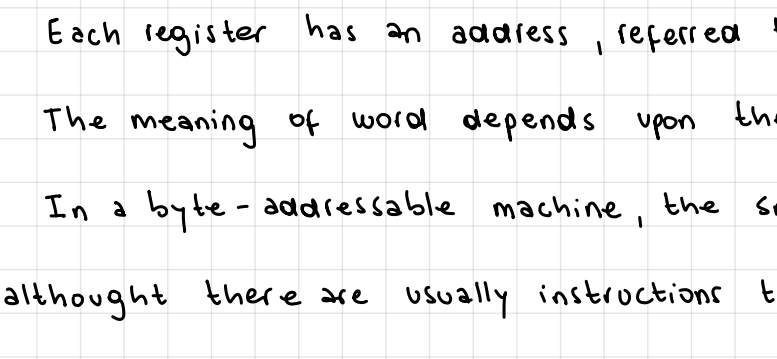
CHAPTER 4: MACHINE LANGUAGE AND ASSEMBLY LANGUAGE

- **ISA (Instruction Set Architecture):** The ISA view corresponds to the Assembly language / Machine code level. It is between the High level language (none of the machine hardware is visible or of concern) and the Control level (machine instructions are interpreted as register transfer actions) at the Functional Unit level.

HARDWARE COMPONENTS OF THE ISA

The CPU interacts with its internal main memory and perform input and output with the outside world

THE SYSTEM BUS MODEL:



The CPU is interconnected with its memory and I/O systems via shared system bus.

The CPU generates addresses that are placed onto the address bus, and the memory receives addresses from the address bus.

User writes a high level program → A compiler translates it into assembly language → An assembler translates the assembly language into machine code → The machine code is stored on disk → The machine code is loaded from disk into the main memory → The program is executed

During program execution, each instruction is brought into the ALU from the memory (one instruction at a time). The output of the program is placed, by a control unit, on a device such as video display or a disk. Communication among the three components (CPU, Memory, I/O) is handled with buses.

MEMORY:

Computer memory consists of a collection of consecutively numbered (addresed) registers, each one of wich normally holds one byte

Each register has an address, referred to as memory location.

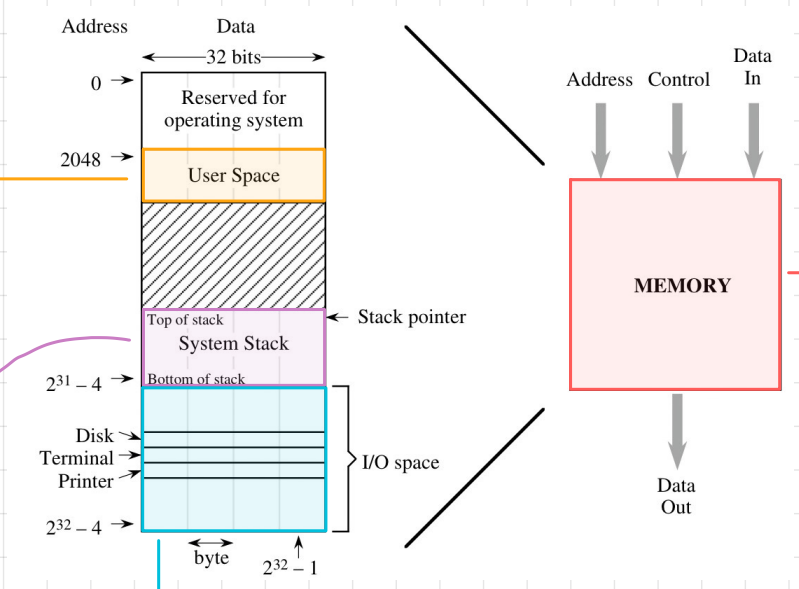
The meaning of word depends upon the particular processor (typical are 16, 32, 64 and 128 bits)

In a byte-addressable machine, the smallest object that can be referenced in memory is the byte, althought there are usually instructions that read and write multi-byte words.

When multi-byte words are used, there are two types of orders in wich the bytes are stored:

- **BIG-ENDIAN.** Most significant byte at lowest address.
- **LITTLE-ENDIAN:** least significant byte at lowest address.

The highest address is one less than the size of the memory.



Where a user's assembled program is loaded and can grow during operation from location until 2048 until it meets up with the system stack

Grows toward lower addresses

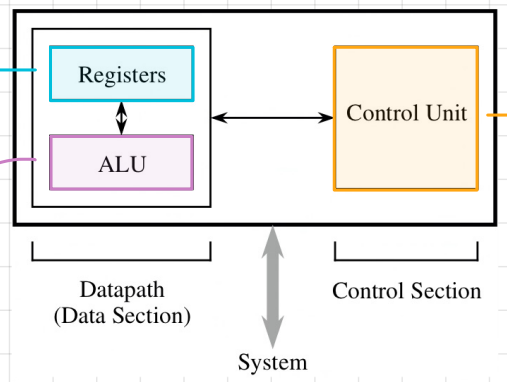
Used for reading and writing devices

Organized in a single address space (numerical range of memory addresses to wich the CPU can refer). A memory address that is n bits wide can specify one of 2^n items

THE CPU :

Consists of a **data section** (that contains registers and an ALU) and a **control section** (wich interprets instructions and effects register transfers)

Small fast memory, separate from the system memory used for temporary storage during computation



Arithmetic and logic Unit

Executes the program instructions wich are stored in the main memory by the fetch-execute cycle :

1. Fetch the next instruction to be executed from memory.
2. Decode the opcode
3. Read operand (s) from main memory, if any.
4. Execute the instructions and store results.
5. Go to step 1.

There are two registers that form the interface between the control unit and the data unit known as the **program counter** (PC) and the **instruction register** (IR).

- program counter → contains the address of the instruction being executed
- instruction register → the instruction that is pointed to by the PC is fetched from the memory and is stored in the IR where it's interpreted

The ALU implements a variety of binary (two-operand) and unary (one-operand) operations. Operations and operands to be used during the operations are selected by the Control Unit. The two source operands are fetched from the register file onto buses labeled "Register Source 1 (rs1)" and "Register Source 2 (rs2)" The output from the ALU is placed on the bus labeled "Register Destination (rd)" where the results are conveyed back to the register file.

- THE INSTRUCTION SET : The collection of instructions that a processor can execute. It defines it
- COMPILER : Computer program that transforms programs written in a high-level language into machine language.

In the process of **compiling** a program (translation process), a high-level source program is transformed into **assembly language**, and the assembly language is then translated into machine code for the target machine by an assembler. It takes place at **compile time** and **assembly time** respectively.
The resulting object program can be linked with other object programs at **link time**. The linked program (usually stored on a disk) is loaded into main memory at **load time** and executed by the CPU at **run time**.

ARC : A RISC COMPUTER

- SPARC (Scalable Processor Architecture) : Processor that was developed at Sun Microsystems in the mid-1980's.
- ARC (A RISC Computer) : Has most of the important features of the SPARC architecture, but without some of the more complex features that are present in a commercial processor.

ARC MEMORY.

The ARC is a 32-bit machine with byte-addressable memory.

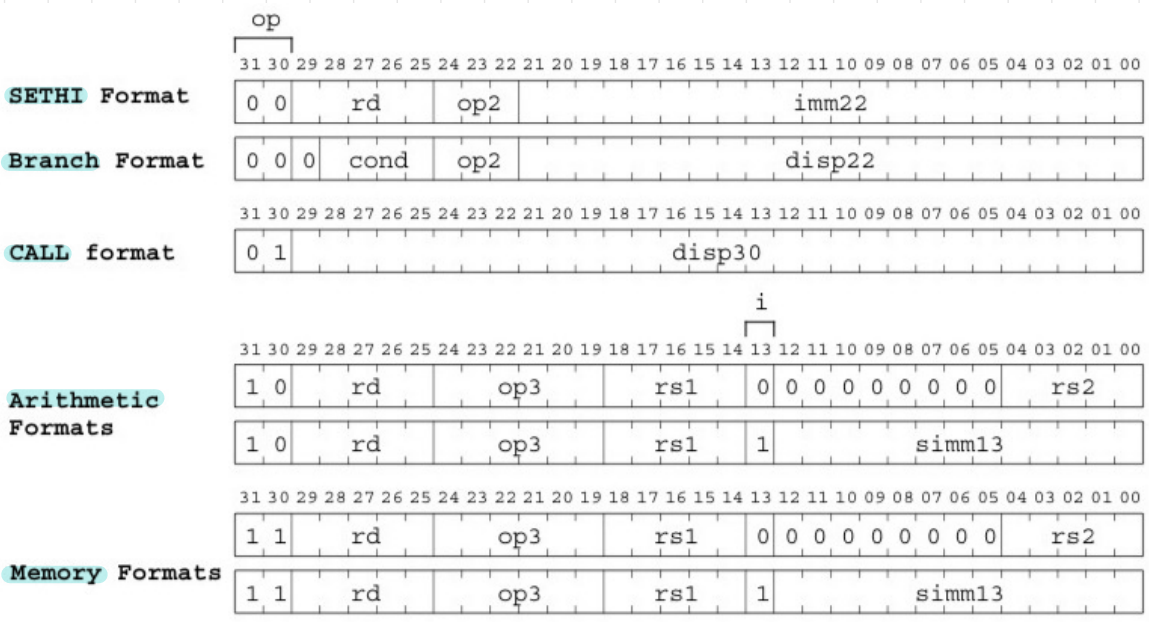
Each integer is stored in memory as a collection of four bytes. ARC is a big-endian architecture. The largest possible byte address in the ARC is $2^{32}-1$, so the address of the highest word in the memory map is three bytes lower than this, or $2^{32}-4$.

ARC INSTRUCTION SET:

- The ARC has 32 32-bit general-purpose registers, as well as PC and an IR.
- There is a **Processor Status Register** (PSR) that contains information about the state of the processor, including information about the results of arithmetic operations. The "arithmetic flags" in the PSR are called the **condition codes**. They specify weather a specified arithmetic operation resulted in a zero value (z), a negative value (n), a carry out from the 32-bit ALU (c), and an overflow (v) wich is set when the results of the arithmetic operation are too large to be handled by the ALU.
- All instructions are one word (32 bits) in size.
- The ARC is a **load-store** machine: the only allowable memory access operations load a value into one of the registers, or store a value contained in one of the registers into a memory location. All arithmetic operations operate on values that are contained in registers, and the results are placed in a register. Each instruction is represented by a **mnemonic**, wich is a name that represents the instruction.

ARC INSTRUCTION FORMATS:

The **instruction format** defines how the various bit fields of an instruction are laid out by the assembler, and how they are interpreted by the ARC control unit. The five formats are:



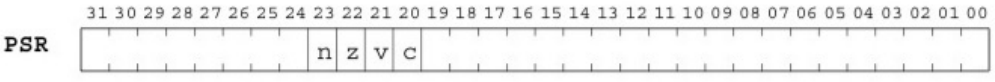
op	Format
00	SETHI/Branch
01	CALL
10	Arithmetic
11	Memory

op2	Inst.
010	branch
100	sethi

op3 (op=10)	
010000	addcc
010001	andcc
010010	orcc
010110	orncc
100110	srl
111000	jmpl

op3 (op=11)	
000000	ld
000100	st

cond	branch
0001	be
0101	bcs
0110	bneg
0111	bvs
1000	ba



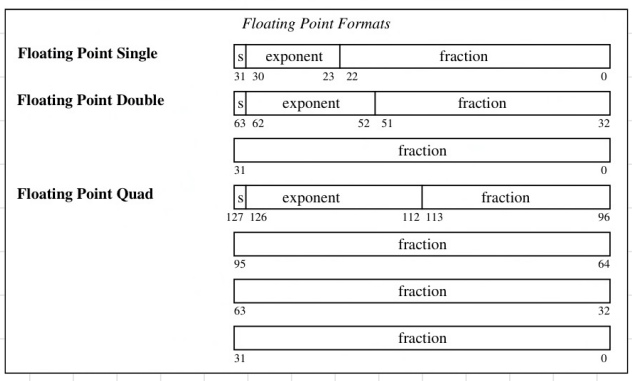
- The **imm22** and **disp22** fields each hold a 22-bit constant that is used as the operand for the SETHI format (for imm22) or for calculating a displacement for a branch address (for a disp22).
- The **disp30** field contains a 30-bit displacement that is used in calculating the address of the called routine.
- The **simm13** field is a 13-bit immediate value that is sign extended to 32 bits for the second source when the **i** (immediate) field is 1.

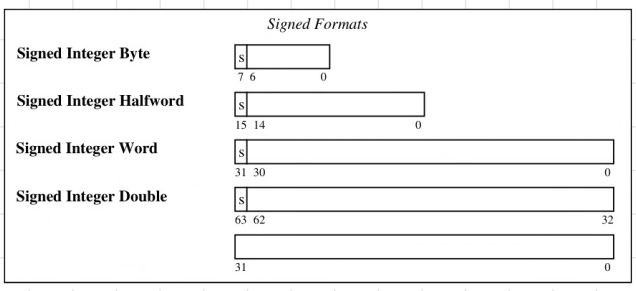
ARC DATA FORMATS:

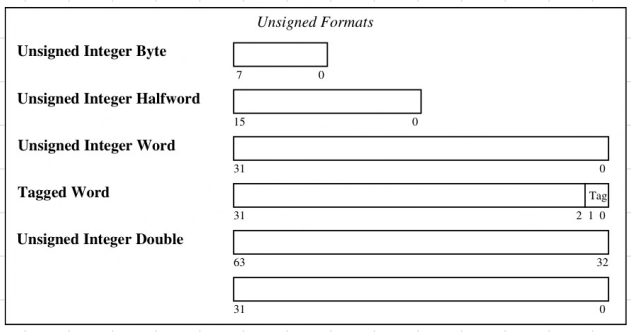
The ARC supports 12 different data formats grouped into: signed integer, unsigned integer and floating point.

Allowable formats widths are: **byte** (8b), **half-word** (16b), **word** / **single-word** (32b), **tagged word** (32b: 30 most significant from the value and 2 least from a tag), **doubleword** (64b) and **quadword** (128b).

Unsigned and signed integers







are both stored and manipulated as 2's complement integers but it's their interpretation that varies. One subset of the branch instructions assumes that the value(s) being compared are signed integers while the other assumes they are unsigned, so then the c bit indicates unsigned integer overflow and the v bit signed overflow.

The tagged word uses the 2 least significant bits to indicate overflow in wich an attempt is made to store a value that is larger than 30 bits into the allocated 30b of the 32b word.

ARC INSTRUCTION DESCRIPTION.

- ld : ld [x], %r1

  Copy the contents of memory location x into register %r1.

- st : st %r1, [x]

  Copy the contents of register %r1 into memory location x.

- sethi : sethi 0x304f15, %r1

  Set the high 22 bits of %r1 to $304f15|_{16}$ and set the low 10 bits to zero.

- andcc : andcc %r1, %r2, %r3

  Logically AND %r1 and %r2 and place the result in %r3

- orcc : orcc %r1, 1, %r1

  Set the least significant bit of %r1 to 1.

- orncc : orncc %r1, %r0, %r1

  Complement %r1.

- srl : srl %r1, 3, %r2

  Shift %r1 right by three bits and store it in %r2. Zeros are copied into the three most significant bits of %r2.

- addcc : addcc %r1, 5, %r1

  Add 5 to %r1.

- call : call sub_r

  Call a subroutine that begins at location sub_r.

- jmpl : jmpl %r15+4, %r0

  Return from subroutine. The value of the PC for the call instruction was previously saved in %r15 and so the return address should be computed for the instruction that follows the call, at %r15+4. The current address is discarded in %r0.

- be be label

  Branch to label if z=1.

- bneg : bneg label

  Branch to label if n=1

- bcs : bcs label

  Branch to label if c=1.

- **bvs** : bvs label

  Branch to label if V=1.

- **ba** : ba label

  Branch to label regardless of the settings of the condition codes.

PSEUDO-OPS

Pseudo-Op	Usage	Meaning
.equ X	.equ #10	Treat symbol X as $(10)_{16}$
.begin	.begin	Start assembling
.end	.end	Stop assembling
.org	.org 2048	Change location counter to 2048
.dwb	.dwb 25	Reserve a block of 25 words
.global	.global Y	Y is used in another module
.extern	.extern Z	Z is defined in another module
.macro	.macro M a, b, ...	Define macro M with formal parameters a, b, ...
.endmacro	.endmacro	End of macro definition
.if	.if <cond>	Assemble if <cond> is true
.endif	.endif	End of .if construct

Not opcodes but instructions to the assembler to perform some action at assembly time

Unlike the processor opcodes (specific to a given machine), pseudo-ops are specific to a given assembler because they are executed by the assembler itself.

SUBROUTINE LINKAGE AND STACKS

A subroutine (AKA function or procedure) is a sequence of instructions invoked in a manner that makes it appear to be a single instruction in a high level view. When a program calls it, control is passed from the program to the subroutine wich then returns to the location just past wher it was called.

The methods for passing arguments to and from the called routine are called calling conventions and the process of passing arguments between routines is refeired to as subroutine linkage.

CALLING CONVENTIONS:

1. Using Registers: Places the arguments in registers.
2. Using Data link area: The address of the data link area is passed in a register to the called routine.
3. Using a Stack: The calling routine pushes all of its arguments onto a LIFO stack. The called routine then pops the passed arguments from the stack, and pushes any return values onto the stack. The calling routine then retrieves the return value(s) from the stack and continues execution.

CHAPTER 5: LANGUAGES AND THE MACHINE

THE COMPILATION PROCESS

THE STEPS OF COMPILATION:

- Lexical Analysis: Reduce the program text to the basic symbols of the language.
- Syntactic Analysis: Parse the symbols to recognize the underlying program structure.
- Name Analysis: Associate the names with program variables and then with memory locations.
- Type Analysis: Determine the type of all data items.
- Action Mapping and Code Generation: Associate program statements with their assembly language sequence.
- Additional Steps: Allocate variables to registers, track register usage, optimize the program.

THE COMPILER MAPPING SPECIFICATION:

It's the information about the particular ISA for the compiler.

HOW THE COMPILER MAPS THE THREE INSTRUCTIONS CLASSES INTO ASSEMBLY CODE:

Only global variables (static variables) have addresses that are known at compile time.

local variables (automatic variables) only come to existance when the function or block is entered and they disappear when the function or block is exited for the last time.

DATA MOVEMENT:

① Structures

```
struct point {
   int x;
   int y;
   int z;
}

struct point pt;
```

The compiler would lay out this structure in memory as three consecutive memory locations.

The x component would be located at address pt, the y at address pt+4 and the z at pt+8.

② Arrays

```
int A[10];
```

The array index can be computed at run time.

The general expression for computing the machine address of an array element at run time is given by: Element Address = BASE + (INDEX - START) * SIZE

- BASE: Starting address of the array
- INDEX: Index of the desired element
- START: starting index of the array
- SIZE: size of an individual element in bytes

```
sub  %l3, %l4, %l6     ! %l6 ← INDEX - START
SLL  %l6, 2, %l6       ! %l6 * 4
ld   [A + %l6], %f1    ! array value
```

It costs more instructions if SIZE ≠ power of 2

ARITHMETIC INSTRUCTIONS:

When the compiler finds an arithmetic instruction that requiers more registers than are available in the machine, it must temporarily store variables on the stack, called register spill.

To decide when the value in a register is no longer needed to be stored, compilers use a technique known as live-dead analysis.

THE ASSEMBLY PROCESS

Commercial assemblers provide at least the following capabilities:

- Allow the programmer to specify the run-time location of data values and programs.

- Provide a means for the programmer to initialize data values in memory prior to program execution.
- Provide assembly-language mnemonics for all machine instructions and addressing modes, and translate valid assembly language statements into their equivalent machine language binary values.
- Permit the use of symbolic labels to represent address and constants.
- Provide a means for the programmer to specify the starting address of the program, if there is one.
- Provide a degree of assemblee-time arithmetic.
- Include a mechanism that allows variables to be defined in one assembly language program and used in another, separately assembled program.
- Provide for the expansion of macro routines (routines that can be defined once and then instantiated as many times as needed).

ASSEMBLY AND TWO PASS ASSEMBLERS:

The first pass is dedicated to determining the address of all data items and machine instructions, and selecting wich machine instruction should be produced for each assembly language instructions.

The address of data items and instructions are determined by the location counter wich keeps track of the address of the current instruction or data item as assembly proceeds. The .org pseudo-op sets the location counter to the value specified by current statement.

During this pass the assembler also inserts the definitions of all labels and constant values into a table, referred as the symbol table.

On the second pass the assembler allow symbols to be used in the program before they are defined, known as forward referencing, generating the machine code and inserting the values of symbols.

FINAL TASKS OF THE ASSEMBLER:

After assembly is complete the assembler must add additional information for the linker and loader:

- The module name and size.
- The address of the start symbol.
- The addresses of global symbols and information about wich symbols remain undefined in the module because they are defined as global in another.
- Information about any library routines that are referenced by the module.
- The values of any constants that are to be loaded into memory.
- Relocation information.

LINKING AND LOADING

A linkage editor (linker) is a software program that combines separately assembled programs (object modules) into a single program (load module).

LINKING:

The linker must:

- Resolve address references that are external to modules as it links them.
- Relocate each module by combining them end-to-end as appropiate.
- Specify the starting symbol of the load module.

- If the memory model includes more than one memory segment, the linker must specify the identities and contents of the various segments.

In resolving address references the linker needs to distinguish local from global symbol names: the .global pseudo-op instructs the assembler to mark a symbol as being available to other object modules during linking phase; while the .extern identifies a label that is used in one module but is defined in another.

The assembler marks symbols that may have their address changed during linking as relocatable. Absolute numbers (marked by .equ or that appear in memory locations) are not relocatable. Memory locations that are positioned relative to a .org statement are generally relocatable.

LOADING:

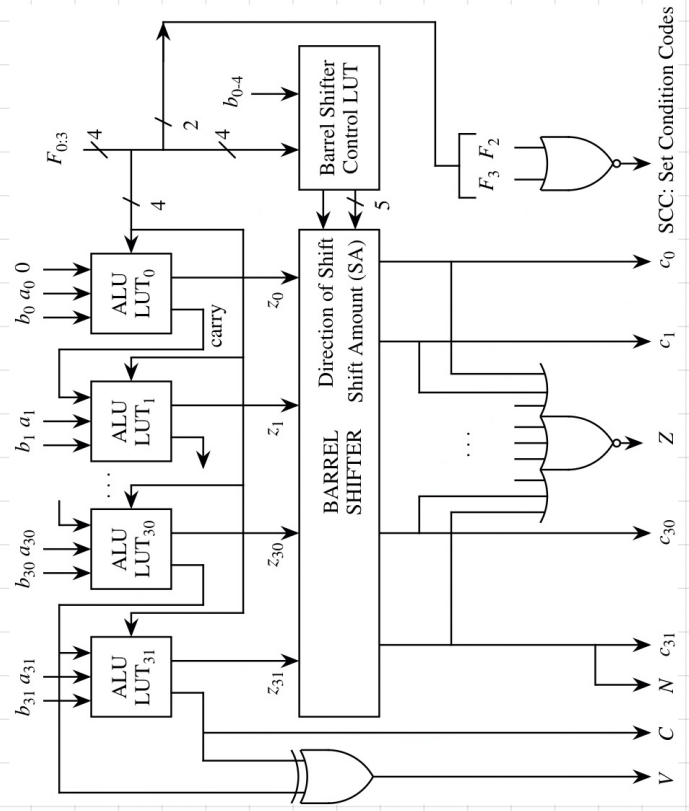
The loader is a software program that places the load module into main memory and intializes the stack pointer (%.sp) and the program counter (%.pc) to their intial values.

If there are several programs in memory at any time, the loader must relocate this modules at load time by adding an offset to all of the relocatable code in a module. This kind of loader is known as relocating loader. It doesn't simply repeat the job of the linker, wich has to combine several object modules into a single load module, but it modifies relocatable addresses within a single load module so that several programs can reside in memory simultaneously. A linking loader performs both the linking process and the loading process.

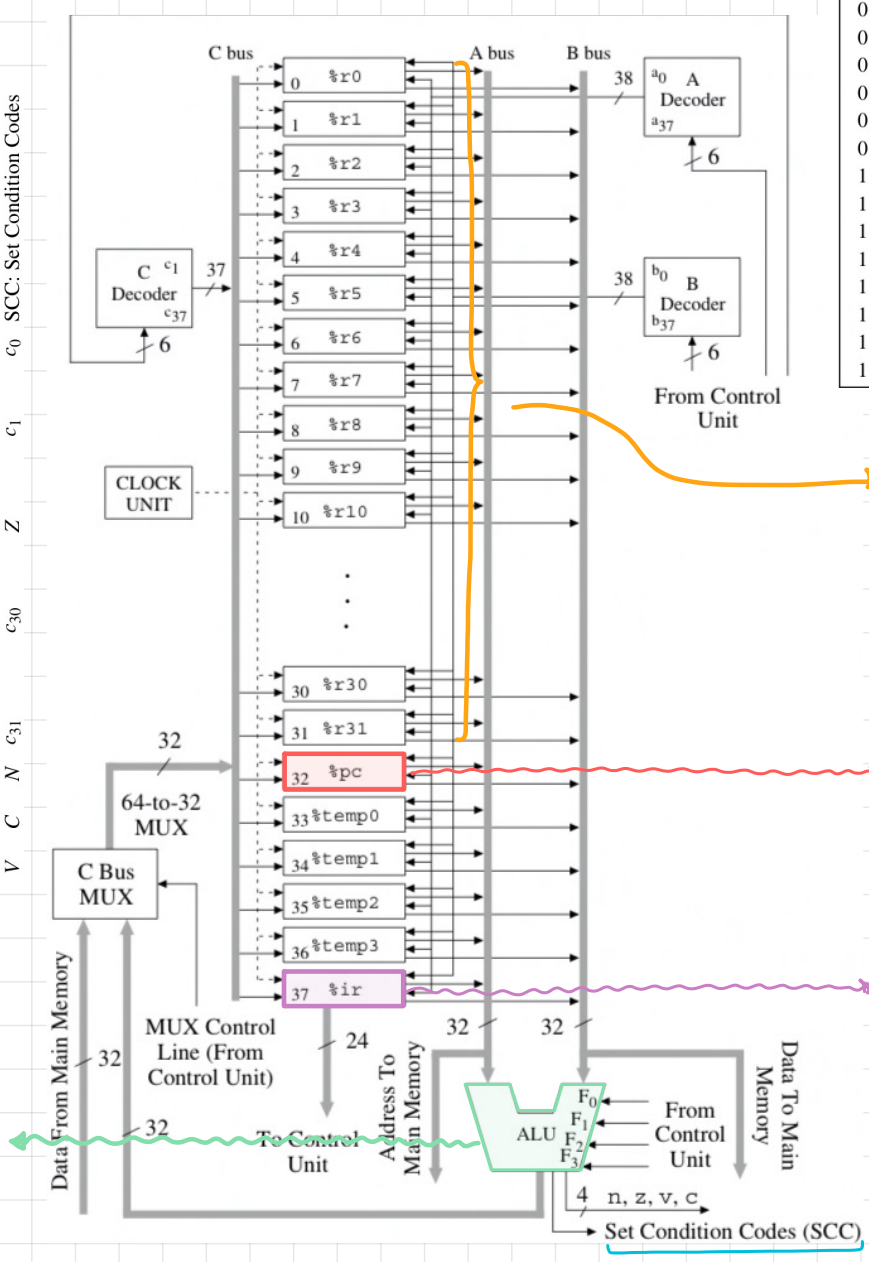
CHAPTER 6: DATAPATH AND CONTROL

A MICROARCHITECTURE FOR THE ARC

THE DATAPATH:



Performs one of 16 operations on the A and B buses. The 32b result is placed on the C bus, unless it's blocked CMux when a word of memory is placed on the C bus instead.



F_3 F_2 F_1 F_0	Operation	Changes Condition Codes
0 0 0 0	ANDCC (A, B)	yes
0 0 0 1	ORCC (A, B)	yes
0 0 1 0	NORCC (A, B)	yes
0 0 1 1	ADDCC (A, B)	yes
0 1 0 0	SRL (A, B)	no
0 1 0 1	AND (A, B)	no
0 1 1 0	OR (A, B)	no
0 1 1 1	NOR (A, B)	no
1 0 0 0	ADD (A, B)	no
1 0 0 1	LSHIFT2 (A)	no
1 0 1 0	LSHIFT10 (A)	no
1 0 1 1	SIMM13 (A)	no
1 1 0 0	SEXT13 (A)	no
1 1 0 1	INC (A)	no
1 1 1 0	INCPC (A)	no
1 1 1 1	RSHIFT5 (A)	no

directly accesible by a user

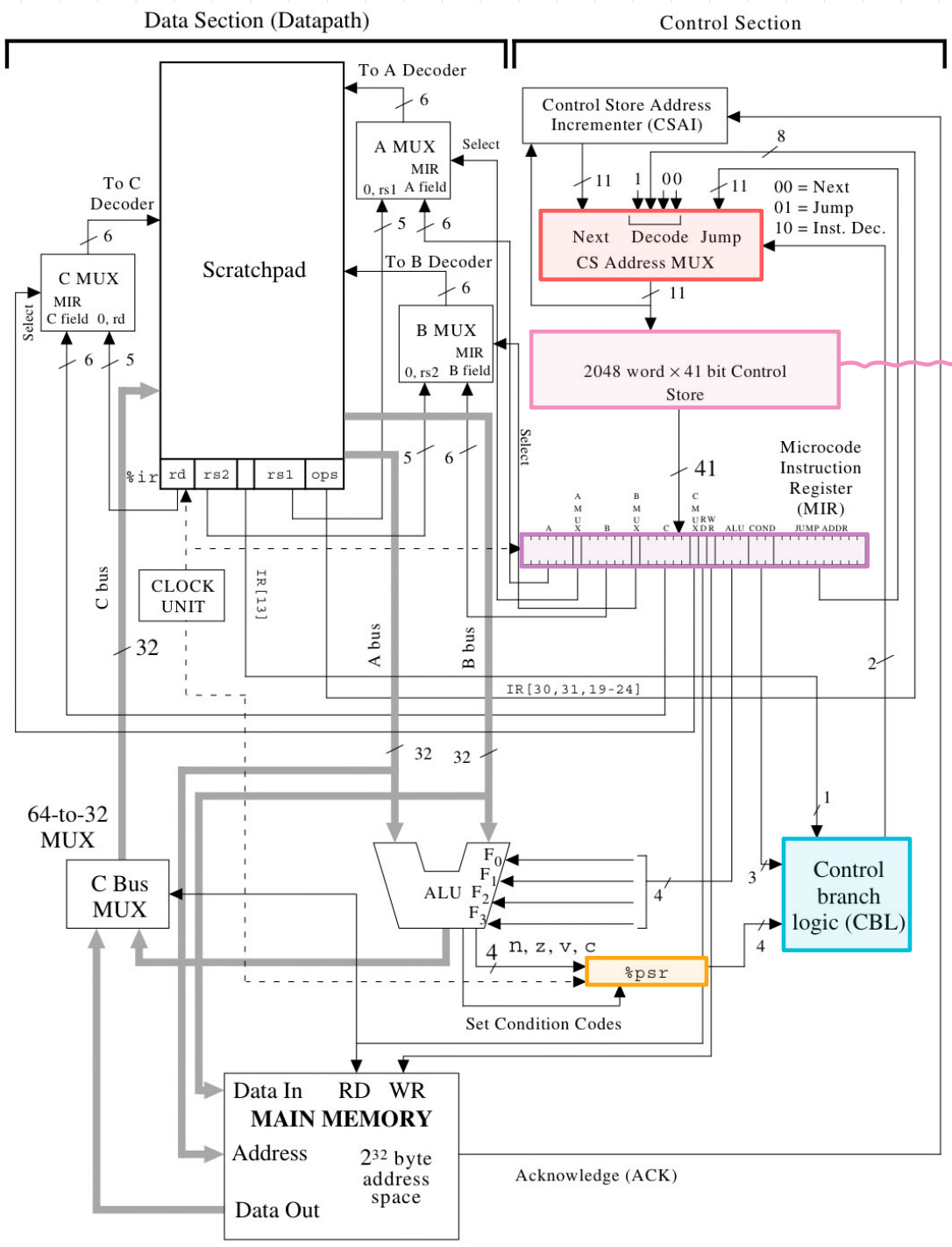
Program Counter: keeps track of the next instruction to be read from the main memory. Only accesible by the user through the call and jmpl.

Holds the current instruction that is being executed

Tells the %psr when to update the condition codes

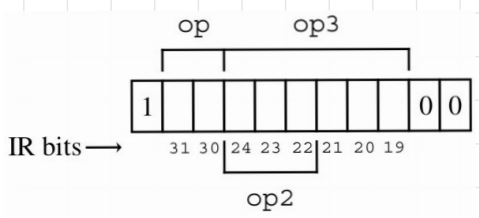
THE CONTROL SECTION:

When the micro-architecture begins operation, a reset circuit places the microword at location 0 in the control store into the MIR and executes it. From that point, a micro word is selected for execution from either the Next, the Decode or the Jump inputs to the CS Address MUX according to the settings in the COND field of the MIR and the output of the CBL logic. After each microword



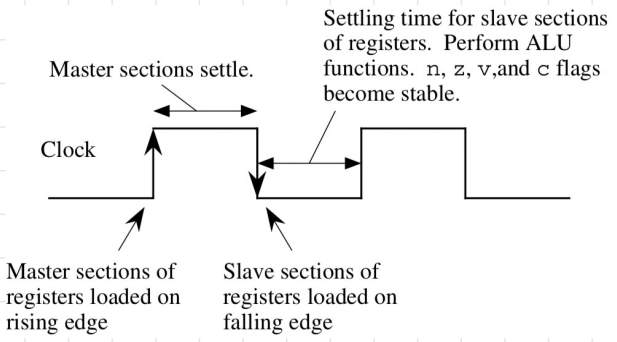
ROM (AKA Control Store). Each 41b word is called microinstruction and are fetched and executed by the ROM. This microinstruction execution is controlled by the MIR, the psr, and a mechanism for determining the next microinstruction to be executed: the CBL and the CS Address MUX.

C_2 C_1 C_0	Operation
0 0 0	Use NEXT ADDR
0 0 1	Use JUMP ADDR if n = 1
0 1 0	Use JUMP ADDR if z = 1
0 1 1	Use JUMP ADDR if v = 1
1 0 0	Use JUMP ADDR if c = 1
1 0 1	Use JUMP ADDR if IR[13] = 1
1 1 0	Use JUMP ADDR
1 1 1	DECODE



is placed in the MIR, the datapath performs operations according to the settings in the individual fields of the MIR.

TIMING:



While the clock is low, the ALU, CBL, and MUX functions are performed, wich settle in time for the rising edge of the clock. On the rising edge of the clock, the new values of the registers are written into the masters sections. The registers settle while the clock is high.

DEVELOPING THE MICROPROGRAM.

```
    0: R[ir] ← AND(R[pc],R[pc]); READ;
    1: DECODE;
       / sethi
 1152: R[rd] ← LSHIFT10(ir); GOTO 2047;
       / call
 1280: R[15] ← AND(R[pc],R[pc]);
 1281: R[temp0] ← ADD(R[ir],R[ir]);
 1282: R[temp0] ← ADD(R[temp0],R[temp0]);
 1283: R[pc] ← ADD(R[pc],R[temp0]);
       GOTO 0;
       / addcc
 1600: IF R[IR[13]] THEN GOTO 1602;
 1601: R[rd] ← ADDCC(R[rs1],R[rs2]);
       GOTO 2047;
 1602: R[temp0] ← SEXT13(R[ir]);
 1603: R[rd] ← ADDCC(R[rs1],R[temp0]);
       GOTO 2047;
       / andcc
 1604: IF R[IR[13]] THEN GOTO 1606;
 1605: R[rd] ← ANDCC(R[rs1],R[rs2]);
       GOTO 2047;
 1606: R[temp0] ← SIMM13(R[ir]);
 1607: R[rd] ← ANDCC(R[rs1],R[temp0]);
       GOTO 2047;
       / orcc
 1608: IF R[IR[13]] THEN GOTO 1610;
 1609: R[rd] ← ORCC(R[rs1],R[rs2]);
       GOTO 2047;
 1610: R[temp0] ← SIMM13(R[ir]);
 1611: R[rd] ← ORCC(R[rs1],R[temp0]);
       GOTO 2047;
       / orncc
 1624: IF R[IR[13]] THEN GOTO 1626;
 1625: R[rd] ← NORCC(R[rs1],R[rs2]);
       GOTO 2047;
 1626: R[temp0] ← SIMM13(R[ir]);
 1627: R[rd] ← NORCC(R[rs1],R[temp0]);
       GOTO 2047;
       / srl
 1688: IF R[IR[13]] THEN GOTO 1690;
 1689: R[rd] ← SRL(R[rs1],R[rs2]);
       GOTO 2047;
 1690: R[temp0] ← SIMM13(R[ir]);
 1691: R[rd] ← SRL(R[rs1],R[temp0]);
       GOTO 2047;
       / jmpl
 1760: IF R[IR[13]] THEN GOTO 1762;
 1761: R[pc] ← ADD(R[rs1],R[rs2]);
       GOTO 0;
 1762: R[temp0] ← SEXT13(R[ir]);
 1763: R[pc] ← ADD(R[rs1],R[temp0]);
       GOTO 0;
       / ld
 1792: R[temp0] ← ADD(R[rs1],R[rs2]);
       IF R[IR[13]] THEN GOTO 1794;
 1793: R[rd] ← AND(R[temp0],R[temp0]);
       READ; GOTO 2047;
 1794: R[temp0] ← SEXT13(R[ir]);
 1795: R[temp0] ← ADD(R[rs1],R[temp0]);
       GOTO 1793;
       / st
 1808: R[temp0] ← ADD(R[rs1],R[rs2]);
       IF R[IR[13]] THEN GOTO 1810;
 1809: R[ir] ← RSHIFT5(R[ir]); GOTO 40;
   40: R[ir] ← RSHIFT5(R[ir]);
   41: R[ir] ← RSHIFT5(R[ir]);
   42: R[ir] ← RSHIFT5(R[ir]);
   43: R[ir] ← RSHIFT5(R[ir]);
   44: R[0] ← AND(R[temp0], R[rs2]);
       WRITE; GOTO 2047;
 1810: R[temp0] ← SEXT13(R[ir]);
 1811: R[temp0] ← ADD(R[rs1],R[temp0]);
       GOTO 1809;
       / Branch instructions: ba, be, bcs,
 1088: GOTO 2;                  bvs, bneg
    2: R[temp0] ← LSHIFT10(R[ir]);
    3: R[temp0] ← RSHIFT5(R[temp0]);
    4: R[temp0] ← RSHIFT5(R[temp0]);
    5: R[ir] ← RSHIFT5(R[ir]);
    6: R[ir] ← RSHIFT5(R[ir]);
    7: R[ir] ← RSHIFT5(R[ir]);
    8: IF R[IR[13]] THEN GOTO 12;
       R[ir] ← ADD(R[ir],R[ir]);
    9: IF R[IR[13]] THEN GOTO 13;
       R[ir] ← ADD(R[ir],R[ir]);
   10: IF Z THEN GOTO 12;
       R[ir] ← ADD(R[ir],R[ir]);
   11: GOTO 2047;
   12: R[pc] ← ADD(R[pc],R[temp0]);
       GOTO 0;
   13: IF R[IR[13]] THEN GOTO 16;
       R[ir] ← ADD(R[ir],R[ir]);
   14: IF C THEN GOTO 12;
   15: GOTO 2047;
   16: IF R[IR[13]] THEN GOTO 19;
   17: IF N THEN GOTO 12;
   18: GOTO 2047;
   19: IF V THEN GOTO 12;
   20: GOTO 2047;
 2047: R[pc] ← INCPC(R[pc]); GOTO 0;
```

In a microprogrammed architecture, instructions are interpreted by the microprogram (often referred to as firmware because it bridges the gap between the hardware and the software) in the control store.

In the control store, each microstatement is stored in coded form in a single microword.

The first task in the execution of a user-level program is to bring the instruction pointed to by the PC from main memory into the IR so then the next step is to decode the opcode fields.

Additional decoding is needed for the branch instructions because the type of branch is determined by the COND field of the branch format (bits 25-28), wich is not used during a DECODE operation.

CHAPTER 7: MEMORY

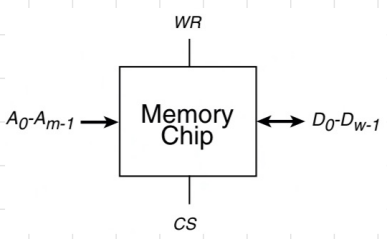
RANDOM ACCESS MEMORY (RAM)

Any memory location can be accessed in the same amount of time, regardless of its position in the memory.

- SRAM (Static RAM): RAM chips that are based upon flip-flops. The contents of each location persist as long as power is applied to the chips.
- DRAM (Dynamic RAM): RAM chips, employ a capacitor which stores a minute amount of electric charge, in which the charge level represents a 1 or a 0. Capacitors are much smaller than flip-flops, and so a DRAM can hold much more information in the same area than an SRAM. Since the charges on the capacitors dissipate with time, the charge in the capacitor storage cells in DRAMs must be restored or refreshed feqeuntly.

Although DRAMs are very economical, SRAMs offer greater speed. The refresh cycles, error detection circuitry, and the low operation powers of DRAMs create a speed difference that is 1/4 of SRAM speed, but SRAMs also incur a significant cost.

CHIP ORGANIZATION



The address lines $A_0 - A_{m-1}$ in the RAM chip contain an address, which is decoded from an m-bit address into one of 2^m locations within the chip, each of which has a w-bit word associated with it. The chip contains $2^m \times w$ b.

READ ONLY MEMORY (ROM)

for some applications, the program never changes, and so it is hardwired into a ROM.

- PROM (Programmable ROM): Allow their contents to be written by a user with a deviced called PROM burner. It allows the designer to delay decisions about what information is stored. It can only be written once, or can be rewritten only if the existing pattern is a subset of the new pattern.
- EPROM (Erasable ROM): Can be written several times, after being erased with ultraviolet light (UVPROMs).
- EEPROM (Electrically Erasable ROM). Allow their contents to be rewritten electrically.

PROMs will be used for control units and for arithmetic logic units (ALUs), generating a truth table with all possible combinations of operands and all combinations of functions, and send the truth table to a PROM burner wich loads it into the PROM This approach is called lookup table (LUT).

CACHE MEMORY

- LOCALITY PRINCIPLE: When a program executes on a computer, most of the memory references are made to a small number of locations.
- TEMPORAL LOCALITY: when a program references to a memory location, it is likely to reference that same memory location again soon.
- SPATIAL LOCALITY: A memory location that is near a recently referenced location is more likely to be referenced than a memory location that is farther away.

A small but fast cache memory (in which the contents of the most commonly accessed locations are

mantained) can be placed between the main memory and the CPU. When a program executes, the cache memory is searched first, and the referenced word is accessed in the cache if the word is present. If the referenced word is not in the cache, then a free location is created in the cache and the referenced word is brought into the cache from the main memory. The word is then accessed in the cache.

The cache is placed both physically and logically closer to the CPU than the main memory, avoiding communication delays over a shared bus.

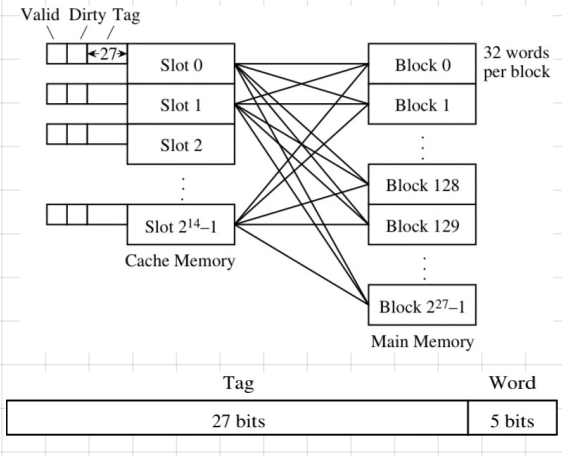
MAPPED CACHE:

To keep track of which one of the possible blocks is in each slot, a **tag** field is added to each slot. All the tags are stored in a special tag memory where they can be searched in parallel.

When a program is first loaded into main memory, the cache is cleared, and so while a program is executing, a **valid** bit is needed to indicate whether or not the slot holds a block that belongs to the program being executed. The **dirty** bit keeps track of whether or not a block has been modified while it is in the cache. A slot that is modified must be written back to the main memory before the slot is reused for another block.

A referenced location that is found in the cache results in a **hit**, otherwise, the result is a **miss**.

ASSOCIATIVE MAPPED CACHE.



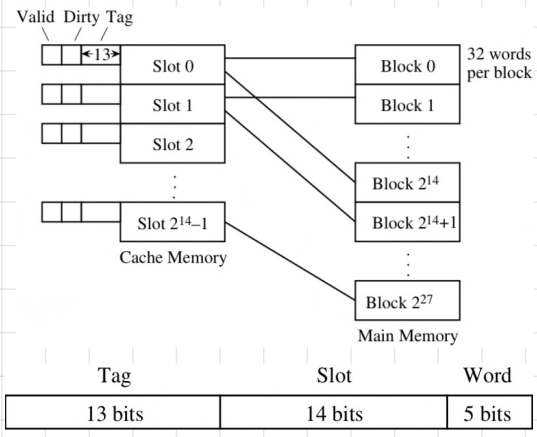
When a reference is made to a main memory address, the cache hardware intercepts the reference and searches the cache tag memory to see if the requested block is in the cache. For each slot, if the valid bit is 1, then the tag field of the referenced address is compared with the tag field of the slot. All of the tags are searched in parallel. If any tag in the cache tag memory matches the tag field of the memory reference, then the word is taken from the position in the slot specified by the word field. If the referenced word is not found in the cache, then the main memory block that contains the word is brought into the cache and the referenced word is then taken from the cache. The tag, valid and dirty fields are updated, and the program resumes execution.

When all of the valid bits are 1, one of the slots must be freed for a new block by any replacement police:

- ★ **Least Recently Used (LRU)**. A time stamp is added to each slot, which is updated when any slot is accessed. The last recently used slot is discarded when a slot must be freed for a new block.
- ★ **Firs In - First Out (FIFO)**. Replaces slots one after the next in the order of their physical locations in the cache.
- ★ **Least frequently Used (LFU)**: The least frequently used slot is freed. Only one slot is updated at a time by incrementing a frequency counter that is attached to each slot.
- ★ **Random**: Chooses a slot at random.

The LFU is only slightly better than the Random policy. The LRU can be implemented efficently.

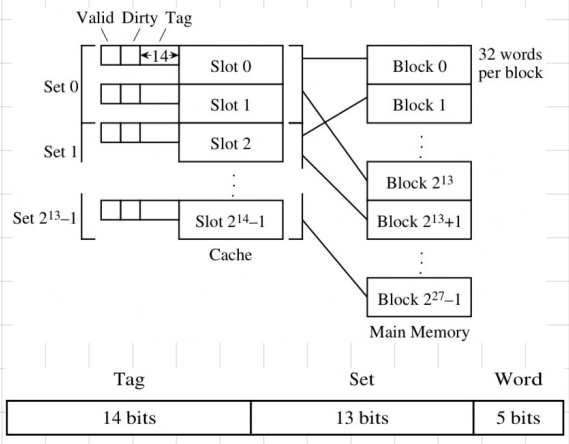
DIRECT MAPPED CACHE:



Each slot corresponds to an explicit set of main memory blocks.

When a reference is made to a main memory address, the slot fields identifies in which of all the slots the block will be found if its in the cache. If the valid bit is 1, then the tag field of the referenced address is compared with the tag field of the slot. If the tag fields are the same, then the word is taken from the position in the slot specified by the word field. If the valid bit is 1 but the tag fields are not the same, then the slot is written back to main memory if the dirty bit is set, and the corresponding main memory block is then read into the slot. for a program that has just started execution, the valid bit will be 0 and so the block is simply written to the slot. The valid bit for the block is then set to 1, and the program resumes execution.

SET ASSOCIATIVE MAPPED CACHE:



Combines the simplicity of direct mapping with the flexibility of associative mapping. The associative portion is limited to just a few slots that make up a set.

If n blocks make up a set, it is a n-way set associative cache.

When an address is mapped to a set, the direct mapping scheme is used, and then associative mapping is used within a set.

The tag memory increases only slightly from the direct mapping and only two tags need to be searched for each memory reference.

CACHE PERFORMANCE:

- READ
  - Data in Cache → forward to CPU.
  - ! Data in Cache
    - Load Through → forward the word as cache line is filled.
    - Fill cache line and then forward the word.
- WRITE
  - Data in Cache
    - Write Through → Write data to both cache and main memory.
    - Write Back → Write data to cache only. Defer main memory write until block is flushed.
  - ! Data in Cache
    - Write Allocate → Bring line to line cache, then update it
    - Write No Allocate → Update main memory only.

- SPLIT CACHE: Instructions and data are stored in separate sections of memory. Since instructions slots can never be dirty, an instruction cache is simpler than a data cache. Most of the memory traffic moves away from main memory rather than toward it. Instructions in an executing program are only read from the main memory, and are never written to the memory except by the system loader. Operations on data typically involve two operands and storing a single result.

MULTILEVEL CACHES:

The fastest level (L1) is on the same chip as the processor, and the remaining cache is placed off.