CG2028 Cheatsheet AY21/22 Sem 1

by Richard Willie

ARMv7-M Instructions

.equ

```
.equ LABEL, 0x12345678     @ Sets value of LABEL to 0x12345678
```

Usage:

```
LDR R1, =LABEL             @ Load R1 with 0x12345678
```

.word

```
LABEL:    .word 123, 456   @ LABEL = address of 123
POINTER:  .word LABEL+4    @ POINTER = address of 456
```

.lcomm

```
.lcomm LABEL 4             @ Reserves 4 bytes (static variable)
```

LDR

Loading memory content

Offset addressing:

```
LDR R0, [R1]          @ R1 stores an address, R0 = content of the address
LDR R0, [R1, #4]      @ EA = R1 + 4, R0 = EA
LDR R0, [R1, #4]!     @ EA = R1 + 4, R0 = EA, R2 = EA
LDR R0, [R1], #4      @ EA = R1, R0 = EA, R1 = R1 + 4
```

PC-relative addressing:

```
LDR R0, NUM1          @ NUM1 is declared via .word
```

Pseudo-instruction:

```
LDR R0, =0x12345678   @ Loading a 32-bit value
```

Loading memory address

Pseudo-instruction:

```
LDR R0, =NUM1         @ NUM1 is declared via .word
```

MOV

Format:

```
MOV Rd, OP2
MOV Rd, #imm16
MOVW Rd, #imm16
```

Loading memory content

```
MOV R0, R1            @ R1 stores a content, R0 = R1
MOV R0, #0x12345678   @ Loading a 32-bit value
```

Note:

- Use LDR for transfer to register from memory.
- Use MOV for transfer to register from register or constant.

ADD

```
ADD {Rd,} Rn, OP2
ADD {Rd,} Rn, #imm12
```

SUB

```
SUB {Rd,} Rn, OP2
SUB {Rd,} Rn, #imm12
```

MUL, MLA

```
MUL {Rd,} Rn, Rm
MLA Rd, Rn, Rm, Ra
```

DIV

```
SDIV {Rd,} Rn, Rm
UDIV {Rd,} Rn, Rm
```

CMP, CMN

```
CMP Rn, OP2    @ Performs Rn - OP2 and updates condition flags
CMP Rn, OP2    @ Performs Rn + OP2 and updates condition flags
```

B, BL, BLX, BX

```
B{cond} LABEL
BL{cond} LABEL
BLX{cond} Rm
BX{cond} Rm
```

Note:

- BL: Jump from main to function.
- BLX: Jump from main to function.
- BX: Jump from function to main.

IT

Example:

```
ITTETT NE
ADDNE R0, R0, R1
ADDEQ R0, R0, R3
ADDNE R2, R4, #1
MOVNE R5, R3
```

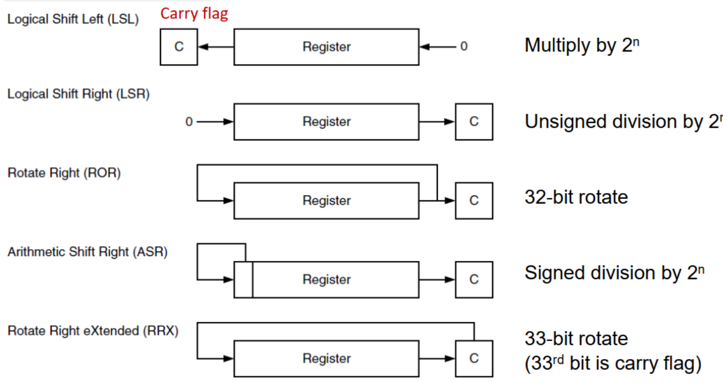
Note: not more than 4 instructions.

ADD, ORR, EOR

```
AND{S} {Rd,} Rn, OP2
ORR{S} {Rd,} Rn, OP2
EOR{S} {Rd,} Rn, OP2
```

Shift & Rotate

```
LSL{S} Rd, Rm, Rs
LSL{S} Rd, Rm, #n     @ n = 0...31
LSR{S} Rd, Rm, Rs
LSR{S} Rd, Rm, #n     @ n = 1...32
ASR{S} Rd, Rm, Rs
ASR{S} Rd, Rm, #n     @ n = 1...32
ROR{S} Rd, Rm, Rs
ROR{S} Rd, Rm, #n     @ n = 1...31
RRX{S} Rd, Rm, Rs
```



Note: The **S suffix** should be specified in order to update the Carry flag, e.g. LSL**S** R0, R1, #2

Instruction Formats

The design and encoding of the instructions used in CG2028 is not compliant with any version of ARM Architecture. There are the 3 main instruction formats (DP, Memory, and Branch).

Data-processing

List of DP operations:

cmd	Instruction	Operation
0000	AND	Logical AND
0001	EOR	Logical Exclusive OR
0010	SUB	Subtract
0011	RSB	Reverse Subtract
0100	ADD	Add
0101	ADC	Add with Carry
0110	SBC	Subtract with Carry
0111	RSC	Reverse Subtract with Carry
1000	TST	Test Update flags after AND
1001	TEQ	Test Equivalence Update flags after EOR
1010	CMP	Compare Update flags after SUB
1011	CMN	Compare Negated Update flags after ADD
1100	ORR	Logical OR
1101	MOV	Move
1110	BIC	Bit Clear
1111	MVN	Move Not

Note: Multiplication is not one of the 16 ALU operations, though it is considered a DP operation.

DP Register Operand2 Format

```
OP{S} Rd, Rn, Rm
```

31:28	27:26	25	24:21	20	19:16	15:12	11:5	4	3:0
X	op	I	cmd	S	Rn	Rd	X	M	Rm
			funct						
4 bits	2 bits		6 bits		4 bits	4 bits	7 bits	1 bit	4 bits

- op = 00
- I = 0
- S = 1 if suffix S is specified, 0 otherwise
- M = 0

DP Immediate Operand2 Format

```
OP{S} Rd, Rn, #imm8
```

31:28	27:26	25	24:21	20	19:16	15:12	11:8	7:0
X	op	I	cmd	S	Rn	Rd	X	imm8
			funct					
4 bits	2 bits		6 bits		4 bits	4 bits	4 bits	8 bits

- op = 00
- I = 1
- S = 1 if suffix S is specified, 0 otherwise
- imm8 = 8-bit **unsigned** immediate

Multiply Instruction Format

```
MUL Rd, Rm, Rs
MLA Rd, Rm, Rs, Rn
```

31:28	27:26	25	24:21	20	19:16	15:12	11:8	7:5	4	3:0
X	op	I	cmd	S	Rn	Rd	Rs	X	M	Rm
			funct							
4 bits	2 bits		6 bits		4 bits	4 bits	4 bits	3 bits	1 bit	4 bits

- op = 00
- I = 0
- cmd = 0000 for MUL, 0001 for MLA
- S = 1 if suffix S is specified, 0 otherwise
- M = 1

Memory

`OP Rd, [Rn, #imm8]`

31:28	27:26	25	24	23	22	21	20	19:16	15:12	11:8	7:0
X	op	X	P	U	X	W	L	Rn	Rd	X	imm8
		funct									
4 bits	2 bits	6 bits						4 bits	4 bits	4 bits	8 bits

- op = 01
- P = 1
- U = 1 for EA = Rn + imm8, 0 for EA = Rn - imm8
- W = 0
- L = 1 for load, 0 for store
- imm8 = 8-bit **unsigned** immediate

Branch

`B{cond} LABEL`

`LABEL encoded as #imm8`

31:28	27:26	25	24	23	22	21	20	19:8	7:0
cond	op	X	X	U	X	X	X	X	imm8
		funct							
4 bits	2 bits	6 bits						12 bits	8 bits

- op = 10
- U = 1 for BTA = PC + 4 + imm8, 0 for BTA = PC + 4 - imm8
- imm8 = 8-bit **unsigned** immediate

Branch condition codes:

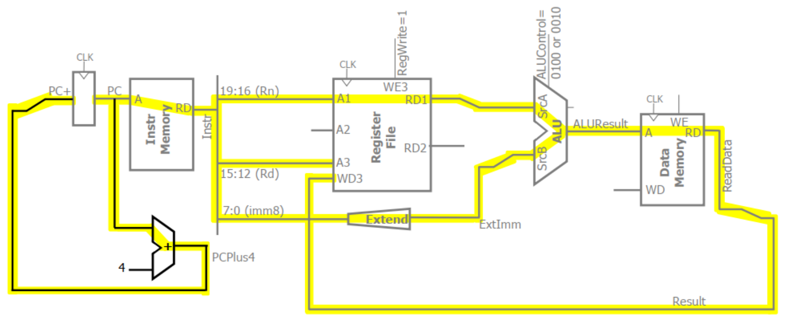
cond	Mnemonic	Name	Condition Checked
0000	EQ	Equal	Z
0001	NE	Not equal	$\overline{Z}$
0010	CS/HS	Carry set/Unsigned higher or same	C
0011	CC/LO	Carry clear/Unsigned lower	$\overline{C}$
0100	MI	Minus/Negative	N
0101	PL	Plus/Positive of zero	$\overline{N}$
0110	VS	Overflow/Overflow set	V
0111	VC	No overflow/Overflow clear	$\overline{V}$
1000	HI	Unsigned higher	$\overline{Z}$ AND C
1001	LS	Unsigned lower or same	Z OR $\overline{C}$
1010	GE	Signed greater than or equal	$\overline{N \oplus V}$
1011	LT	Signed less than	$N \oplus V$
1100	GT	Signed greater than	$\overline{Z}$ AND $(\overline{N \oplus V})$
1101	LE	Signed less than or equal	Z OR $(N \oplus V)$
1110	AL (or none)	Always/Unconditional	Ignored

Note: Flags are set by instructions with suffix S, except for CMP, CMN, TST, TEQ which automatically set flags but the result is discarded.

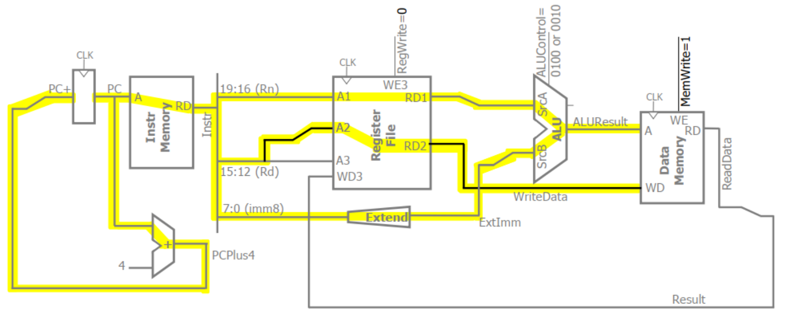
Microarchitecture Design

Datapath

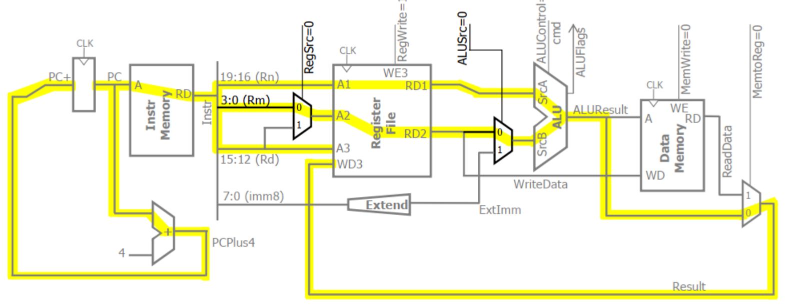
LDR



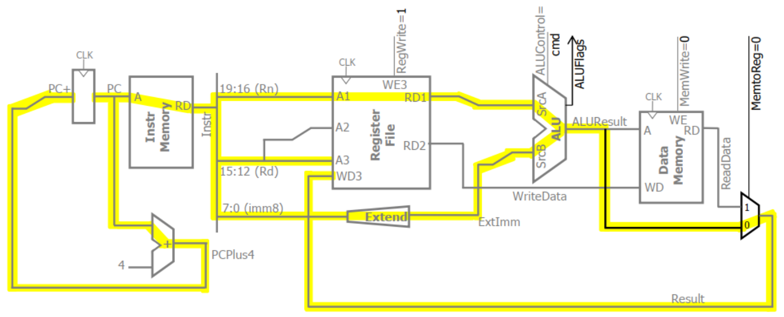
STR



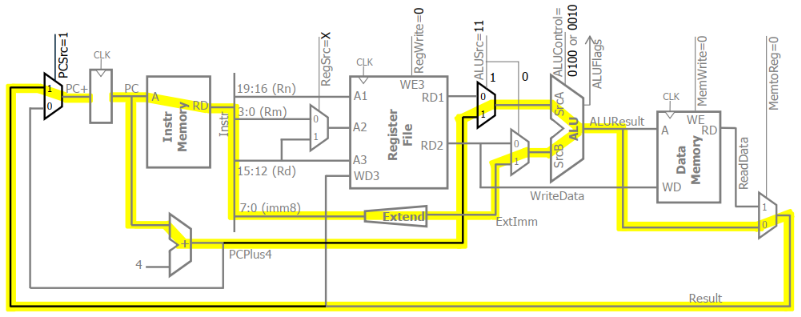
Data Processing (Register)



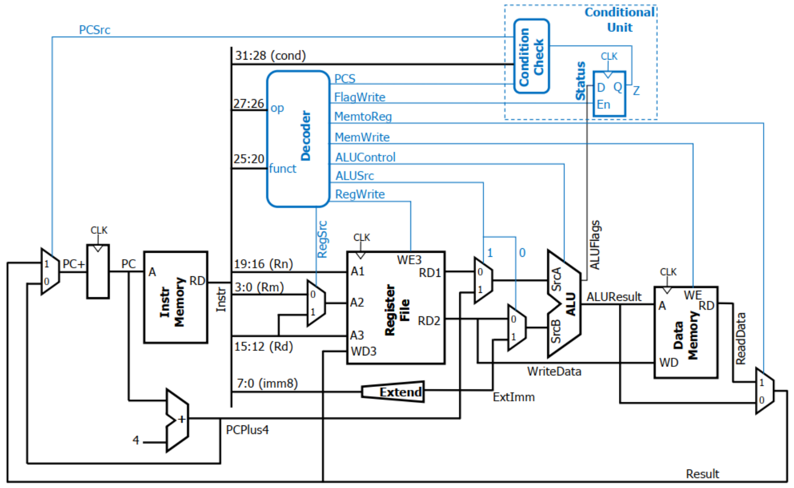
Data Processing (Immediate)



Branch



Single-Cycle Processor with Control



- CLK = 1 bit
- PCS = 1 bit
- PCSrc = 1 bit
- FlagWrite (En) = 1 bit
- MemtoReg = 1 bit
- MemWrite (WE) = 1 bit
- RegWrite (WE3) = 1 bit
- RegSrc = 1 bit
- Z (Q) = 1 bit
- D = 1 bit
- ALUFlags = 1 bit if only output Z, otherwise 4 bits
- op = 2 bits
- ALUSrc = 2 bits
- cond = 4 bits
- ALUControl = 4 bits
- Rn = 4 bits
- Rm = 4 bits
- Rd = 4 bits
- funct = 6 bits
- imm8 = 8 bits
- All other connections = 32 bits

Control Unit Design

Decoder

- PCS = (op == 10)
  - op = Instr[27:26]
  - Asserted only for branch, to write branch target to PC. Passed through conditional unit before being used in the datapath.
- FlagWrite = (op == 00) && (S == 1)
  - S = funct[0] = Instr[20]
  - Asserted for DP with S suffix, as only they modify flags.
- MemtoReg = (op == 01) && (L == 1)
  - L = funct[0] = Instr[20]
  - Asserted only for load, as the destination register gets data read from the data memory.
- MemWrite = (op == 01) && (L == 0)
  - Asserted only for store, as store alone writes to the data memory.
- ALUControl = (op == 00) ? cmd : (U ? 0100 : 0010)
  - U = funct[3] = Instr[23]
  - 0100 = ALUControl for addition
  - 0010 = ALUControl for subtraction
  - For DP, ALUControl is cmd. For memory and branch, U bit decides whether imm8 is added or subtracted (i.e., whether the offset is positive or negative).
- ALUSrc[0] = !((op == 00) && (I == 0))
  - I = funct[5] = Instr[25]
  - For all except DP with register as Operand2, ALU_SrcB is immediate.
- ALUSrc[1] = PCS

- ALU_SrcA is PCPlus4 only for branch (doesn't matter whether branch is taken or not. ALUResult is discarded when the branch is not taken away).

- RegWrite = (op == 00) || ((op == 01) && (L == 1))
  - All DP instructions and load write to a destination register, branch and store doesn't.
  - Extended functionality for CMP:
    - RegWrite = ((op == 00) && !(cmd == 1010) || ((op == 01) && (L == 1))
    - S = 1
    - cmd = 1010 (can be extended to accommodate other DP instructions such as TST, TEQ, CMN)
    - Like SUBS, but the result is not written to a register. So we need to modify RegWrite signal.
    - No change to datapath needed for implementation.
- RegSrc = MemWrite
  - For store, RA2 = Rd. For all other instructions reading a second register, RA2 is Rm.

Condition Check

- PCSrc = PCS && ((cond == 0000) ? (Z == 1) : 1)
  - For a branch instruction:
    - When the condition specified is EQ (0000) and when Z flag is set, branch is taken.
    - When the condition specified is AL (1110), branch is taken irrespective of the flags. For simplicity, we just ignore flags if the condition specified is not EQ.
  - This will cause ALUResult (PCPlus4 +/-imm8) to be written to PC instead of PCPlus4.

Cache Memory Principles

Memory Hierarchy

1. M_0: Registers in CPU
2. M_1: Cache memories (SRAM)
3. M_2: Main memory (DRAM)
4. M_3: Disk Storage
5. M_4: Tape units/optical disks

Basic idea:

- Each level holds the most frequently accessed data from the immediate higher level.
- Reduces the effect of lower speed of the higher level without increasing the overall cost significantly.

Properties:

- Coherence (consistency)
  - Copies of the same data must have the same information at all levels where the data is currently residing. In other words, if a word is modified in a cache, it must be updated at all levels.
- Locality of references
  - The memory access pattern tend to be clustered in certain regions in time, space, and ordering.
  - 90-10 rule by Hennessy and Patterson (1990) - a typical program may spend 90% of its execution time on only 10% of the code such as the innermost loop of a nested loop.
  - Temporal: Recently referenced items are likely to be referenced in the near future - keep recently accessed data at a faster level.
  - Spatial: Refers to the tendency of a process to access the items whose addresses are near to one another - when accessing data, bring nearby data also into a faster level.

Memory Capacity Planning

- Hit ratios
  - When a memory M_i is accessed and if the desired word is found, it is referred to as a *hit*, otherwise *miss*.
  - The hit ratio (h_i) is the probability that a word/information will be found when accessed in M_i. Miss ratio is $1 - h_i$.
  - The hit ratios at successive levels are a function of memory capacities, management policies, and program behaviour.
  - $h_0 = 0$ and $h_n = 1$. This means that the CPU always access M_i first and access to the outermost level is always a hit.
- Access frequency at a level i is defined as
  - $f_i = (1 - h_1)(1 - h_2) \dots (1 - h_{i-1})h_i$
  - Note that $f_1 + f_2 + \dots + f_n = 1$ and $f_1 = h_1$
  - Due to locality property, the access frequency at level i is greater than $i + 1$.
  - This means that the inner levels are accessed moire often than the outer levels.
- Effective access time is defined as

$$T_{eff} = f_1 t_1 + f_2 t_2 + \dots + f_n t_n$$

  where t_i is the access time at level i.

- The total cost of a memory hierarchy is estimated as

$$C_{total} = c_1 s_1 + c_2 s_2 + \dots + c_n s_n$$

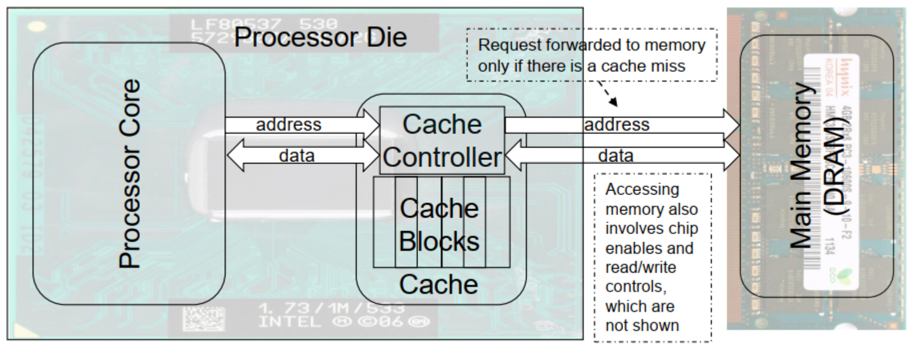
  where c_i is the cost/MB and s_i is the size (in MB) at level i.

- Hierarchy optimization involves minimizing

$$T_{eff} \text{ given } C_{total} < C_{max} \quad \text{or} \quad C_{total} \text{ given } T_{eff} < T_{max}$$

- The optimal design should result in a T_{eff} close to T_1 and a total cost close to c_n.

Cache Working Principle


Processor Die
Request forwarded to memory only if there is a cache miss
Processor Core
address
data
Cache Controller
Cache Blocks
Cache
address
data
Accessing memory also involves chip enables and read/write controls, which are not shown
Main Memory (DRAM)

- When a read request is received from CPU, the contents of a block memory words containing the location specified are transferred to the cache
  - Block is also called *cache line*, typically 64 bytes.

  - Where to place the incoming block in the cache is decided by the mapping function.

- When the program asks for any of the location from this block, the desired contents are read directly from cache
  - CPU need not even be aware of the presence of the cache and issues addresses meant for main memory.
  - Checking if the required data/block is present in the cache is performed by the cache controller. If yes, a *cache hit* is said to occur.
- When a block occupying cache is not referenced fora long time, it is pushed back to MM to make space for another block.
  - Which block to replace is decided by the replacement algorithms.

Read Misses

- Read miss
  - When a read miss happens, the block containing the word is loaded into the cache and then the desired word is sent to the CPU.
- Load-through (early restart)
  - The desired word may be sent to the CPU as soon as it is read from the MM.
  - Reduces CPU's waiting time, but additional circuitry needed.
- Valid bit
  - If a location which is currently cached is modified in the main memory by an action which bypasses the CPU (e.g., DMA), a valid bit for the corresponding cache block is cleared.
  - The cache controller treats access to this location as a *cache miss*.
  - Valid bits are set to 0 on power on.

Note: DMA is a technique for moving data between memory and secondary storage or I/O devices where the data transfer is managed by a separate hardware called DMA controller rather than through repeated LDR-STR by the processor.

Handling Writes

- Write through
  - The cache and MM locations are simultaneously updated.
  - Simple, but results in unnecessary write operations in MM when cache is updated several times.
- Write-back
  - Update only the cache location and mark is ad updated with an associated flag bit, often called as *dirty* or modified bit.
  - The MM word is updated later, then the block containing the word is removed from the cache by a replacement algorithm.
  - May also lead to unnecessary write operations - when cache block is written back to the memory, all the words of the block are written back, even if only a single word in that block was modified when it was in the cache.

Mapping Techniques

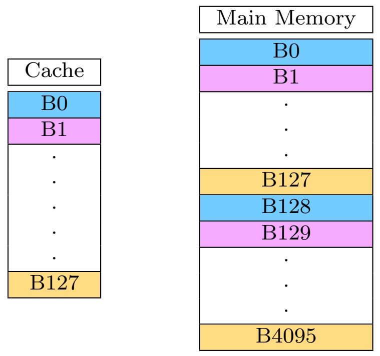
There are 3 different mapping techniques that are followed in practice:

- Direct mapping
- Associative mapping
- Set-associative mapping

The following example is used to illustrate the mapping algorithms:

- The cache consists of 128 blocks of 16 words each, which is a total of 2048 words.
- Assume that the MM is addressable by a 16-bit word address (not byte address, for simplicity).
- MM has 64k words, which we will view as 4k blocks of 16 words each.

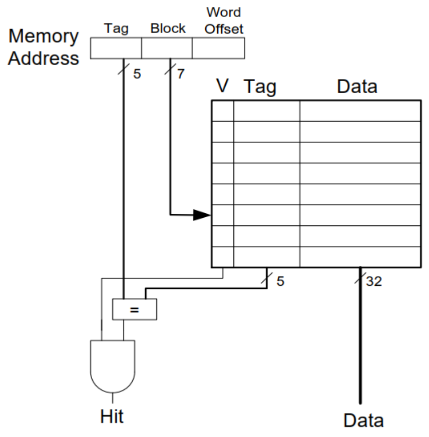
Direct Mapping



Block i of MM $\longrightarrow$ Block i mod 128 of cache
Address generated by CPU:

Tag	Block	Word
5 bits	7 bits	4 bits
16 bits		

- High order: which of the 32 blocks (4k/128 = 32) from MM is currently residing in the cache block
- Middle order: block number in the cache
- Lower order: select a word within the block



- Note that the tag field in the above example is nothing but the higher order 5 bits of the word address
- These 5 bits are stored along with that block in the cache
- The tag field can be used to determine whether the block at this location is the required block – the tag field is unique for each block from MM which can be mapped to the same block in the cache
- Note that even when the cache is not full, contention may arise for a location
- In this case, the replacement algorithm is trivial (a main memory block is mapped to a unique cache block)

Associative Mapping

A block of MM can be placed anywhere else.
Address generated by CPU:

Tag	Word
12 bits	4 bits
16 bits	

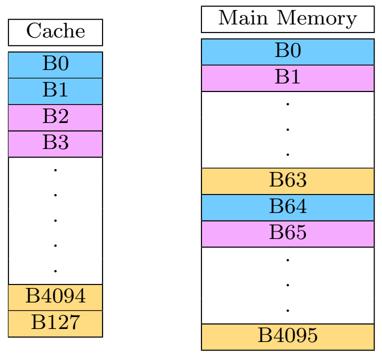
- When the request arrives, the tag field is compared for all the blocks in the cache to see if there is a match.
- Advantage:
  - Complete freedom in choosing where in the cache a particular block of memory is placed (i.e., cache space is utilized more efficiently).
- Disadvantage:
  - Search 128 blocks to match for a single tag. This comparison has to be done for every memory access.
  - Parallel search schemes can be used.
  - Costly and difficult to achieve high speeds.
- The replacement follows one of the standard techniques such as LRU, FIFO, etc.

Set-Associative Mapping

Blocks of cache are grouped into sets, and the mapping allows a block of the MM to reside in any block within a specific set (there is associativity within a set)

- The contention problem of the direct method is eased by having a few choices for block placement
- The hardware cost is reduced and speed is increased by decreasing the size of the associative search procedure
- If there are N blocks per set, the memory is called N-way set associative

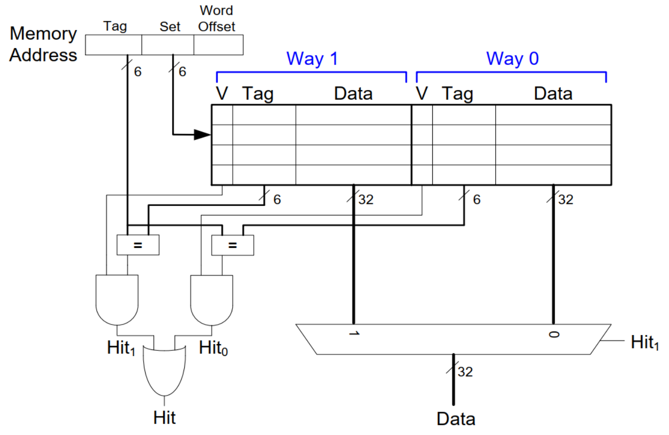
2-Way Set-Associative Mapping



- Suppose if we allow two blocks per set in the cache. The memory blocks 0, 64, 128, . . . , 4032 map into cache set 0, and they can occupy either of the two block positions within the set
- With 128 cache blocks and 2 blocks per set, we have 64 sets → we need 6 bits to identify the right set and 4 bits for a word, leaving 6 bits for the Tag field (which makes sense as each cache block can be from any of the 4096/64 = 64 = 2^6 MM blocks)

Address generated by CPU:

Tag	Set	Word
6 bits	6 bits	4 bits
16 bits		

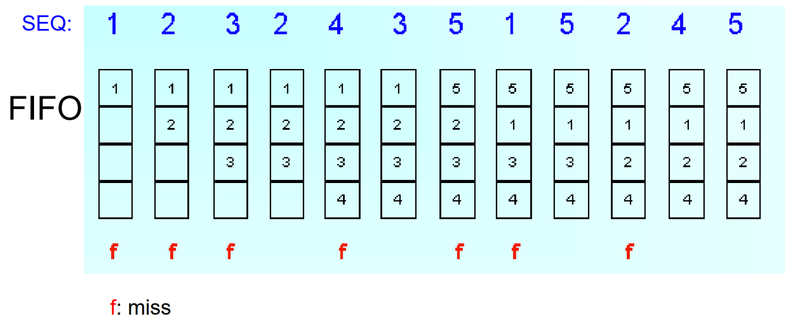


Replacement Algorithms

- First-In-First-Out
  - Replace the oldest block in the memory
- Least Recently Used (LRU)
  - Replace the block that has not been referenced for a long time
- Optimal Algorithm: (Ideal - assumes knowing the future)
  - Replace the block that will not be used for a longest period of time
  - Cannot be implemented in practice, used only for analysis purpose

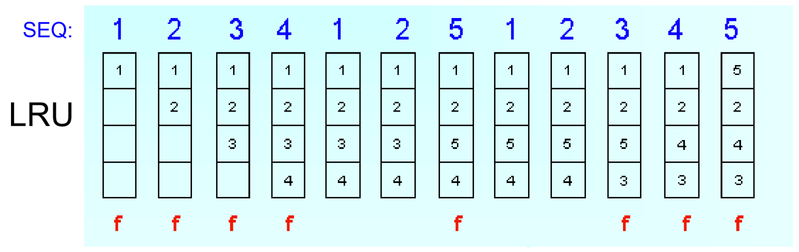
FIFO

- In the example, assume that the cache is fully associative and has 4 blocks
- FIFO works well if the access follows a sequential pattern (arrays etc.)

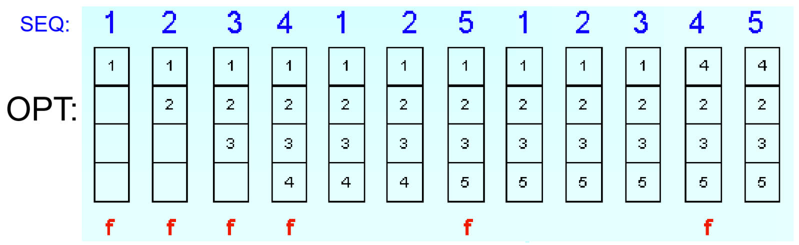


LRU

- It makes sense to overwrite a block that resided in the cache for a long time (LRU block) without being referenced
  - Temporal locality of reference



Optimal

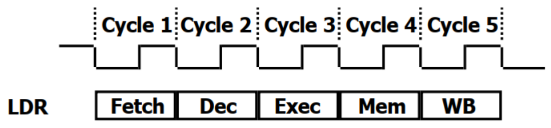


Pipelining Basics

Pipelining: start the next instruction before the current one has completed.

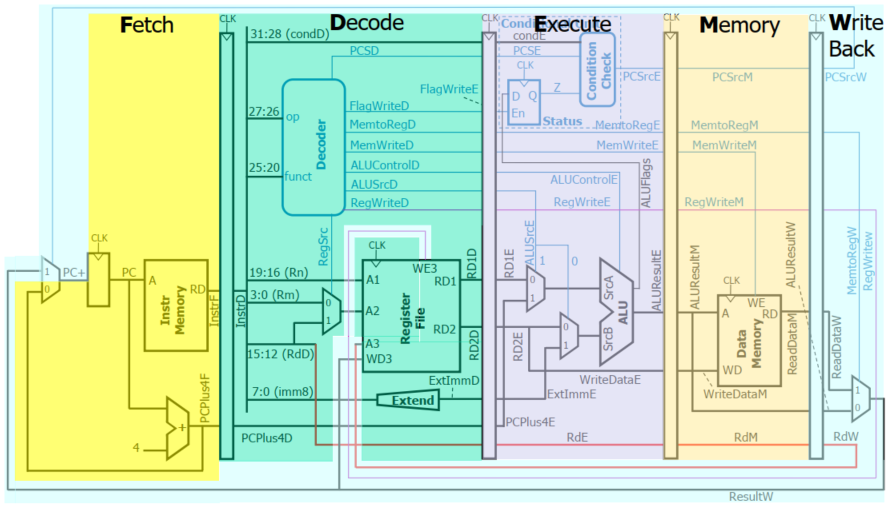
Under ideal conditions and with a large number of instructions, the speedup from pipelining is approximately equal to the number of pipe stages.

The five stages of load instruction



- Fetch: Instruction fetch and update PC.
- Decode: Registers fetch and instruction decode.
- Execute: Execute DP-type, calculate memory address
- Memory: Read/Write the data from/to the data memory.
- WriteBack: Write the result data into the register file.

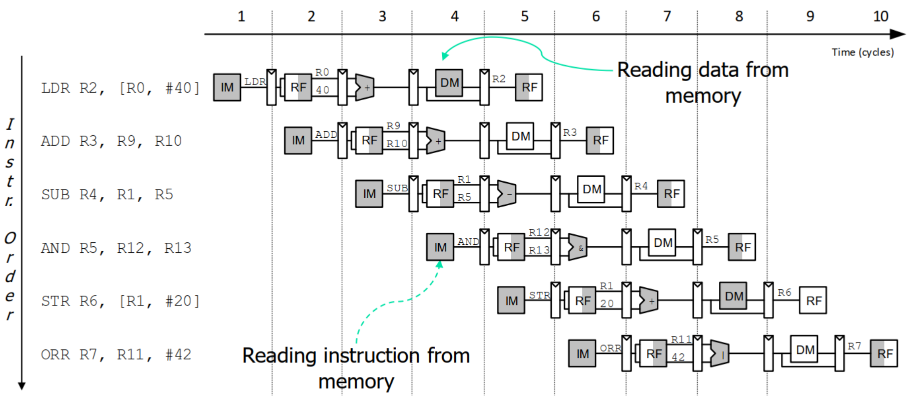
Pipeline Stages



Pipeline Hazards

- Structural hazards: attempt to use the same resource by two different instructions at the same time
- Data hazards: attempt to use data before it is ready
  - An instruction's source operand(s) are produced by a prior instruction still in the pipeline
- Control hazards: attempt to make a decision about program control flow before the condition has been evaluated and the new PC target address calculated
  - Branch instructions, writes to R15, interrupts/exceptions

Structural Hazards



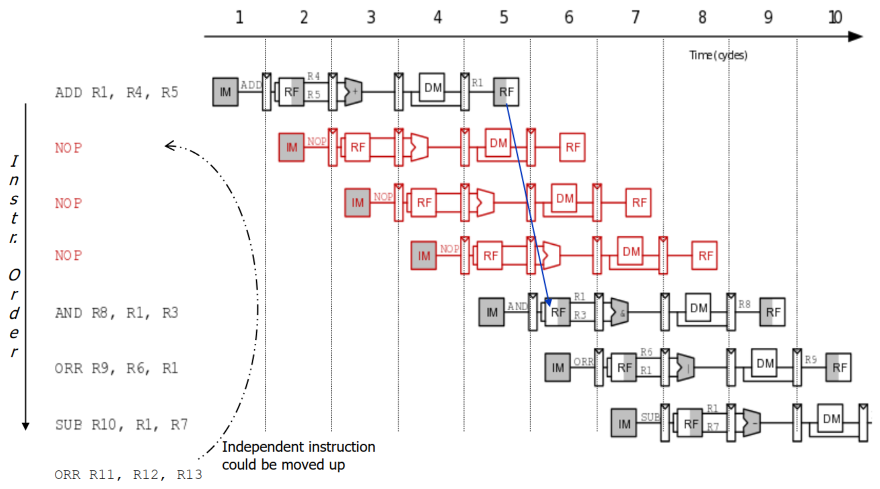
- Fix with separate instruction and data memories (IM and DM), or at least separate caches.

Data Hazards

- Also known as RAW (read after write) hazard or true data dependency
- Occurs very frequently in practice → represents the flow of information in the program

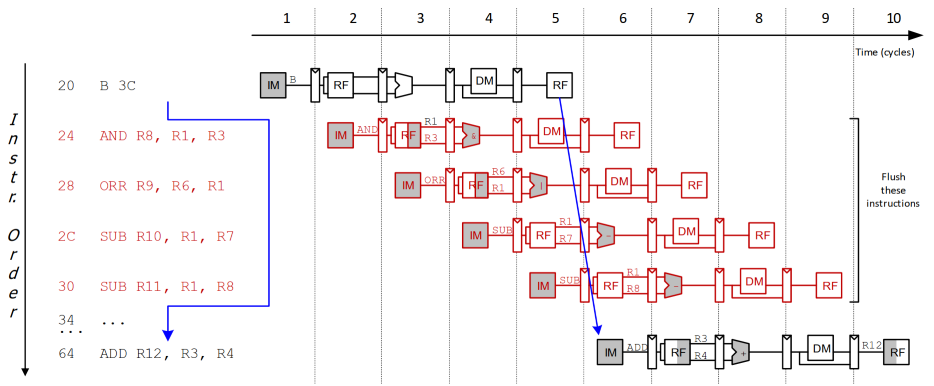
How to handle?

- Insert NOPs (MOV R0, R0) in code at compile time
  - Insert enough NOPs for result to be ready
  - NOPs waste time ⇒ No useful work is done
  - NOPs waste code memory ⇒ makes the code bulkier / bloated
  - Compiler needs to know the microarchitecture to know the number of NOPs ⇒ code is not very portable



- Rearrange code at compile time
  - Move independent useful instructions forward
  - Might not be possible all the time
- Stall the processor at run time
  - Impacts performance, needs additional hardware
- Forward data at run time
  - Good performance, needs even more hardware

Control Hazards



- Branch instructions, writes to R15 can cause control hazards
- Control hazards occur less frequently than data hazards, but are (much) harder to deal with than data hazards

How to handle?

- Compile time NOPs (4 NOPs after the branch in this case) can help, but
  - NOPs reduce performance, even when the branch is not taken
  - Increases the code size and makes the code microarchitecture dependent
- Possible approaches (all need additional hardware)
  - Stall until branch decision and BTA are available (performance ↓)
  - Early BTA - move decision point as early in the pipeline as possible (rather than in WB stage), thereby reducing the number of stall cycles
  - Branch prediction (a form of speculative execution)
    - ∗ Guess the branch outcome as well as the BTA
    - ∗ Start execution from BTA if branch is predicted to be taken, else next instruction.
    - ∗ If prediction is correct, no penalty associated with branches!
    - ∗ If prediction is wrong, need to restart the pipeline and discard the results from the speculatively executed instructions
  - Predicated/conditional execution (IT statements can help eliminate some branches)
  - Fine grained multithreading, . . . , . . .