Qualifying Examination: Computer Architecture (September 2014)

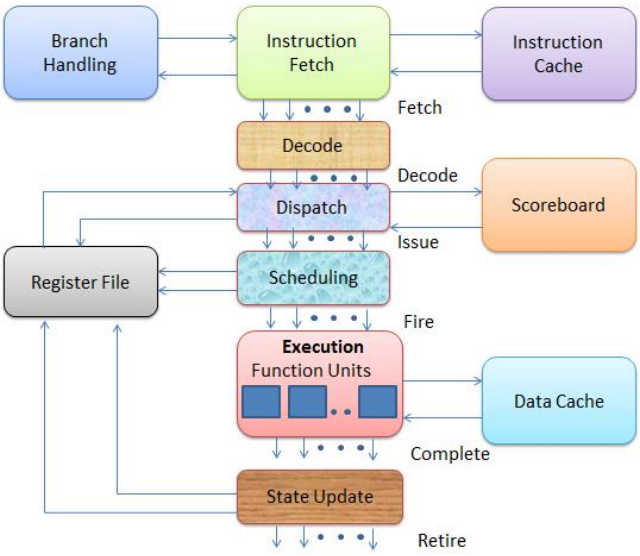
Suvendu Kumar Mohapatra
St.Id: D0121007

1. Explain the difference between superscalar and VLIW (Very Long Instruction Word) processors. Give application scenarios (Practical industry applications) for the two kinds of processors.

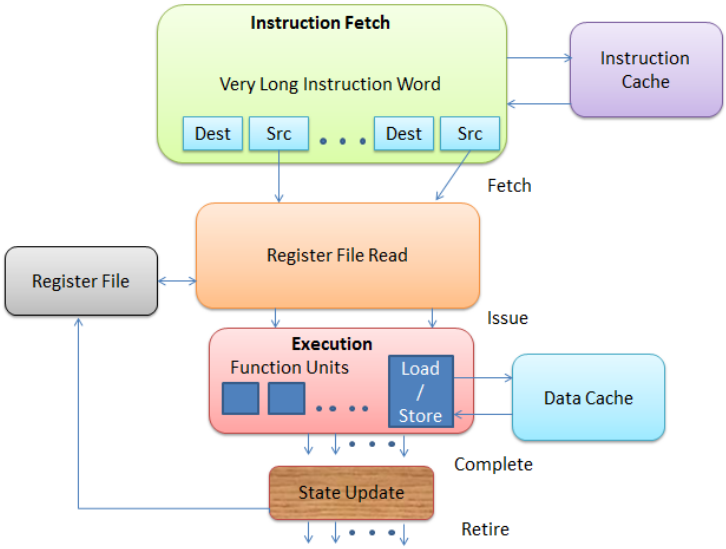
   Ans:

   The processing instructions in parallel require checking dependencies between instructions to determine which instructions can be grouped together for parallel execution and assigning instructions to the functional units on the hardware along with determining when instructions are initiated placed together into a single word. Here we are interested to show the difference between the Superscalar and VLIW processors.

   The architectural differences:



Superscalar Architecture



VLIW Architecture

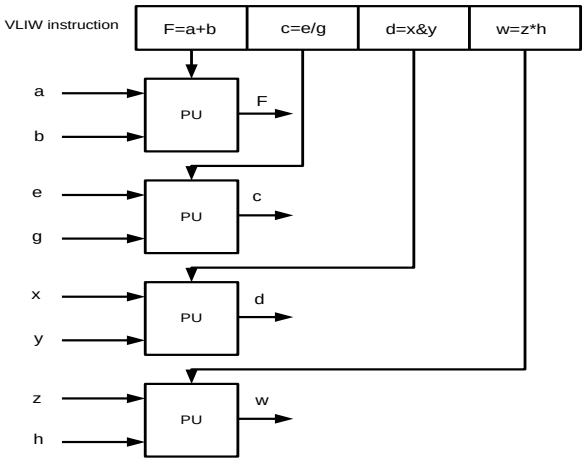
CISC instructions are varying in size and often specify a sequence of operations. CISCs tend to have few registers, and the registers may be special-purpose, which restricts the ways in which they can be used. Memory references are typically combined with other operations (such as add memory to register) and instruction sets are designed to take advantage of microcode.

RISC instructions are fixed in size and easy to decode which can deal with simple operations. RISC architectures have a relatively large number of general-purpose registers. Instructions can reference main memory only through simple load-register-from-memory and store-register-to-memory operations. RISC instruction sets do not need microcode and are designed to simplify pipelining. Available performance improvement from superscalar techniques is limited by the degree of intrinsic parallelism in the instruction stream. The complexity and time cost of the dispatcher and associated dependency checking logic along with the branch instruction processing are two major dis advantage of superscalar processor.

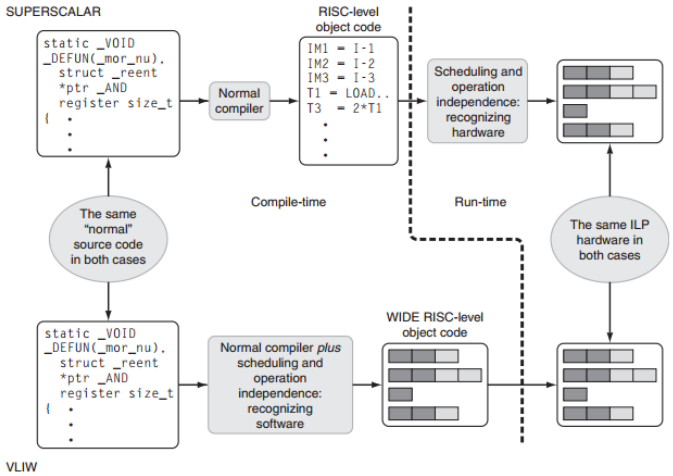
VLIW (very long instruction word) instructions are having longer as compared to RISC to allow them to specify multiple, independent simple operations. A VLIW instruction can be thought of as several RISC instructions joined together. A typical VLIW machine has

instruction words hundreds of bits in length. Multiple functional units are used concurrently in a VLIW processor. All functional units share the use of a common large register file. Instruction-level parallelism can be achieved by executing instructions without dependencies (known at compile-time) in parallel. For example of a single VLIW instruction:

F=a+b; c=e/g; d=x&y; w=z*h.



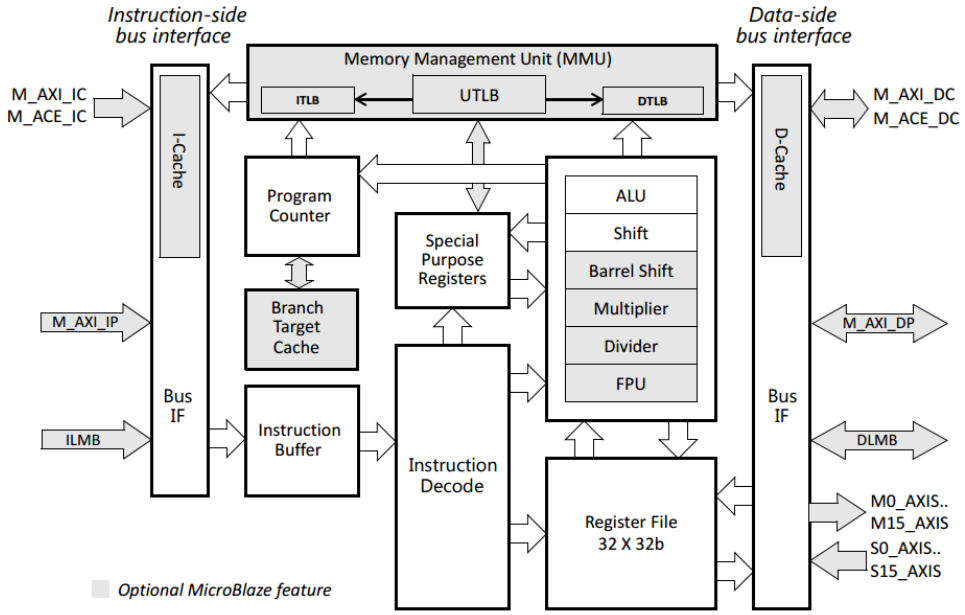
The basic advantages of VLIW are the dependencies determine by compiler and use to schedule according to function unit latencies. The function units are assigned by compiler and correspond to the position within the instruction packet. Compiler produces fully-scheduled, hazard-free code, i.e. hardware doesn't have to find again the dependencies or schedule. But it has some disadvantages. Mainly, compatibility across implementations is a major problem. VLIW code won't run properly with different number of function units or different latencies. Unscheduled events (e.g., cache miss) stall entire processor. Code density is another problem where low slot utilization and reduce by compression.



Architecture characteristics	RISC	VLIW
Instruction size	One size(32bit)	One size
Instruction format	Exploiting ilp using multiple issue and static scheduling	Regular, consistent placement of fields
Instruction Semantics	Almost always one simple operation	Many simple, independent operations
Registers	Many, general-purpose	Many, general-purpose
Memory references	Not bundled with operations, i.e., load/store architecture	Not bundled with operations, I.e., load/store architecture
Dependence information contained in the program	Implicit in register names	A description of some operations that are independent of each other

Practical industry application for superscalar processor:

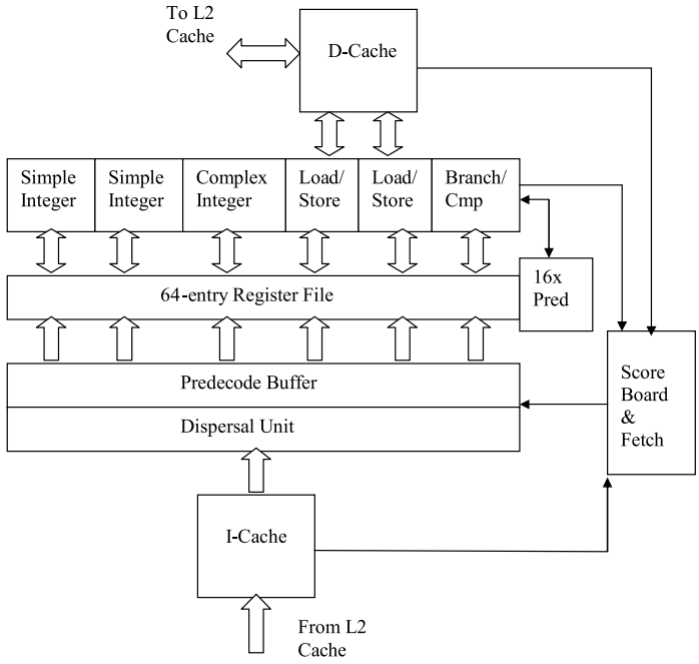
The "MicroBlaze" embedded processor soft core is a reduced instruction set computer (RISC) and optimized for implementation in Xilinx Field Programmable Gate Arrays (FPGAs). A functional block diagram of the MicroBlaze core is shown below.



MicroBlaze Core Block Diagram

The MicroBlaze soft core processor is highly configurable, allowing you to select a specific set of features required by your design. The fixed feature set of the processor includes a 32-bit general purpose registers, 32-bit instruction word with three operands and two addressing modes, 32-bit address bus and Single issue pipeline. Xilinx recommends that all new designs use the latest preferred version of the MicroBlaze processor. More details are available in Ref. 3.

VLIW processors have seen a significant commercial success since past decay. Some of the notable VLIW processors of recent years are the IA-64 or Itanium from Intel, the Crusoe processor from Transmeta, the Trimedia media-processor from Philips and the TMS320C62x series of DSPs from Texas Instruments. Now we are going to describe “Defoe” which is an example of VLIW processor. The detail of “Defoe” is available in Ref.4.



Defoe Architecture

Defoe is a 64-bit compressed VLIW architecture as shown in above figure. In an uncompressed VLIW system, MultiOps have a fixed length. When suitable operations are not available to fill the issue slots within a MultiOp, NOPs are inserted into those slots. A compressed VLIW architecture uses variable length MultiOps to get rid of those NOPs and achieve high code density. In the Defoe, individual operations are encoded as 32-bit words. A special stop bit in the 32-bit word indicates the end of a MultiOp. Common arithmetic operations have an immediate mode, where a sign or zero extended 8-bit constant may be used as an operand. For larger constants of 16, 32 or 64 bits, a special NOP code may be written into opcode field of the next operation and the low order bits may be used to store the constant. In that case, the predecoder concatenates the bits from 2 or more different words to assemble a constant. Figure 2 depicts the instruction format.

Stop bit (1 bit)	Predicate (4 bits)	Opcode (9 bits)	Rdest (6)	Rsrc1 (6)	Rscr2 (6)

Instruction Encoding

Example: This example demonstrates the execution model of the Defoe by computing the following set of expressions:

a = x + y - z

b = x + y - 2 * z

c = x + y - 3 * z

Register assignments: r1 = x, r2 = y, r3 = z, r32 = a, r33 = b, r34 = c
Line # Code Comments
1. add r4 = r1, r2 // r4 = x + y

2. shl r5 = r3, 1 // r5 = z $<<$ 1, i.e. z * 2

3. mul r6 = r3, 3 ; // r6 = z * 3. Stop bit.

4. sub r32 = r4, r3 // r5 = a = gets x + y - z

5. sub r33 = r4, r5 ; // r33 = b = x + y - 2 * z.

// Stop bit.

6. sub r34 = r4, r6 ; // r34 = c = x + y - 3 * z.

// Stop bit.

The first three lines are followed by a stop bit to indicate that those three operations constitute a MultiOp and that they should be executed in parallel. Unlike a super scalar processor where independent operations are detected by the processor, the programmer/compiler has indicated to the processor by means of the stop bit that these 3 operations are independent. The multiply operation will typically have a higher latency than the other instructions. In that case we have two different ways of scheduling this code. Since Defoe already uses score boarding to deal with variable load latencies, it is only natural for the scoreboard to stall issue till the multiply operation is done. In a traditional VLIW processor, the compiler will insert additional NOPs after the first MultiOp. Lines 4-6 show how structural hazards are handled in a VLIW system. The compiler is aware that Defoe has only two simple integer ALUs. Even though instruction 6 is independent of instructions 4 and 5, because of the unavailability of a suitable function unit, instruction 6 is issued as a separate MultiOp, one cycle after its two predecessors.

2. Describe the constraints for compiler instruction scheduling to parallelize the execution of instructions. Give examples to show the compiler instruction scheduling improve the performances of the following processors,
   2.1 A Single-pipelined RISC processor
   2.2 A VLIW processor
   Note that your program example should also encounters the constraints on instruction scheduling and show how the compiler deals with the constraints.

Ans: Now-a-days most of the processors are using pipelining to overlap the execution of instructions and improve performance is called instruction-level parallelism (ILP). If more than one instruction are executed in parallel, they can execute simultaneously in a pipeline of unknown depth without causing any stalls (assume that pipeline has sufficient resources and no structural hazards exist). If two instructions are dependent, they are not parallel and must be executed in order, although they may often be partially overlapped. Then there is a possibility of collision. There are **three major constraints** for compiler instruction scheduling to parallelize the execution of instructions. **Data dependence** is the first one in which the operations must generate the same results as the corresponding ones in the original program. The second one is **Control dependence**, in which all the operations executed in the original program must be executed in the optimized program. **Name dependence** is the third one. Here I am going to explain one by one with example.

For a Single-pipelined RISC processor:

An instruction j is data dependent on instruction i, if either instruction i produces a result that may be used by instruction j or instruction j is data dependent on instruction k, and instruction k is data dependent on instruction i. However, the instruction dependences may be within two or many instructions. If two instructions are data dependent, they cannot execute simultaneously or be completely overlapped. The dependence implies that there would be a chain of one or more data hazards between the two instructions. Executing the instructions simultaneously will cause a processor with pipeline interlocks (and a pipeline depth longer than the distance between the instructions in cycles) to detect a hazard and stall, thereby reducing or eliminating the overlap. The data dependence can be resulted three things, i.e. the possibility of a hazard, the order in which results must be calculated, and an upper bound on how much parallelism

can possibly be exploited. A dependence can be overcome in two different ways: maintaining the dependence but avoiding a hazard, and eliminating a dependence by transforming the code. Scheduling the code is the primary method used to avoid a hazard without altering a dependence, and such scheduling can be done both by the compiler and by the hardware.
For Example:

```
Loop: L.D      F0,0(R1) ;F0=array element
      ADD.D    F4,F0,F2 ;add scalar in F2
      S.D      F4,0(R1) ;store result
      DADDUI R1,R1,#-8 ;decrement pointer 8 bytes
      BNE      R1,R2,LOOP ;branch R1!=R2
```

The dependencies can be show as bellow:

```
Loop: L.D F0,0(R1) ;F0=array element

      ADD.D F4,F0,F2 ;add scalar in F2

      S.D F4,0(R1) ;store result

      DADDIU R1,R1,-8 ;decrement pointer ;8 bytes (per DW)

      BNE R1,R2,Loop ;branch R1!=R2
```

Above dependent sequences, as shown by the arrows, have each instruction depending on the previous one. The dependence implies that there would be a chain of one or more data hazards between the two instructions. Data hazards can be classified as one of three types, depending on the order of read and write accesses in the instructions. The hazards are named by the ordering in the program that must be preserved by the pipeline. Consider two instructions i and j, with i preceding j in program order. The possible data hazards are **RAW** (read after write): j tries to read a source before i writes it, so j incorrectly gets the old value. This hazard is the most common type and corresponds to a true data dependence. Program order must be preserved to ensure that j receives the value from i. **WAW** (write after write): j tries to write an operand before it is written by i. The writes end up being performed in the wrong order, leaving the value written by i rather than the value written by j in the destination. This hazard corresponds to an output dependence. WAW

hazards are present only in pipelines that write in more than one pipe stage or allow an instruction to proceed even when a previous instruction is stalled. **WAR** (write after read): j tries to write a destination before it is read by i, so i incorrectly gets the new value. This hazard arises from an antidependence. A WAR hazard occurs either when there are some instructions that write results early in the instruction pipeline and other instructions that read a source late in the pipeline, or when instructions are reordered.

The second type of dependence is **name dependence**. A name dependence occurs when two instructions use the same register or memory location, called a name, but there is no flow of data between the instructions associated with that name. There are two types of name dependences between an instruction i that precedes instruction j in program order.

An "anti-dependence" between instruction i and instruction j occurs when instruction j writes a register or memory location that instruction i reads. The
original ordering must be preserved to ensure that I reads the correct value. An antidependence between S.D and DADDIU on register R1.
Example:

S.D F4,0(R1) ;store result

DADDIU R1,R1,-8 ;decrement pointer ;8 bytes (per DW)

An "output dependence" occurs when instruction i and instruction j write the same register or memory location. The ordering between the instructions must be preserved to ensure that the value finally written corresponds to instruction j.

A **control dependence** determines the ordering of an instruction, i, with respect to a branch instruction so that the instruction i is executed in correct program order and only when it should be. Every instruction, except for those in the first basic block of the program, is control dependent on some set of branches, and, in general, these control dependences must be preserved to preserve program order. One of the simplest examples of a control dependence is the dependence of the statements in the "then" part of an if statement on the branch.

For example:

```
if p1 {
        S1;
     };
if p2 {
        S2;
     }
```

S1 is control dependent on p1, and S2 is control dependent on p2 but not on p1.

In general, there are two constraints imposed by control dependences:

1. An instruction that is control dependent on a branch cannot be moved before the branch so that its execution is no longer controlled by the branch. For example, we cannot take an instruction from the then portion of an if statement and move it before the if statement.
2. An instruction that is not control dependent on a branch cannot be moved after the branch so that its execution is controlled by the branch. For example, we cannot take a statement before the if statement and move it into the then portion.

Example: Consider a loop like this one:

```
for (i=1; i<=100; i=i+1) {
  A[i] = A[i] + B[i];       /* S1 */
  B[i+1] = C[i] + D[i];     /* S2 */
}
```

Statement S1 uses the value assigned in the previous iteration by statement S2, so there is a loop-carried dependence between S2 and S1 as shown in Bold. Despite this loop-carried dependence, this loop can be made parallel.

```
A[1] = A[1] + B[1];
for (i=1; i<=99; i=i+1) {
  B[i+1] = C[i] + D[i];
  A[i+1] = A[i+1] + B[i+1];
}
B[101] = C[100] + D[100];
```

Now the dependencies no longer exist.

We also shoe how it affect the performances by using both scheduled and unscheduled, including any stalls or idle clock cycles. Schedule delays from floating-point operations not for delayed branches with loop Unrolling.

Without any scheduling, the loop will execute as follows, taking 9 cycles:

		Clock cycle issued
Loop:	L.D F0,0(R1)	1
	Stall	2
	ADD.D F4,F0,F2	3
	Stall	4
	Stall	5
	S.D F4,0(R1)	6
	DADDUI R1,R1,#-8	7
	Stall	8
	BNE R1,R2,Loop	9

We can schedule the loop to obtain only two stalls and reduce the time to 7 cycles:

		Clock cycle issued
Loop:	L.D F0,0(R1)	1
	DADDUI R1,R1,#-8	2
	ADD.D F4,F0,F2	3
	Stall	4
	Stall	5
	S.D F4,8(R1)	6
	BNE R1,R2,Loop	7

VLIW that could issue two memory references, two FP operations, and one integer operation or branch in every clock cycle.

```
for (i=1; i<=100; i=i+1) {
  A[i+1] = A[i] + C[i]; /* S1 */
  B[i+1] = B[i] + A[i+1]; /* S2 */
}
```

There are two different dependences:

1. S1 uses a value computed by S1 in an earlier iteration, since iteration i

Computes A[i+1] , which is read in iteration i+1. The same is true of S2 for B[i] and B[i+1].
2. S2 uses the value, A[i+1] , computed by S1 in the same iteration.

Example 2 for VLIW:

```
for (i=1; i<=100; i=i+1) {
Y[i] = X[i] / c; /* S1 */
X[i] = X[i] + c; /* S2 */
Z[i] = Y[i] + c; /* S3 */
Y[i] = c - Y[i]; /* S4 */
}
```

Dependencies: There are true dependences from S1 to S3 and from S1 to S4 because of Y[i]. These are not loop carried, so they do not prevent the loop from being considered parallel. These dependences will force S3 and S4 to wait for S1 to complete. There is an antidependence from S1 to S2, based on X[i]. There is an antidependence from S3 to S4 for Y[i]. There is an output dependence from S1 to S4, based on Y[i].

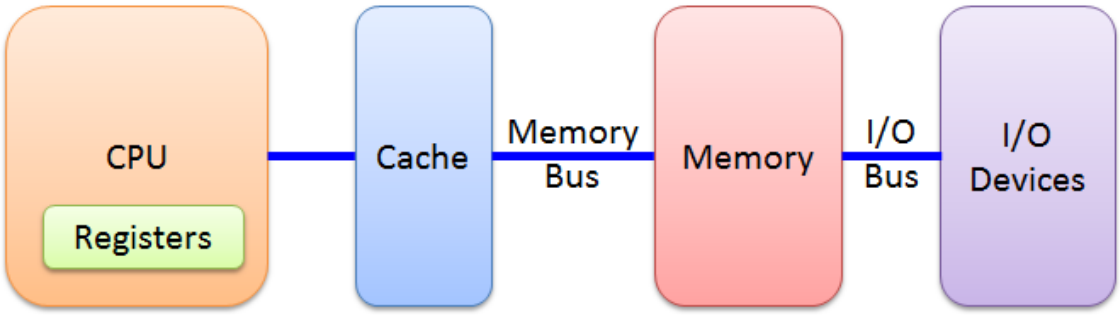
The following version of the loop eliminates these false (or pseudo) dependences.

```
for (i=1; i<=100; i=i+1 {
/* Y renamed to T to remove output dependence */
T[i] = X[i] / c;
/* X renamed to X1 to remove antidependence */
X1[i] = X[i] + c;
/* Y renamed to T to remove antidependence */
Z[i] = T[i] + c;
Y[i] = c - T[i];
}
```

After the loop, the variable X has been renamed X1. In code that follows the loop, the compiler can simply replace the name X by X1. In this case, renaming does not require an actual copy operation but can be done by substituting names or by register allocation. In other cases, however, renaming will require copying.

3. Draw the typical architecture of a cache memory and explain why the cache memory may help to improve the program execution performance.

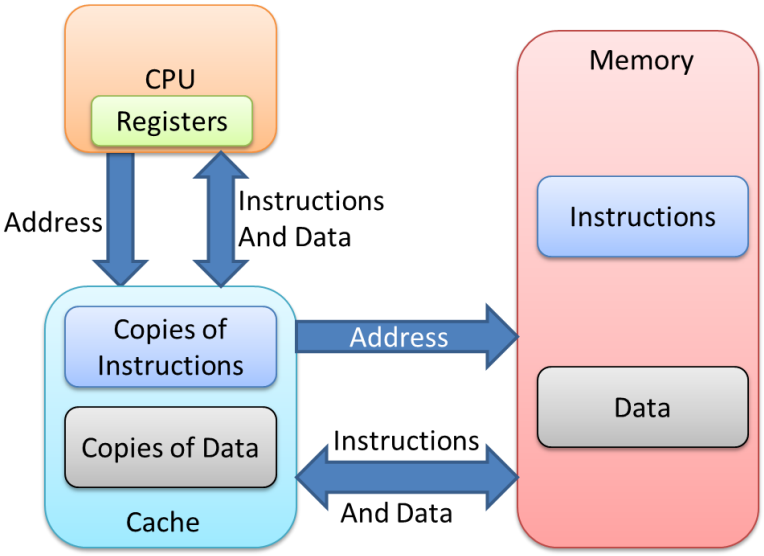
Ans: The cache is a small amount of high-speed memory that retains copies of recently used memory values. It is usually implemented on the same chip as the processor. Effectiveness of the cache relies on the programs normally display the property of locality, Locality occurs in time (temporal locality) and in space (spatial locality). An access to an item which is in the cache is called a hit, and an access to an item which is not in the cache is a miss. A processor can have one of the following two organizations, i.e. a unified cache and Separate instruction and data caches. The cache memory hierarchy is shown below.



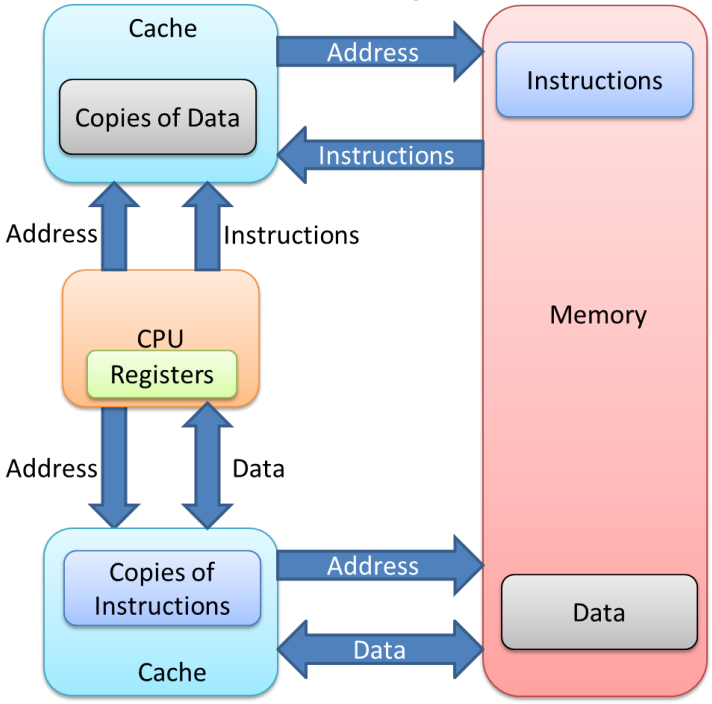
	Register Reference	Cache Reference	Memory Reference	Disk Memory Reference
	Register	Cache	Memory	Disk Memory
Size	500 Bytes	64KB	1GB	1TB
Speed	250ps	1ns	100ns	10ms
Current Technology		SRAM	DRAM	Magnetic Disk

Table1: Cache memory configuration

A unified cache consists of a single cache for both instructions and data as shown below.



But in and separate instruction and data caches, the instruction and data are in two different caches as shown in figure below.



Cache memory can be categorized in different levels that describe its closeness and accessibility to the microprocessor. Level 1 represents the **L1 cache** close to the processor, small in size bit extremely fast. In level 2, **L2 cache** is located in between the process and the system bus which is fairly fast and medium in size. In level 3, **L3 cache** is present which is relatively larger than L1 and L2. In cache, data can be transparently

stored so that future requests for that data can be access faster. If requested data is contained in the cache (cache hit), the request can be served by simply reading the cache, which is comparatively faster. Otherwise if cache miss occurs, the data has to be recomputed or fetched from its original storage location, which is comparatively slower. The greater the number of requests that can be served from the cache, the overall system performance can be higher. Memory cache can work under three different categories, i.e. direct mapping, set associative and fully associative. In direct mapping each block is mapped to exactly one cache location with the formula {(Block Number) modulo (Number of blocks in the cache)}. In set associative, each block is mapped to a subset of cache locations with the formula {(Block Number) modulo (Number of sets in the cache)}. In fully associative mapping, each block is mapped to any cache location.

To know how the cache memory helps to improve the program execution performance we will take an example.

Example:

Given that:
Miss rate of an instruction cache = 2%
Miss rate of the data cache = 4%.
Processor CPI = 2 (without any memory stalls)
Miss penalty = 100 cycles for all misses
Frequency of all loads and stores = 36%

Need to: Estimate how much faster a processor would run.

Assume that: Processor would run with a perfect cache that never missed.
Instruction count = I

Instruction miss cycles = I × 2% × 100 = 2.00 × I

Data miss cycles = I × 36% × 4% × 100 = 1.44 × I

The total number of memory-stall cycles is 2.00 I + 1.44 I = 3.44 I.
This is more than three cycles of memory stall per instruction.

Accordingly, the total CPI including memory stalls is 2 + 3.44 = 5.44. Since there is no change in instruction count or clock rate, the ratio of the CPU execution times is

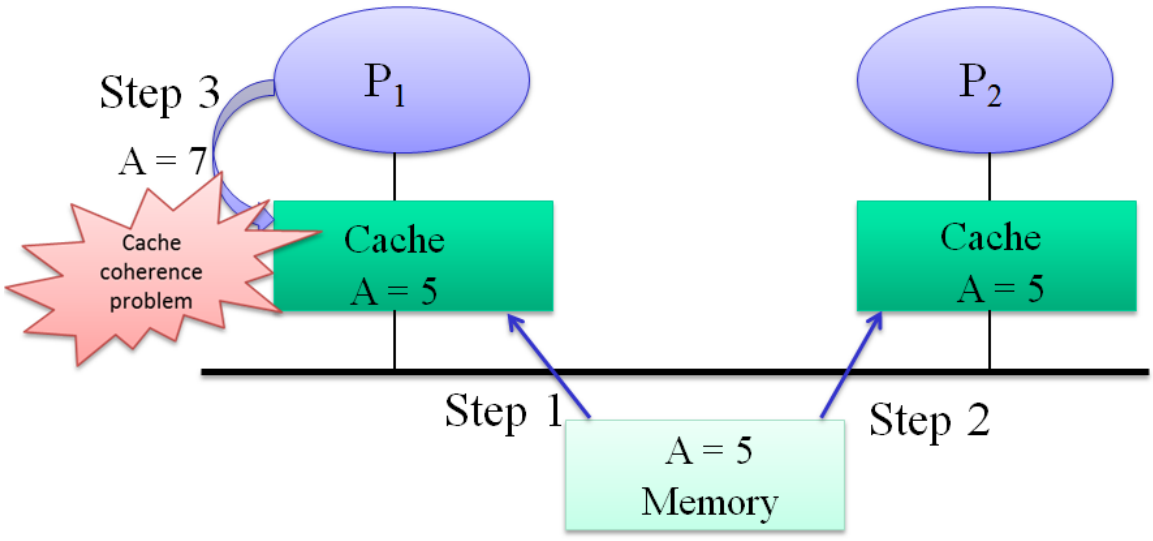
$$\frac{CPU\ time\ with\ stalls}{CPU\ time\ with\ perfect\ cache} = \frac{I * CPIstall * Clock\ Cycle}{I * CPIperfect * Clock\ Cycle}$$

$$= \frac{5.44}{2} = 2.72$$

$$= The\ performance\ of\ processor\ \ is\ better\ by\ 2.72$$

4. Explain a multi-core processor needs a cache coherence protocol. Give an example of an cache coherence protocol and show how it works.

Ans: In a multi-core processor environment, assorted processor cores are present. Those various processors are having one $L1$ cache each along with one common $L2$ cache. If multiple copies of data are distributed throughout the caches then it leads to a coherence problem among the caches. The copies in the caches are coherent if they all equal the same value. However, if one of the processors writes over the value of one of the copies, then the copy becomes inconsistent because it no longer equals the value of the other copies. If data are allowed to become inconsistent, incorrect results will be propagated through the system, leading to incorrect final results. Cache coherence algorithms are needed



to maintain a level of consistency throughout the parallel system. There is a need of cache coherence protocol to enhance the overall performance

widespread thread-level parallelism. Basically, the coherence defines the current/recent value returned by a read, whereas consistency refers to when a written value will be returned by a read. For example, refer to the table 2 below:

Time Stamp	Event	Cache contents for Processor 1	Cache contents for Processor 2	Memory contains for location X
0				5
1	Processor 1 reads X	5		5
2	Processor 2 reads X	5	5	5
3	Processor 1 stores 7 into X	7	5	7

Table2: Cache coherence problem for single memory locations by two processors (1 and 2).

By taking this table as example, we can say that the system is coherent if and only if meet the following conditions:

Condition 1: After time stamp 3, if Processor 1 want to read X, it should get the value 7 instead of 5.

Condition 2: After time stamp 3, the cache value of Processor 2 needs to be update to 7 instead of 5 by using some protocols.
Condition 3: Write serialization.

Therefore we need some cache coherence protocol to track the state of any sharing of a data block. The protocols are divided into two groups, i.e. Software based and hardware based. In Software-based protocol or the compiler based or with run-time system support require hardware assist. This leads to a big problem because perfect information is needed in the presence of memory aliasing and explicit parallelism. Hence, hardware based solutions is more appreciated. In hardware solutions, snooping and directory based protocol are more common.

In **snooping protocol**, it is assumed that a processor has exclusive access to a data item before it writes that item which ensures that no other

readable or writeable copies of an item exist when the write occurs. This is also known as the write invalidate protocol as it invalidate other caches on a write. The snoopy protocols can be again divided into two categories: write-invalidate (when a line is written, all other copies are invalidated) and write update (when all copies are updated).

Processor Activity	Bus Activity	Cache contents for Processor 1	Cache contents for Processor 2	Memory contains for location X
				5
Processor 1 reads X	Cache miss for X	5		5
Processor 2 reads X	Cache miss for X	5	5	5
Processor 1 stores 7 into X	Invalidation for X	7	No Activity	5
Processor 2 reads X	Cache miss for X	7	7	7

Table3: Invalidation protocol working on a snooping bus for a single cache block with write-back caches.

In snoopy-based protocols few processors are connected through a bus. But in directory-based protocols is used distributed shared memory paradigm. The disadvantages in classical snoopy cache coherence protocols are induced lookups consume a significant amount of energy, although most of the snoops require no further action. The size of the shared memory system, updating or invalidating caches using snoopy protocols might become unpractical. For example, it is very expensive when a multistage network is used to build a large shared memory system and use the broadcasting techniques. A directory is a data structure that maintains information on the processors that share a memory block and on its state.

Directory-based protocols can be categorized under three categories: full-map directories, limited directories, and chained directories. In a full-map setting, each directory entry contains N pointers, where N is the number of processors. Therefore, there could be N cached copies of a particular block shared by all processors. For every memory block, an N-bit vector is maintained, where N equals the number of processors in the shared

memory system. Each bit in the vector corresponds to one processor. Limited directories have a fixed number of pointers per directory entry regardless of the number of processors. Restricting the number of simultaneously cached copies of any block should solve the directory size problem that might exist in full-map directories. Chained directories emulate full-map by distributing the directory among the caches. They are designed to solve the directory size problem without restricting the number of shared block copies. Chained directories keep track of shared copies of a particular block by maintaining a chain of directory pointers.

Bibliography

[1]. Book: Advanced Computer Architecture And Parallel Processing
[2]. Book: Computer Organization and Design: The Hardware/ Software Interface, Revised Fourth Edition
[3].http://www.xilinx.com/support/documentation/sw_manuals/xilinx2014_2/ug984-vivado-microblaze-ref.pdf
[4].http://www.siliconintelligence.com/people/binu/coursework/686_vliw/vliw.pdf
[5]. http://esca.korea.ac.kr/teaching/com515_ACA/Appendices/AppG.pdf