Control Dependence Handling : Predicated Execution and Loop Unrolling

Lokwon Kim

How to Handle Control Dependences

- Critical to keep the pipeline full with correct sequence of dynamic instructions.

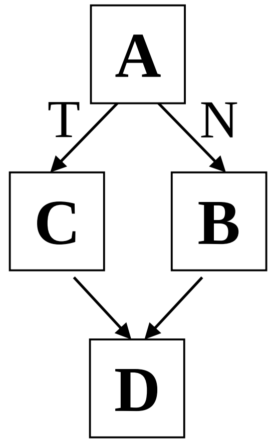
- Potential solutions if the instruction is a control-flow instruction:
- Stall the pipeline until we know the next fetch address
- Backward Taken Forward Not Taken
- Employ delayed branching (branch delay slot)
- Guess the next fetch address (branch prediction)
- Do something else (fine-grained multithreading)
- Eliminate control-flow instructions (predicated execution, Loop Unroll)
- Fetch from both possible paths (if you know the addresses of both possible paths) (multipath execution)

Predicated Execution

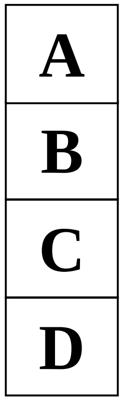
- Idea: Compiler converts control dependence into data dependence → **branch is eliminated!**
  - Each instruction has a predicate bit based on the predicate computation
  - **Only instructions with TRUE predicates are actually executed** (others turned into NOPs)

(normal branch code) (predicated code)

```
if (cond) {
      b = 0;
}
else {
      b = 1;
}
```



```
A   p1 = (cond)
    branch p1, TARGET
B   addi b, $zero, 1
    jump JOIN
C   TARGET:
    addi b, $zero, 0
D   JOIN:
    add  x, b, 1
```



```
A   p1 = (cond)
B   (!p1) addi  b, $zero, 1
C   (p1) addi  b, $zero, 0
D   add  x, b, 1
```

Conditional Move Operation

- Suppose we had a **Conditional** Move instruction…
  - EX) CMOV condition, R1 ← R2
    - R1 = (condition == true) ? R2 : NOP
  - Employed in most modern ISAs (x86, Alpha)

- Code example with branches vs. CMOVs

if (a == 5) {b = 4;} else {b = 3;}

CMPEQ condition, a, 5;

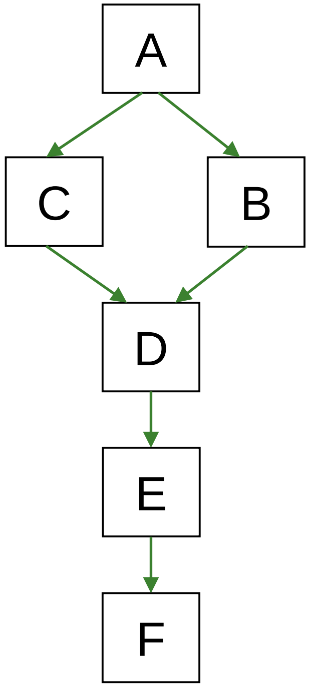
CMOV condition, b ← 4;

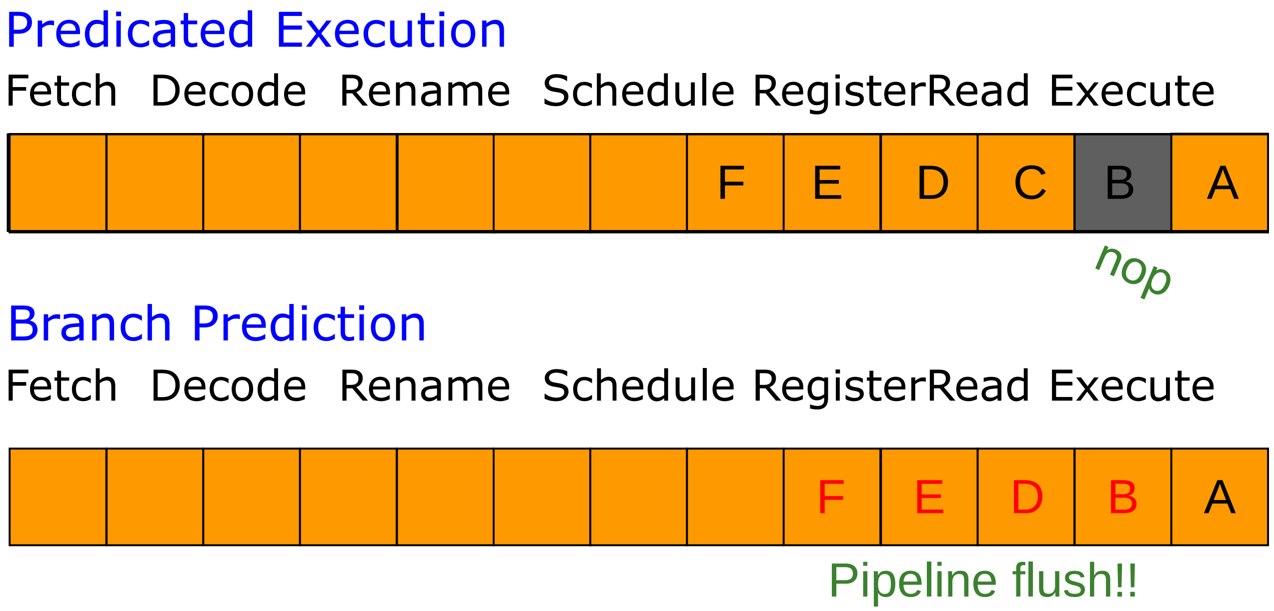
CMOV !condition, b ← 3;

Means insert 4 to b only if condition was met, otherwise insert 3.

Predicated Execution (II)

- Predicated execution can be **high performance and energy-efficient**




Predicated Execution
Fetch Decode Rename Schedule RegisterRead Execute
F E D C B A
nop
Branch Prediction
Fetch Decode Rename Schedule RegisterRead Execute
F E D B A
Pipeline flush!!

Predicated Execution (III)

- Advantages:

  + Eliminates mispredictions for hard-to-predict branches

    + No need for branch prediction for some branches

    + Good if misprediction cost > useless work due to predication

  + Enables code optimizations hindered by the control dependency

    + Can move instructions more freely within predicated code

- Disadvantages:

  -- Causes useless work for branches that are easy to predict

    -- Reduces performance if misprediction cost < useless work

    -- Adaptivity: Static predication is not adaptive to run-time branch behavior. Branch behavior changes based on input set, phase, control-flow path.

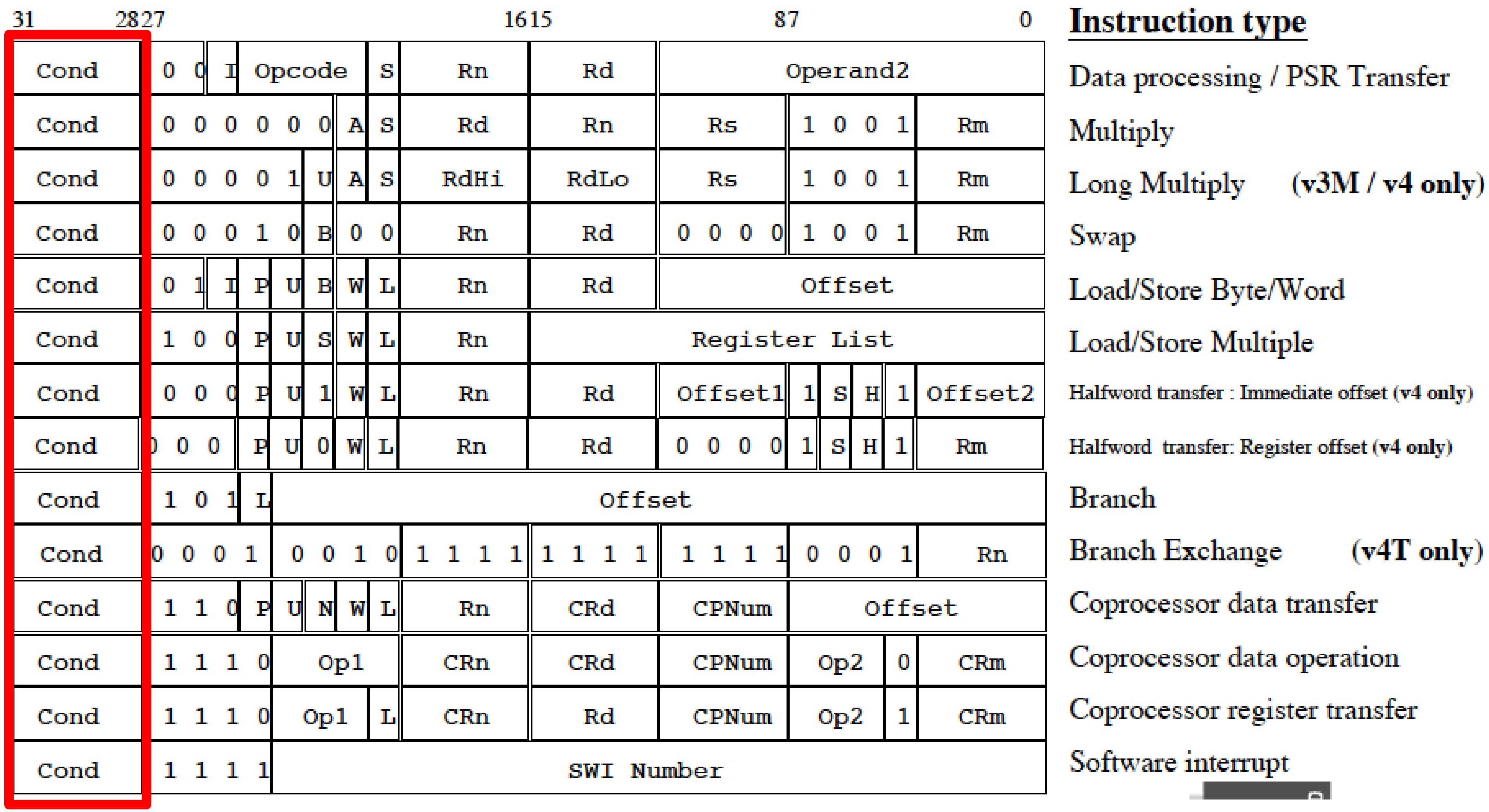
  -- **ISA support** and Additional hardware

Example: Conditional Execution in ARM ISA

- **Almost all ARM instructions can include an optional condition code.**

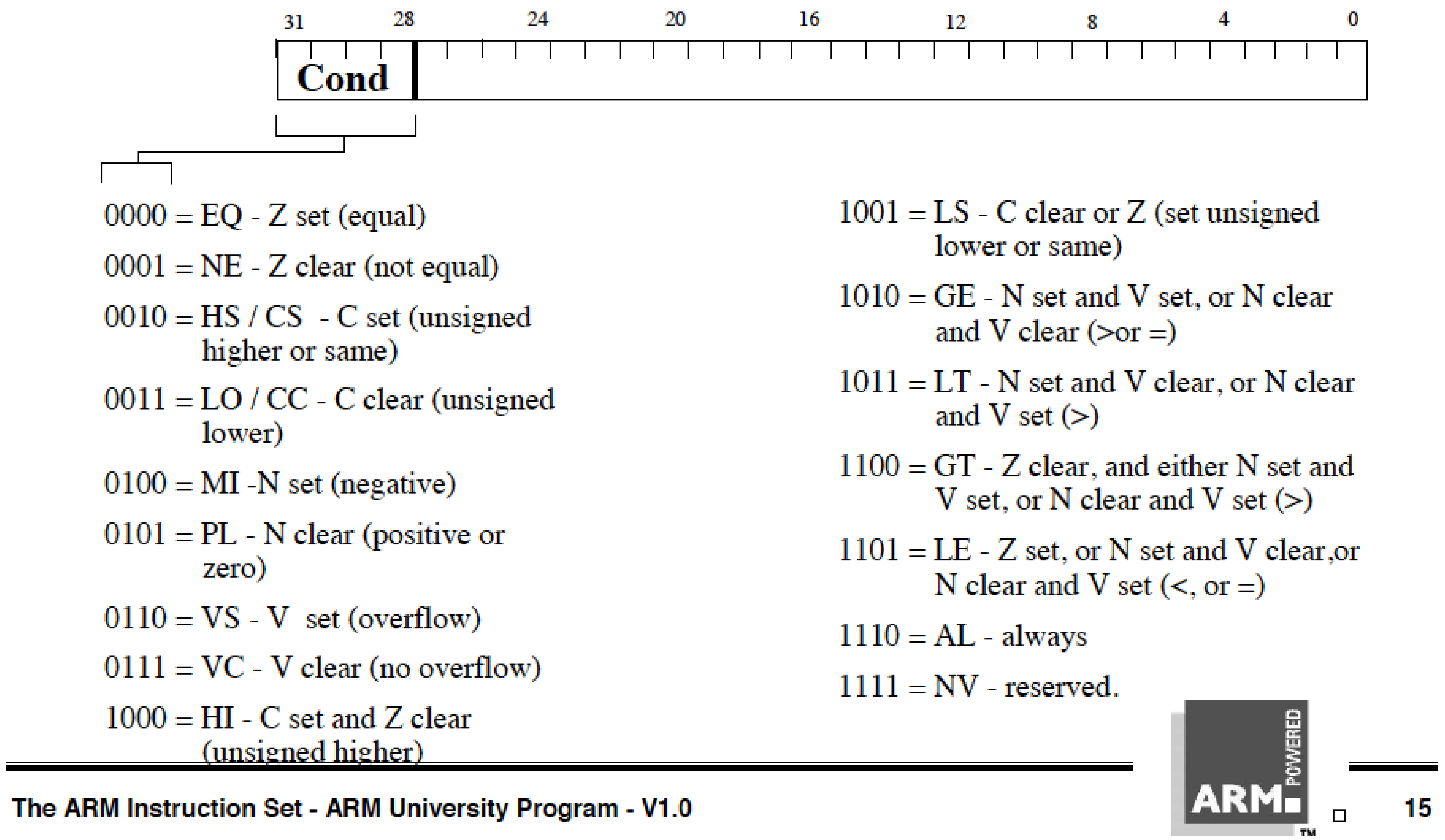
- An instruction with a condition code is only executed if the condition code flags in the CPSR meet the specified condition.

Conditional Execution in ARM ISA



31 28	27	16	15	8	7	0	**Instruction type**
Cond	0 0 I Opcode S	Rn	Rd	Operand2			Data processing / PSR Transfer
Cond	0 0 0 0 0 0 A S	Rd	Rn	Rs	1 0 0 1	Rm	Multiply
Cond	0 0 0 0 1 U A S	RdHi	RdLo	Rs	1 0 0 1	Rm	Long Multiply (v3M / v4 only)
Cond	0 0 0 1 0 B 0 0	Rn	Rd	0 0 0 0	1 0 0 1	Rm	Swap
Cond	0 1 I P U B W L	Rn	Rd	Offset			Load/Store Byte/Word
Cond	1 0 0 P U S W L	Rn	Register List				Load/Store Multiple
Cond	0 0 0 P U 1 W L	Rn	Rd	Offset1	1 S H 1	Offset2	Halfword transfer : Immediate offset (v4 only)
Cond	0 0 0 P U 0 W L	Rn	Rd	0 0 0 0	1 S H 1	Rm	Halfword transfer: Register offset (v4 only)
Cond	1 0 1 L	Offset					Branch
Cond	0 0 0 1 0 0 1 0	1 1 1 1	1 1 1 1	1 1 1 1	0 0 0 1	Rn	Branch Exchange (v4T only)
Cond	1 1 0 P U N W L	Rn	CRd	CPNum	Offset		Coprocessor data transfer
Cond	1 1 1 0 Op1	CRn	CRd	CPNum	Op2 0	CRm	Coprocessor data operation
Cond	1 1 1 0 Op1 L	CRn	Rd	CPNum	Op2 1	CRm	Coprocessor register transfer
Cond	1 1 1 1	SWI Number					Software interrupt

Conditional Execution in ARM ISA


31
28
24
20
16
12
8
4
0
Cond

0000 =	EQ - Z set (equal)
0001 =	NE - Z clear (not equal)
0010 =	HS / CS - C set (unsigned higher or same)
0011 =	LO / CC - C clear (unsigned lower)
0100 =	MI -N set (negative)
0101 =	PL - N clear (positive or zero)
0110 =	VS - V set (overflow)
0111 =	VC - V clear (no overflow)
1000 =	HI - C set and Z clear (unsigned higher)
1001 =	LS - C clear or Z (set unsigned lower or same)
1010 =	GE - N set and V set, or N clear and V clear (>or =)
1011 =	LT - N set and V clear, or N clear and V set (>)
1100 =	GT - Z clear, and either N set and V set, or N clear and V set (>)
1101 =	LE - Z set, or N set and V clear,or N clear and V set (<, or =)
1110 =	AL - always
1111 =	NV - reserved.

The ARM Instruction Set - ARM University Program - V1.0 15

Conditional Execution in ARM ISA

* **To execute an instruction conditionally, simply postfix it with the appropriate condition:**
  - For example an add instruction takes the form:
    - `ADD r0,r1,r2          ; r0 = r1 + r2 (ADDAL)`
  - To execute this only if the zero flag is set:

    -
      ```
      ADDEQ r0,r1,r2        ; If zero flag set then…
                            ; ... r0 = r1 + r2
      ```

Loop Unrolling

- **Loop unrolling is a program transformation that trades code size for execution speed.**

```
for ( int i=0; i<16; i++ )
data[i] = i;
```

Original code

```
for ( int i=0; i<16; i+=2 )
{
data[i] = i;
data[i+1] = i+1;
}
```

It can be 'unrolled' by instantiating the loop body twice.

```
for ( int i=0; i<16; i+=4 )
{
data[i] = i;
data[i+1] = i+1;
data[i+2] = i+2;
data[i+3] = i+3;
}
```

How about 4?

How about 16? -> **the loop disappears**
-> eliminate the branch instructions