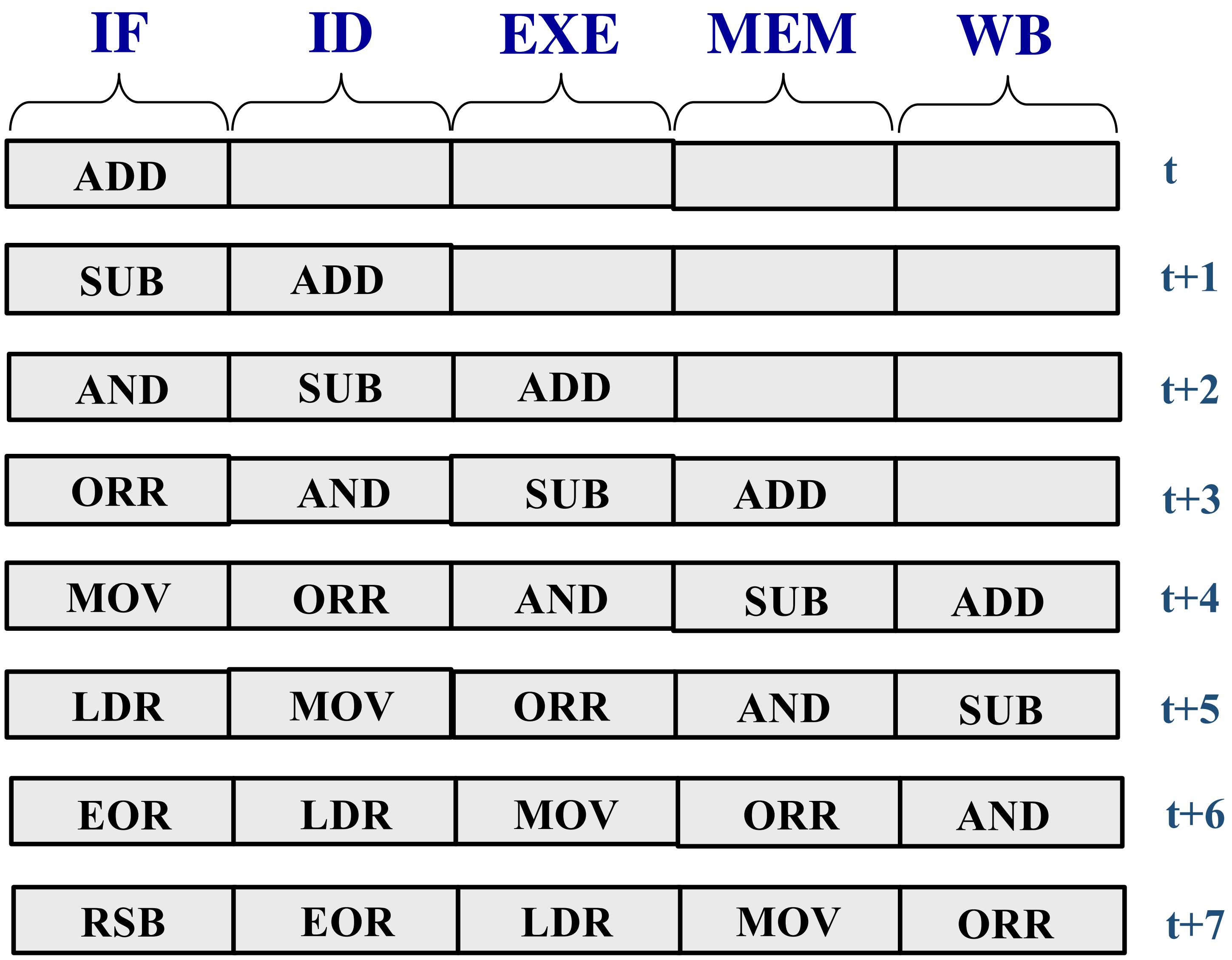
Pipeline of Five Stages

- ***Memory access* (MEM)**
  - *Load – reads the effective address*
  - *Store – writes the effective address*
  - *Any other instruction – no action*
- ***Write-back* (WB)**
  - *Writes the result in the destination register if it is an arithmetic/logic or a load instruction*

Ideal Execution of Instructions on a Pipelined Unit

ARM code:

```
ADD  R2, R3, R1
SUB  R12, R6, R10
AND  R5,  R8, R1
ORR  R9, R3, R10
MOV  R11, R3
LDR  R5, [R4 ,R1]
EOR  R12, R1, R6
RSB  R2, R10, R9
```



IF	ID	EXE	MEM	WB	
ADD					t
SUB	ADD				t+1
AND	SUB	ADD			t+2
ORR	AND	SUB	ADD		t+3
MOV	ORR	AND	SUB	ADD	t+4
LDR	MOV	ORR	AND	SUB	t+5
EOR	LDR	MOV	ORR	AND	t+6
RSB	EOR	LDR	MOV	ORR	t+7

Pipeline Hazards (Interlocks)

- Are the main reason why pipelines do not exhibit ideal performance.
- Are basically instructions interdependencies that have the potential to produce an incorrect outcome of a program.
- May cause instructions to stall in the pipelined until the hazard is resolved
- Have a negative effect on the performance of the processor due to the wasted idle cycles caused by the stalling instructions.

Types of Hazards

- Structural hazards
- Data Hazards
- Control Hazards

Structural Hazards

- Caused by more than one instructions attempting to access the same resource (memory for example) at the same time.
- In a single processor system it happens when a load/store instruction is in the **MEM** stage and the processors attempts to fetch another instruction in the **IF** stage.
- In a multiprocessor system the hazard could arise from two or more processors trying to access memory at the same time.
- To prevent this hazard the load/store instruction in the **MEM** stage is allowed to proceed and the instruction that was to be fetched is prevented from entering the **IF** stage.
- A structural hazard is present any time a load/store instruction is in the **MEM** stage.

Example of Structural Hazard for Single Port Memory

ARM code:

```
LDR  R2, [R3, R1]
SUB  R12, R6, R10
AND  R5,  R8, R1
ORR  R9, R3, R10
MOV  R11, R3
LDR  R5, [R4 ,R1]
EOR  R12, R1, R6
RSB  R2, R10, R8
```

IF	ID	EXE	MEM	WB	
LDR					t
SUB	LDR				t+1
AND	SUB	LDR			t+2
idle	AND	SUB	LDR		t+3
ORR	idle	AND	SUB	LDR	t+4
MOV	ORR	idle	AND	SUB	t+5
LDR	MOV	ORR	idle	AND	t+6
EOR	LDR	MOV	ORR	idle	t+7

Solutions for Structural Hazards

- Using memory systems with more than one memory port (at least one for instructions and one for data).
- Using split caches (one cache for instructions and another for data)
- Both eliminate the structural hazard penalty of one cycle.

Data Hazards

- Occur when data reads and writes could take place in an order different from the order prescribed by the program

- This could happen due to the structure of the pipeline stages or in processors that allows out-of order instruction execution

- There are three types of data hazards:
  - *Read after Write (RAW)*
  - *Write after Read (WAR)*
  - *Write after Write (WAW)*

RAW Hazard

- Arises when an instruction attempts to read the content of a source register that is the destination register of a previous instruction that has not completed the writing of its destination register.

```
ADD   R2, R3, R1
EOR   R7, R9, R2
```

- In a pipeline of five stages this hazard is detected when the EOR instruction attempts to enter the **ID** stage while the ADD instruction has not left the **WB** stage.

- A simple way to prevent this hazard is to halt the EOR instruction in the **IF** stage until the ADD instruction leaves the **WB** stage.

Example of RAW Hazards

ARM code:

```
ADD  R2, R3, R1
EOR  R7, R0, R2
AND  R5,  R8, R7
ORR  R9, R3, R4
MOV  R11, R3
SUB  R12, R1, R6
RSB  R3, R10, R8
LDR  R0, [R4 ,R1]
```

IF	ID	EXE	MEM	WB	
ADD					t
EOR	ADD				t+1
EOR	idle	ADD			t+2
EOR	idle	idle	ADD		t+3
EOR	idle	idle	idle	ADD	t+4
AND	EOR	idle	idle	idle	t+5
AND	idle	EOR	idle	idle	t+6
AND	idle	idle	EOR	idle	t+7

WAR Hazard

- Arises when an instruction attempts to write a register that is a source register of a previous instruction that has not completed the reading of its source registers.

```
ADD  R2, R3, R7
EOR  R7, R9, R2
```

- Can only happen in processors that allow out-of-order instruction execution.

- Not the case in ARM because the reading of registers always take place before the writing of a register of an instruction that follows.

WAW Hazard

- Arises when an instruction attempts to write a register that is a destination register of a previous instruction before that instruction completes the writing of its destination registers.

```
ADD  R7, R3, R7
EOR  R7, R9, R2
```

- Can only happen in processors that allow out-of-order instruction execution (not the case in ARM)

Solution for Data Hazards: Instructions Reordering

ARM code:

```
ADD  R2, R3, R1
EOR  R7, R0, R2
AND  R5,  R8, R7
ORR  R9, R3, R4
MOV  R11, R3
SUB  R12, R1, R6
RSB  R3, R10, R8
LDR  R0, [R4 ,R1]
```

ARM code:

```
ADD  R2, R3, R1
ORR  R9, R3, R4
MOV  R11, R3
SUB  R12, R1, R6
EOR  R7, R0, R2
AND  R5,  R8, R7
RSB  R3, R10, R8
LDR  R0, [R4 ,R1]
```

ARM code:

```
ADD  R2, R3, R1
ORR  R9, R3, R4
MOV  R11, R3
SUB  R12, R1, R6
EOR  R7, R0, R2
RSB  R3, R10, R8
LDR  R0, [R4 ,R1]
AND  R5,  R8, R7
```

Solution for Data Hazards: Instructions Reordering

ARM code:

```
ADD  R2, R3, R1
ORR  R9, R3, R4
MOV  R11, R3
SUB  R12, R1, R6
EOR  R7, R0, R2
RSB  R3, R10, R8
LDR  R0, [R4 ,R1]
AND  R5,  R8, R7
```

IF	ID	EXE	MEM	WB	
ADD					t
ORR	ADD				t+1
MOV	ORR	ADD			t+2
SUB	MOV	ORR	ADD		t+3
EOR	SUB	MOV	ORR	ADD	t+4
RSB	EOR	SUB	MOV	ORR	t+5
LDR	RSB	EOR	SUB	MOV	t+6
AND	LDR	RSB	EOR	SUB	t+7

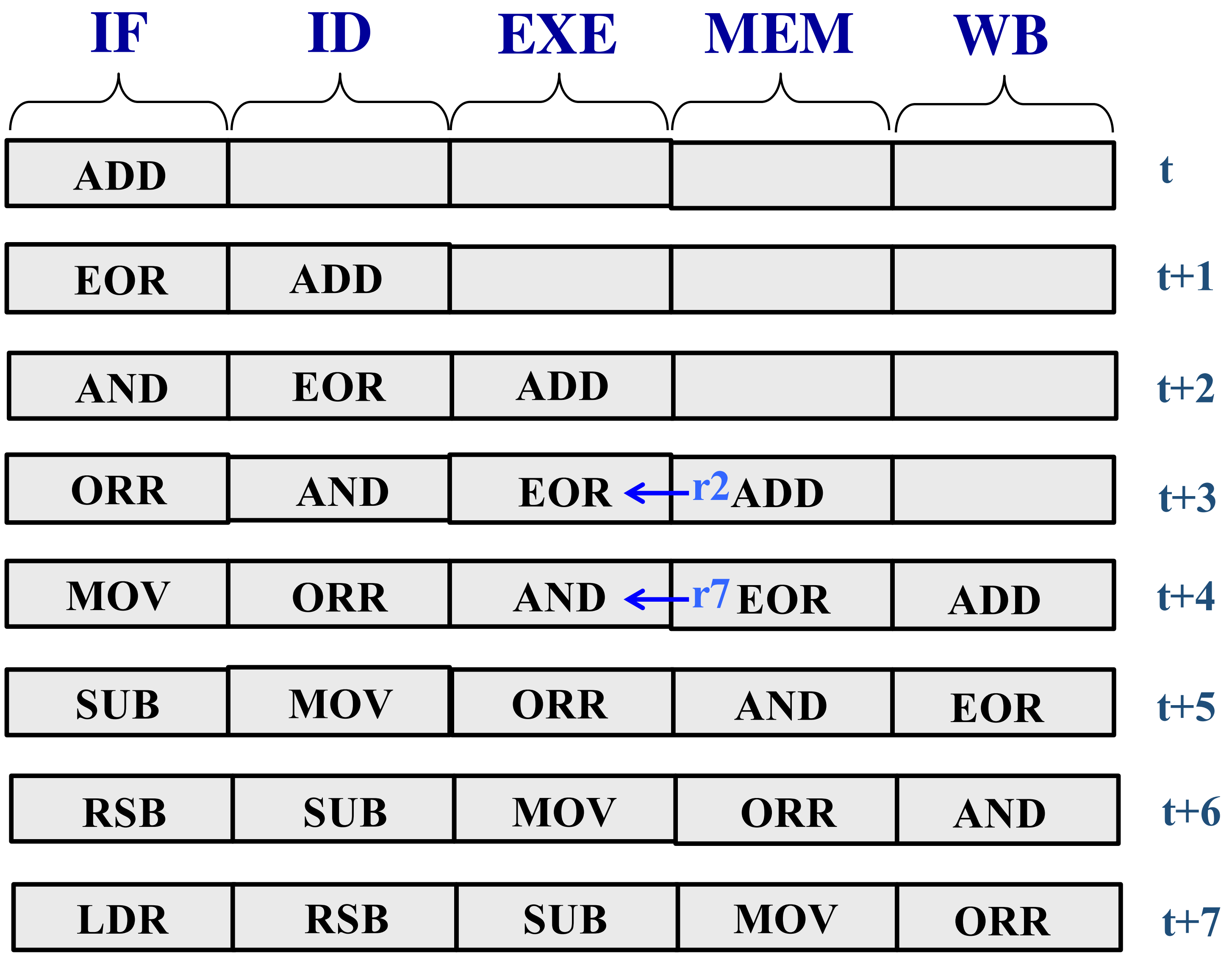
Solution for Data Hazards: Data Forwarding

- Allows the pipeline to use the value of a destination register after it is generated (without having to wait until after the **WB** stage)
    - *The value a destination register is generated in the **EXE** stage for arithmetic/logic instructions.*
    - *The value of a destination register is generated in the **MEM** stage for load/store instructions.*
- Eliminates all the penalty cycles due to the data hazard prevention
- Allows the dependent instruction to advance to the **ID** stage even if a source registers has not been updated

Example of Data Forwarding

ARM code:

```
ADD  R2, R3, R1
EOR  R7, R0, R2
AND  R5,  R8, R7
ORR  R9, R3, R4
MOV  R11, R3
SUB  R12, R1, R6
RSB  R3, R10, R8
LDR  R0, [R4 ,R1]
```

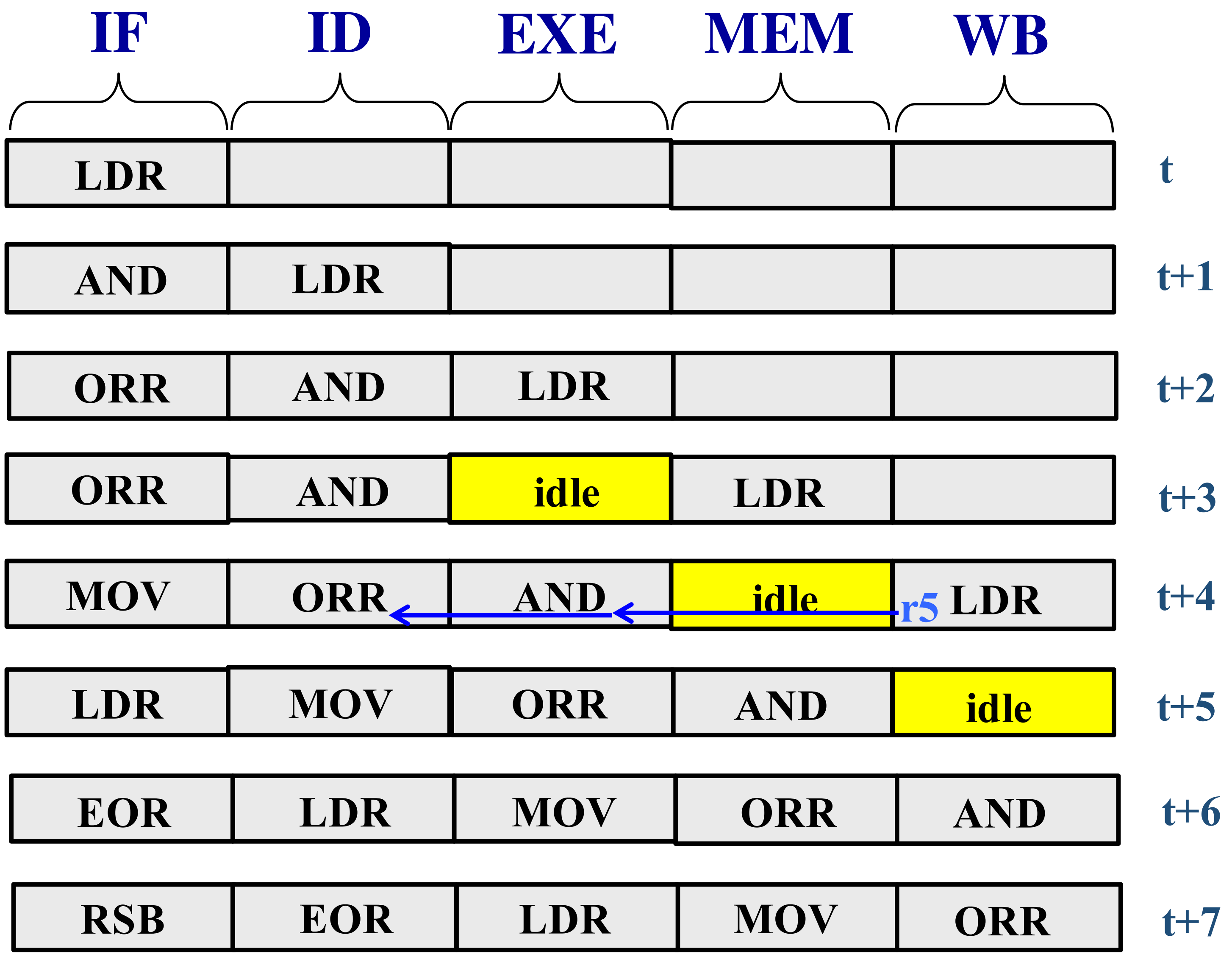


IF	ID	EXE	MEM	WB	
ADD					t
EOR	ADD				t+1
AND	EOR	ADD			t+2
ORR	AND	EOR ← r2	ADD		t+3
MOV	ORR	AND ← r7	EOR	ADD	t+4
SUB	MOV	ORR	AND	EOR	t+5
RSB	SUB	MOV	ORR	AND	t+6
LDR	RSB	SUB	MOV	ORR	t+7

Another Example of Data Forwarding

ARM code:

```
LDR  R5, [R3, R1]
AND  R5,  R8, R5
ORR  R9, R3, R5
MOV  R11, R3
LDR  R5, [R4 ,R1]
EOR  R12, R1, R6
RSB  R2, R10, R8
```



IF	ID	EXE	MEM	WB	
LDR					t
AND	LDR				t+1
ORR	AND	LDR			t+2
ORR	AND	idle	LDR		t+3
MOV	ORR	AND	idle	r5 LDR	t+4
LDR	MOV	ORR	AND	idle	t+5
EOR	LDR	MOV	ORR	AND	t+6
RSB	EOR	LDR	MOV	ORR	t+7

Control Hazards

- Typically, the instructions following a branch instruction in a program are fed into the pipeline following the branch instruction.
- If the branch is not taken there is no penalty
- If the branch is taken, the instructions following the branch on the pipeline must be cancelled.
- After the instructions are cancelled the branch target instructions must be supplied to the pipeline
- The cycles wasted as a result of the cancellation of the instructions plus the cycles it takes for the target instructions to start entering the pipeline after the cancellation of the instructions (the memory latency) constitute the branch execution penalty.

Control Hazard Example (memory latency of two cycles)

ARM code:

```
     BNE   FWD
     SUB
     AND

FWD  LDR
     EOR
     RSB
```

IF	ID	EXE	MEM	WB	
BNE					t
SUB	BNE				t+1
AND	SUB	BNE			t+2
idle	canceled	canceled	BNE		t+3
idle	idle	canceled	canceled	BNE	t+4
LDR	idle	idle	canceled	canceled	t+5
EOR	LDR	idle	idle	canceled	t+6
RSB	EOR	LDR	idle	idle	t+7

Alternatives for Reducing Branch Execution Time

- Loop buffer
- Delayed branch
- Branch prediction
- Multiple Prefetching
- Combinations of the above

Loop Buffer

- A small, very fast cache is used to keep the previous **n** instructions.
- The objective of this cache is to hold the instructions corresponding to a loop so that the target address can be fetched immediately after the branch condition is tested.
- It reduces the memory latency for fetching the target instructions.
- Still pays a penalty for the instructions following the branch that must be canceled

Effect of Loop Buffer (zero memory latency)

ARM code:

```
Back  LDR
      EOR
      RSB
      BNE    Back
      SUB
      AND
```

IF	ID	EXE	MEM	WB	
BNE					t
SUB	BNE				t+1
AND	SUB	BNE			t+2
LDR	canceled	canceled	BNE		t+3
EOR	LDR	canceled	canceled	BNE	t+4
RSB	EOR	LDR	canceled	canceled	t+5
BNE	RSB	EOR	LDR	canceled	t+6
SUB	BNE	RSB	EOR	LDR	t+7

Delayed Branch

- If the condition of the branch is asserted a branch to the target address is taken after the instruction following the branch is executed.
- The machine code programmer or the compiler will try to place a useful instruction after the branch instruction.
- If an useful instruction cannot be placed after the branch instruction a no op instruction must be placed instead (There will be no benefit.).
- SPARC and MIPS support this type of branch

Delayed Branch Example (memory latency of two cycles)

SPARC code:

```
    be    FWD
    sub
    and

FWD ld
    add
    or
```

IF	ID	EXE	MEM	WB	
be					t
sub	be				t+1
and	sub	be			t+2
idle	canceled	sub	be		t+3
idle	idle	canceled	sub	be	t+4
ld	idle	idle	canceled	sub	t+5
add	ld	idle	idle	canceled	t+6
or	add	ld	idle	idle	t+7

Effectiveness of Delayed Branch

- Depends on the compiler
  - *Place useful instructions after the branch instructions*
  - *Place nop instructions after the branch instructions*

- Improves with delay of more than one instruction

Branch Prediction

- The instruction fetch unit always make a prediction of what will be the result of the evaluation of the condition of a conditional branch.

- The instructions that are sent to the pipeline following the branch depend on the prediction
  - ***Prediction that the branch will take place***

    The stream of instructions at the target address are sent immediately after the branch instruction.
  - ***Prediction that the branch will no take place***

    The stream of instructions following the branch are sent immediately after the branch instruction.

Outcome of Branch Prediction

- If the prediction is asserted there is no branch penalty
- If the prediction is not asserted:
  - *The instructions sent to the pipeline after the branch are canceled.*
  - *The other stream of instructions is sent to the pipeline with a delay caused by the memory latency (the time it takes memory to respond).*

Prediction: Branch will not take place
Outcome of Prediction: Not Asserted

ARM code:

```
     BNE   FWD
     SUB
     AND
     STR
     ADD

FWD  LDR
     EOR
     RSB
     STR
```

IF	ID	EXE	MEM	WB	
BNE					t
SUB	BNE				t+1
AND	SUB	BNE			t+2
idle	canceled	canceled	BNE		t+3
idle	idle	cancelled	canceled	BNE	t+4
LDR	idle	idle	cancelled	canceled	t+5
EOR	LDR	idle	idle	canceled	t+6
RSB	EOR	LDR	idle	idle	t+7

Prediction: Branch will not take place
Outcome of Prediction: Asserted

ARM code:

```
        BNE    FWD
        SUB
        AND
        STR
        ADD

FWD     LDR
        EOR
        RSB
        STR
```

IF	ID	EXE	MEM	WB	
BNE					t
SUB	BNE				t+1
AND	SUB	BNE			t+2
STR	AND	SUB	BNE		t+3
ADD	STR	AND	SUB	BNE	t+4

Prediction: Branch will take place
Outcome of Prediction: Not Asserted

ARM code:

	BNE	FWD
	SUB	
	AND	
	STR	
	ADD	
FWD	LDR	
	EOR	
	RSB	
	STR	

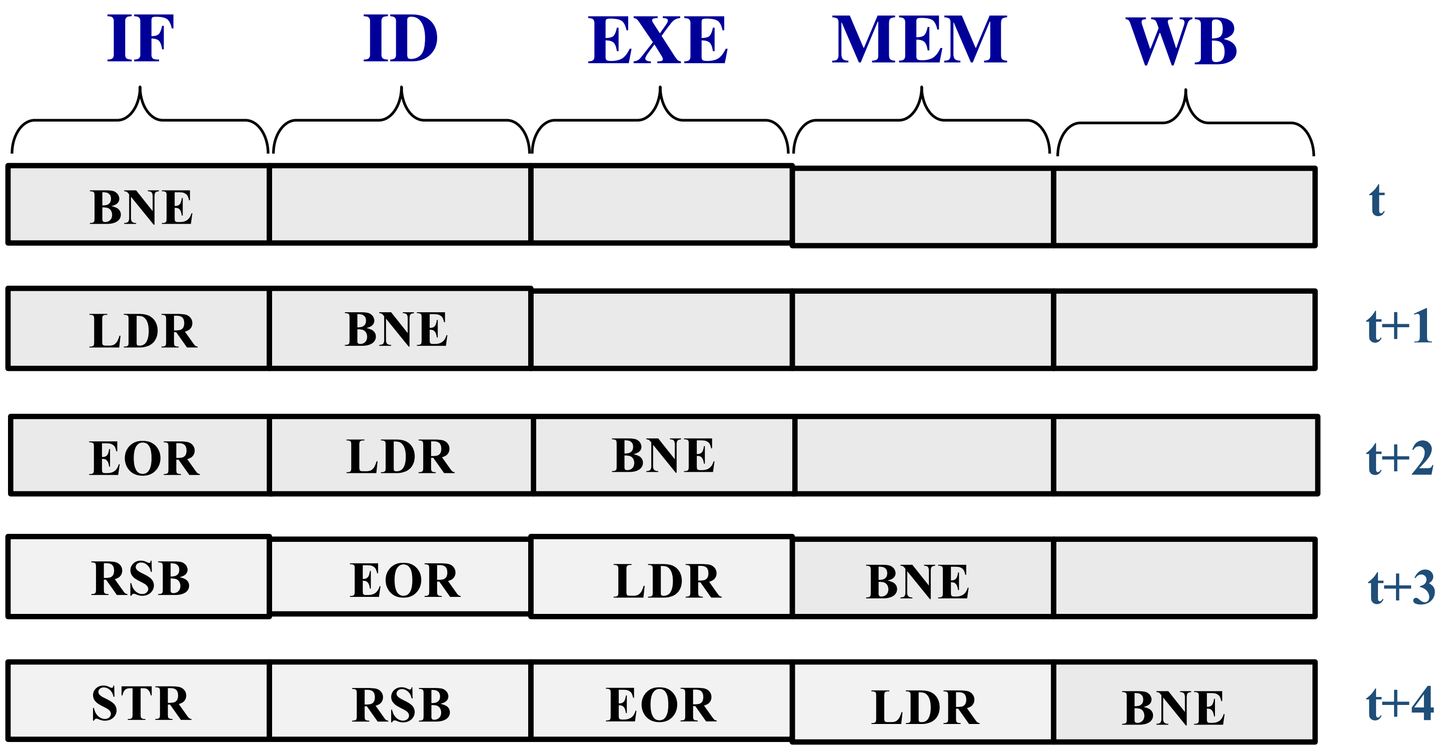
IF	ID	EXE	MEM	WB	
BNE					t
LDR	BNE				t+1
EOR	LDR	BNE			t+2
idle	canceled	canceled	BNE		t+3
idle	idle	canceled	canceled	BNE	t+4
SUB	idle	idle	canceled	canceled	t+5
AND	SUB	idle	idle	canceled	t+6
STR	AND	SUB	idle	idle	t+7

Prediction: Branch will take place
Outcome of Prediction: Asserted

ARM code:

```
     BNE   FWD
     SUB
     AND
     STR
     ADD

FWD  LDR
     EOR
     RSB
     STR
```



	IF	ID	EXE	MEM	WB
t	BNE				
t+1	LDR	BNE			
t+2	EOR	LDR	BNE		
t+3	RSB	EOR	LDR	BNE	
t+4	STR	RSB	EOR	LDR	BNE

Types of Branch Predictions

- Static Prediction
  - *The prediction is specified by the branch instruction*
  - *The compiler decides the prediction*
- Dynamic Prediction
  - *The hardware determines the prediction base on previous branch outcomes (taken/not taken)*

Static Prediction Schemes

- Predict backward branches taken and forward branches not taken
- Hint Bits
  - *Some architectures allow the compiler to specify some bits for a branch instruction that hints the outcome of the evaluation of its condition*

Dynamic Prediction Schemes

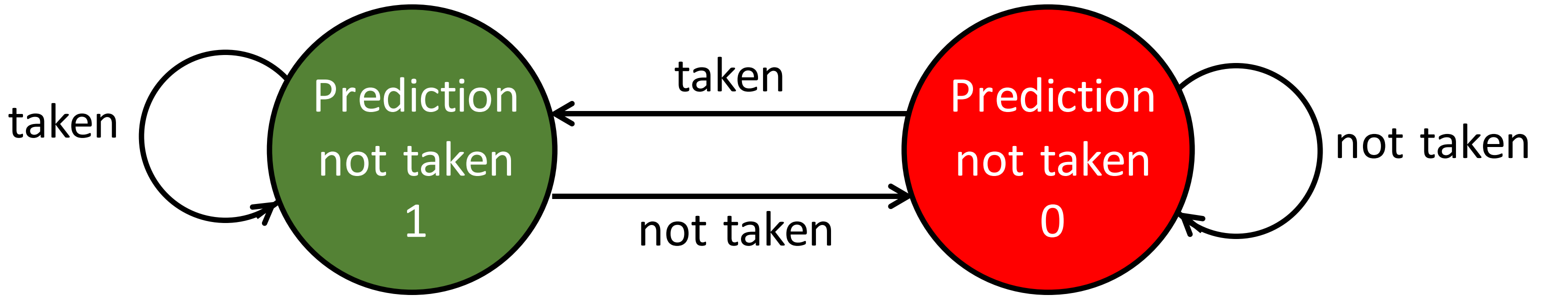
- N-bit Saturated Counter Predictors
  - *1-bit History Predictor*
  - *Two-bit Saturated Counter Predictor*
- Two Level Branch Predictor

N-bit Saturated Counter Predictors

- An n-bit counter is incremented if the outcome of a branch is taken and decremented if the the outcome is not taken.
- When the counter reaches the maximum count and the next branch outcome is taken the count remains the maximun (it saturates).
- When the counter reaches the minimum count and the next branch outcome is not taken the count remains the minimum (it saturates).
- The prediction is not taken if the count is less than 2^{n-1}. Otherwise the prediction is taken.

1 bit History Predictor

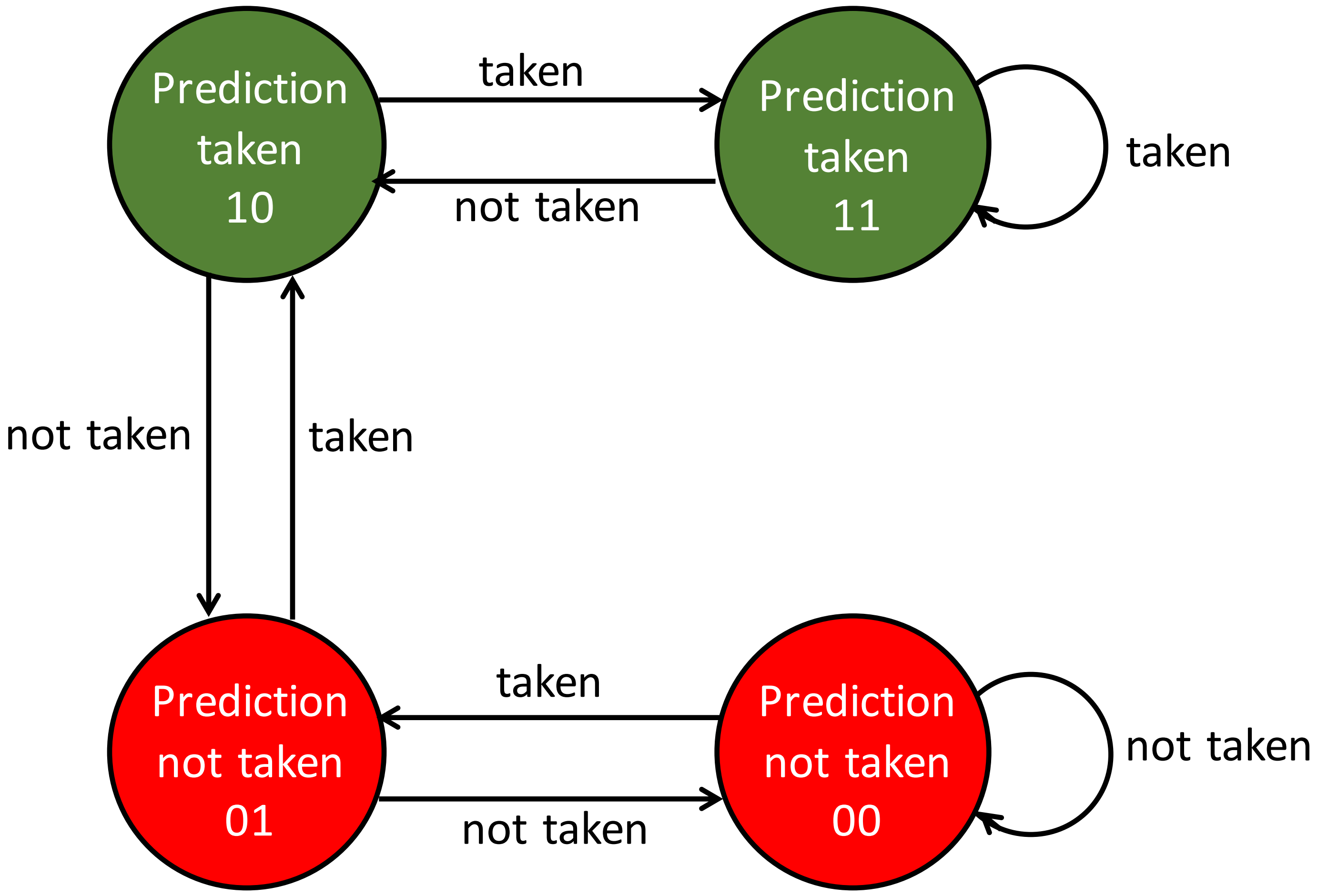
- Predicts the same branch outcome the last time it was executed
  - *Set to one if a branch is taken (a one predicts taken)*
  - *Set to zero if a branch is not taken (a zero predicts not taken)*



Two-bit Saturated Counter Predictor

- Predicts with hysteresis (takes into account the history of a branch outcome further that the last time it was executed)
  - *Counter incremented if branch taken (no further incrementation when a count of three is reached)*
  - *Counter decremented if branch not taken (no further decrementation when a count of zero is reached)*
  - *If the most significant bit of the counter is one predict branch taken, otherwise predict branch not taken*

Two-bit Saturated Counter Predictor



Issues with N-bit Saturated Counter Predictors

- The saturated counters are kept on a one-dimensional table of 2^k entries.
- The branches are mapped to the table using the least significant k bits of the branch address and one of the following types of mapping:
  - *Direct mapping*
  - *Fully associative mapping*
  - *Set associative mapping*
  - *Hashing function*
- Are prone to an aliasing problem because more than one branch could use the same counter (there is not one counter per branch address)

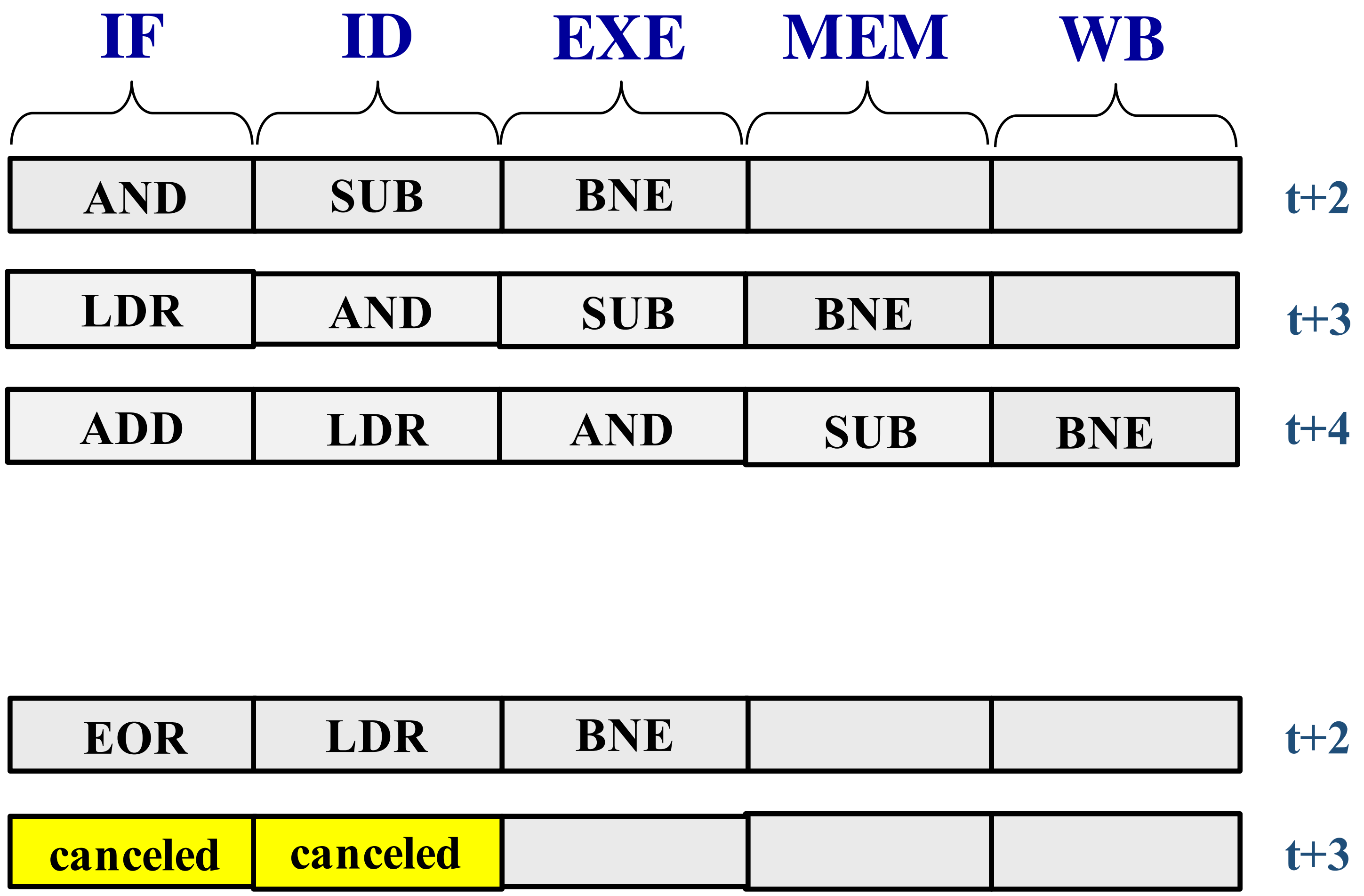
Two Level Branch Predictor

- Takes into consideration that the outcome of a branch depends on the history of other recent branches and on the history of the branch itself
- Uses a shift register to shift in the results of branch outcomes (1 for taken and 0 for not taken)
- Uses the shift register to address a two-dimensional table of saturated counters.
- It is prone to an aliasing problem

Multiple Prefetching

- The instruction stream following the branch instruction and the branch target stream are executed in parallel pipelines until the branch condition is evaluated
- Once the condition is evaluated one of the instructions stream is cancelled and the other proceeds
- Requires a branch management unit

Multiple prefetching: Branch not taken



ARM code:

```
      BNE    FWD
      SUB
      AND
      LDR
      ADD

FWD   LDR
      EOR
      RSB
      STR
```

Multiple prefetching: Branch taken

ARM code:

BNE	FWD
SUB	
AND	
LDR	
ADD	

IF	ID	EXE	MEM	WB	
AND	SUB	BNE			t+2
canceled	canceled				t+3

FWD	LDR
	EOR
	RSB
	STR

IF	ID	EXE	MEM	WB	
EOR	LDR	BNE			t+2
RSB	EOR	LDR	BNE		t+3
STR	RSB	EOR	LDR	BNE	t+4

Disadvantages of Multiple Prefetching

- Introduces bus and registers contention
- May end up requiring additional pipelines when a branch is immediately followed by another branch

Other Mechanisms for Reducing the Effect of Interlocks

- Superscalar
  - *Rely on multiple function units that operate in parallel (adders, multipliers, dividers)*
  - *Potentially, can execute instructions a rate higher than one per cycle*
  - *Instructions requiring more than one cycle for executing do not obstruct others from executing*
- Superpipeline
  - *Many pipeline stages that can allow to reduce the system clock period*
  - *May increase the penalty for branches.*