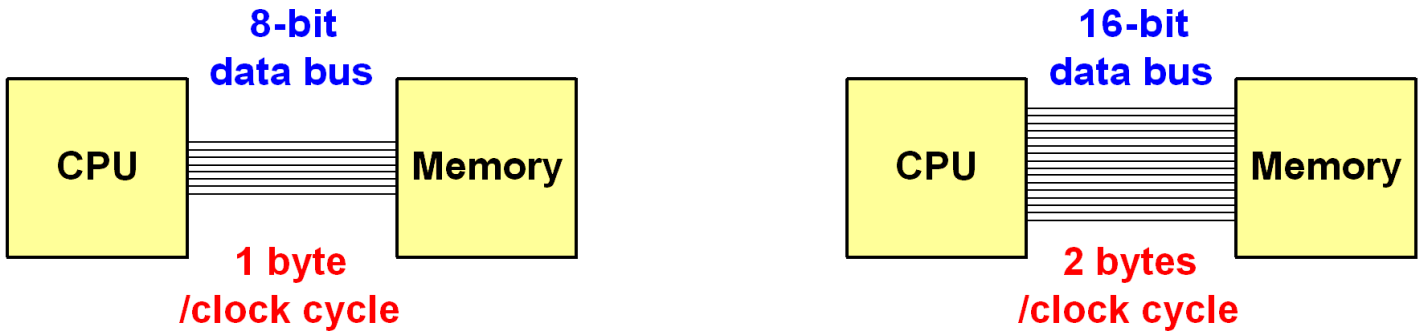
Today's Topics

- Using cache memory to improve performance
- Cache organizations – associative vs. direct-mapped caches
- Branch prediction and superscalar execution
- Using register renaming to eliminate instruction dependencies
- Reordering instructions to improve performance
- Speculative execution
- Comparing the sample microarchitectures:
  - Pentium-4
  - UltraSPARC III
  - 8051
- Introduction to the Instruction Set Architecture level
- Properties of the ISA level
- Comparison of ISA levels for the sample microarchitectures

Cache Memory

In the MIC microarchitectures we saw that the system often had to wait while data is read from memory. In modern systems with very fast clocks this is an even bigger problem because the system may have to wait for many clock cycles before data is delivered from memory.

It's fairly easy to speed up the bandwidth of memory by adding more data lines so that many bytes can be delivered from memory at the same time:



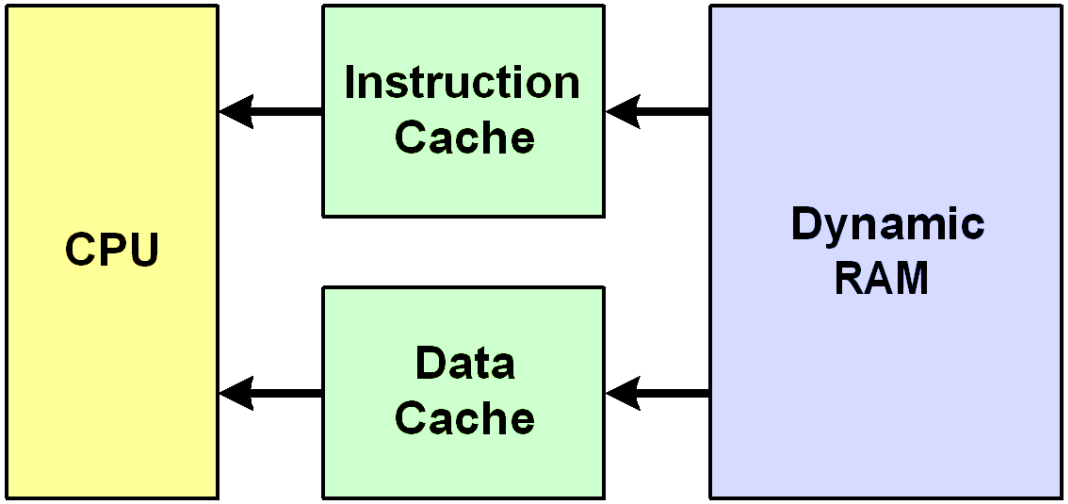
Increasing bandwidth works well for fetching instructions from memory because they are stored in sequential memory locations – the fetch operation can be requested ahead of time so that the memory has already delivered the instruction by the time it's needed.

But the CPU can't determine what data (operands) an instruction will need until it executes the instruction. So a big problem to be solved is how to reduce the memory latency (the time between when a read is requested and the data is delivered).

Caches attempt to reduce memory latency by storing a copy of recently-used data in high speed static RAM. If the program re-uses the same data (or nearby data), it can be returned from the high-speed RAM instead of having to wait for the slower Dynamic RAM.

Split Caches

A good way to improve both bandwidth and latency is to use a separate cache for fetching instructions and for reading data from memory. This technique is called a "split cache".



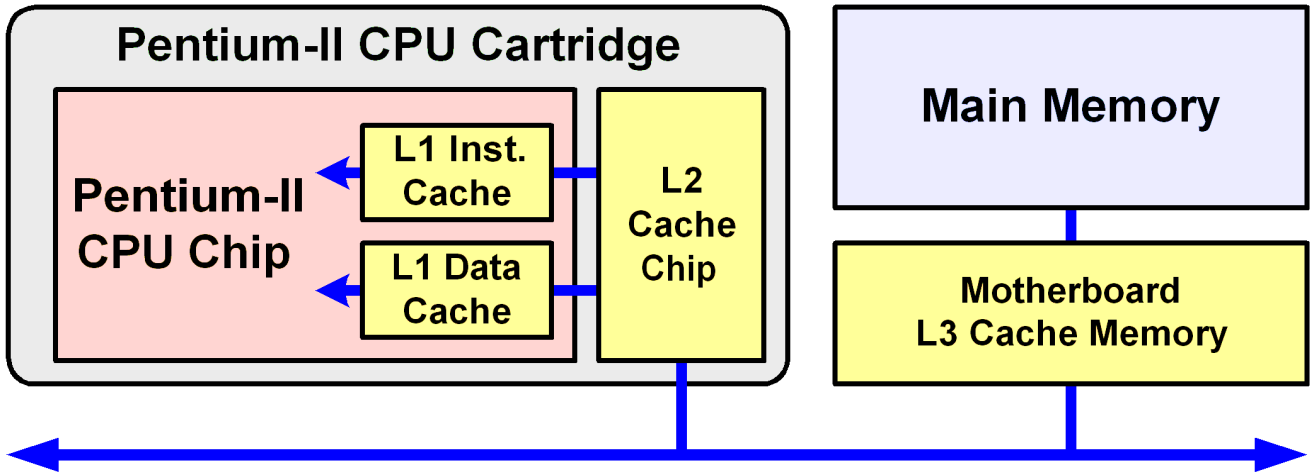
(Split caches are sometimes called "Harvard Architecture", but this term should really be used only for systems that use a split main memory, not just a split cache)

Split caches improve bandwidth because the CPU has two separate data paths to the cache subsystem. And they improve latency because instruction fetches don't have to wait for data reads to complete (and vice versa).

The MIC systems we studied earlier had two memory ports (instructions via PC / MBR and data via MAR / MDR), and so they would have used a split cache.

Cache Levels

Some systems have several cache levels. The Pentium-II had a split 16KB cache right on the CPU chip and a second 512MB cache chip contained within the same CPU cartridge. There may also be a cache system on the motherboard:



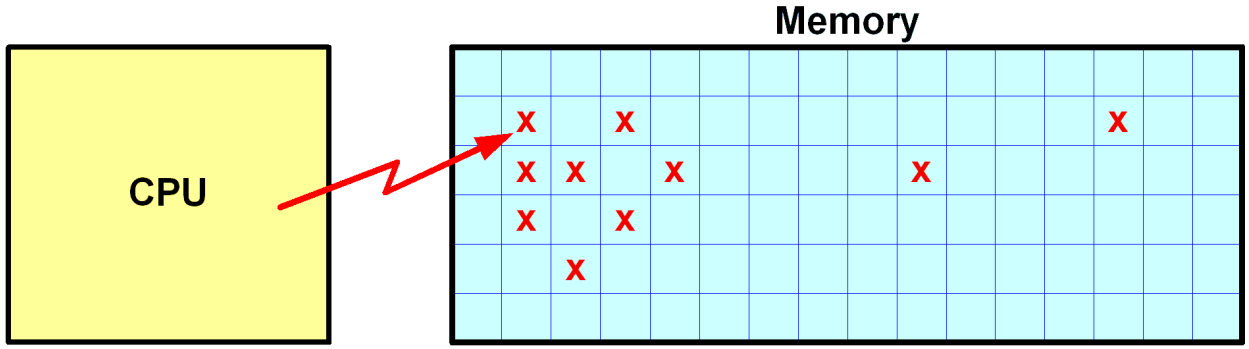
Caches get larger as they move further away from the CPU.

Cache Effectiveness

The effectiveness of cache memory depends on two locality principles that are exhibited by most programs:

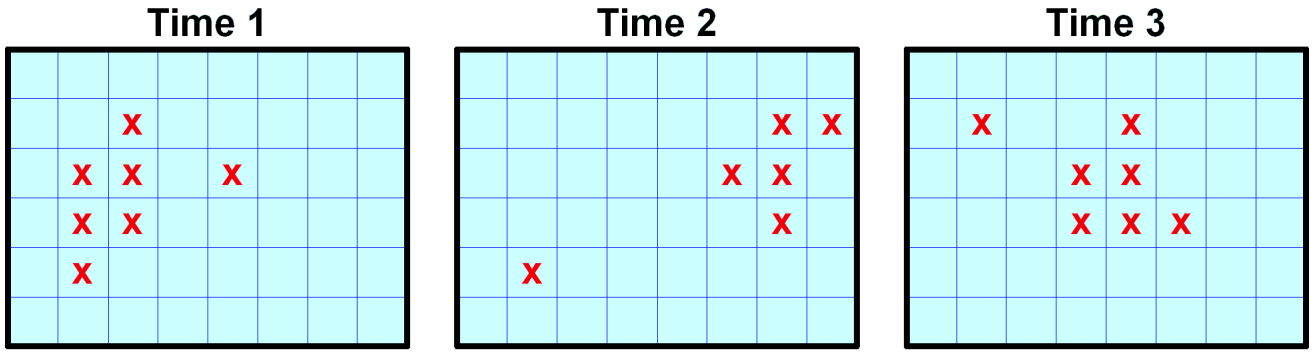
Spatial Locality

Most programs do most of their memory accesses to a relatively small set of memory addresses (the 80/20 rule):



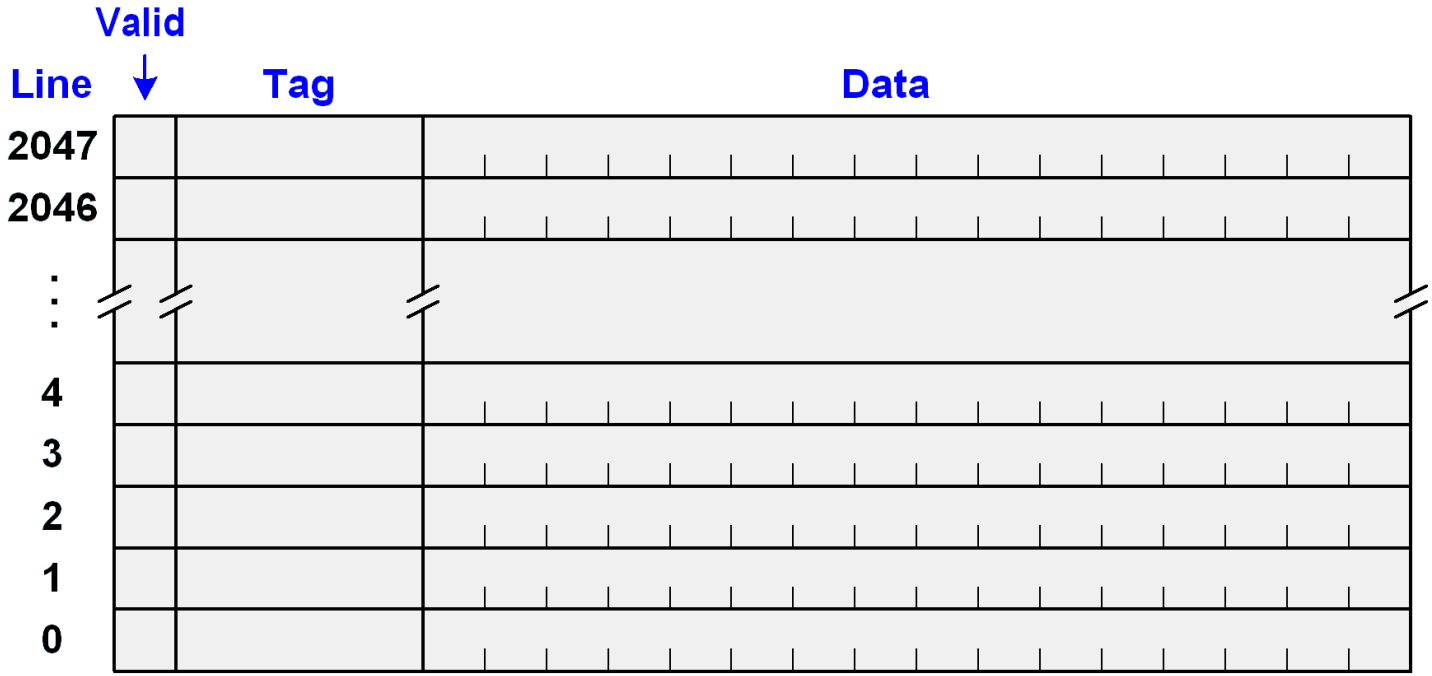
Temporal Locality

The CPU tends to access a certain set of memory locations at a given time, then moves on to access other locations:



Cache Organization

All cache systems store data in a series of cache lines with the following general organization:



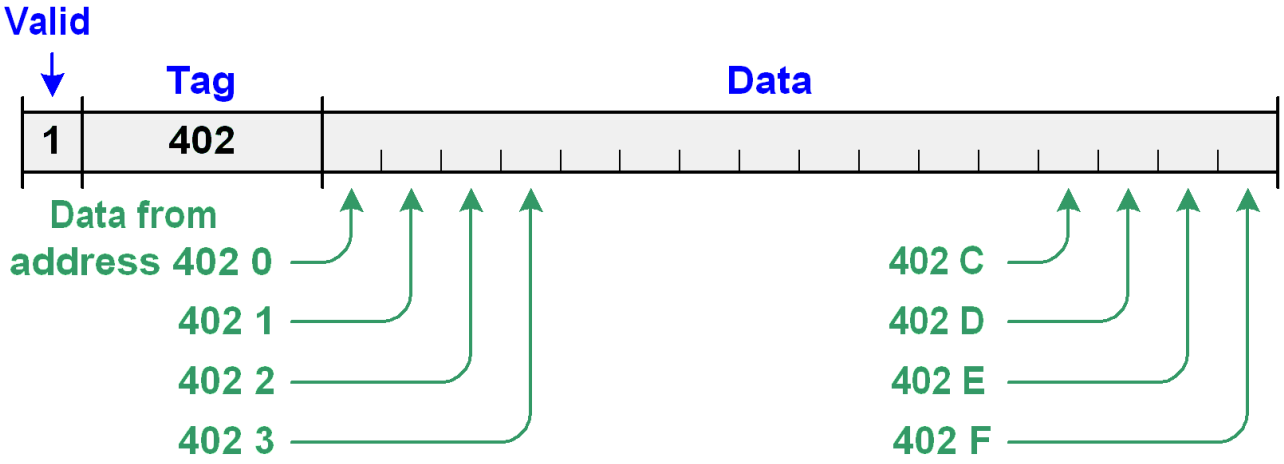
Valid – Each cache line contains a "valid" bit to indicate whether or not the line actually contains valid data. The valid bits are set to zero at startup time and then turned on as data is read into each cache line.

Tag – The tag tells what memory addresses the data was read from.

Data – the cache reads data from memory in blocks of 4 or more bytes – these blocks of data are stored in the cache line. The example above shows a cache that stores data in blocks of 16 bytes.

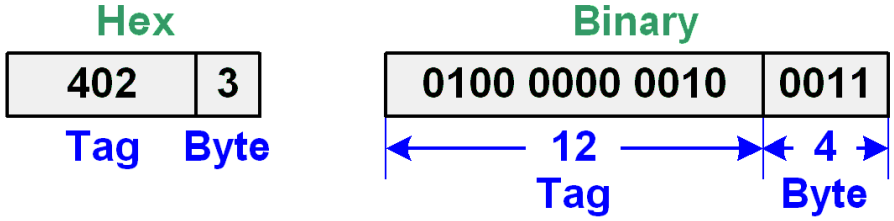
Tag vs. Byte Number

Note that, since the cache stores multiple bytes in each line, the "tag" (copy of the memory address) really specifies several addresses. Let's take an example of a cache line that has a copy of data from memory addresses 4020 through 402F:



If the CPU requests a read from memory address "4023", the cache uses the first part of the memory address ("402") to search for the a cache line containing the matching tag. If a matching tag is found, the cache uses the last part of the address ("3") to choose which byte in the cache line to return.

So the cache uses different parts of the memory address for different purposes:

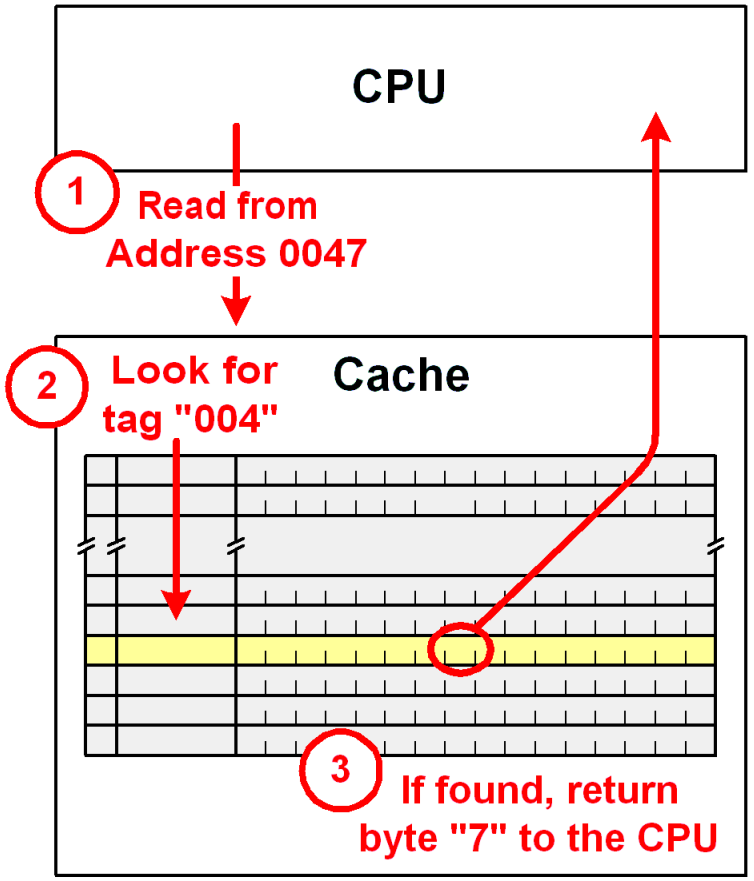


Note that the byte number is 4 bits long because the entry has 16 bytes in it (and $2^4 = 16$)

Cache Operation

The basic operation of a cache subsystem works like this:

1. **The CPU issues a read for a certain memory address.**
2. **The Cache extracts the "tag" from the address and uses it to search for a matching cache line (the matching line must have the same tag and the "valid bit" must also be turned on)**
3. **If a matching line is found, the cache uses the "byte number" part of the address to locate the requested byte within the cache line, and then returns that byte to the CPU.**



If the cache <u>can't</u> find the requested address, then it does the following:

4. **Finds an empty cache line to read the requested information into, or, if there are no empty lines, chooses an old line to replace with the new data.**
5. **Issues a read to memory for 16 bytes to fill the chosen cache line. When the data arrives from memory it is written into the cache line along with the tag, and the "valid" bit is turned on.**
6. **Uses the "byte number" part of the address to locate the requested byte within the new cache line and returns that byte to the CPU.**

Fully Associative Cache

The cache we have described is called a "Fully Associative Cache". A fully associative cache can store information from anywhere in memory in any cache line. This is the most flexible and easy to understand cache organization, but it has a number of drawbacks that make it very difficult to build:

- **The tags for every cache line must be checked on every memory access. So if there are 2048 lines in the cache, then 2048 comparisons must be made in order to tell if the cache contains the right data.**

  The only way to do this with reasonable performance would be to build 2048 comparator circuits so that all 2048 lines can be checked in parallel. But in practice, cache performance is better served by storing more data using the chip real estate that the comparators would use up.

- **It becomes very difficult to determine which cache entry to replace when new data must be read into the cache. The choice of which entry to replace is called the "Cache Replacement Algorithm". Common schemes include "Least Recently Used" and "Last In, First Out". But with 2048 entries to choose from when deciding which entry to use, it's very difficult to maintain all the information needed to make a sensible choice.**

Because of these limitations, fully associative caches are not for general-purpose memory caches. Instead, one of the following organizations is used instead:

Direct-Mapped Cache

Set Associative Cache

Direct-Mapped Cache

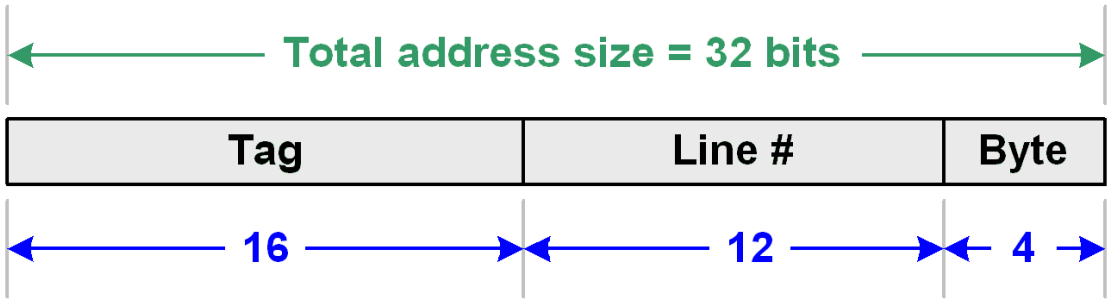
The physical storage of a direct-mapped cache is the same as has been described for a fully associative cache, but the use of the memory address is different:

A direct-mapped cache uses part of the memory address to determine which cache line the cached data will be stored in.

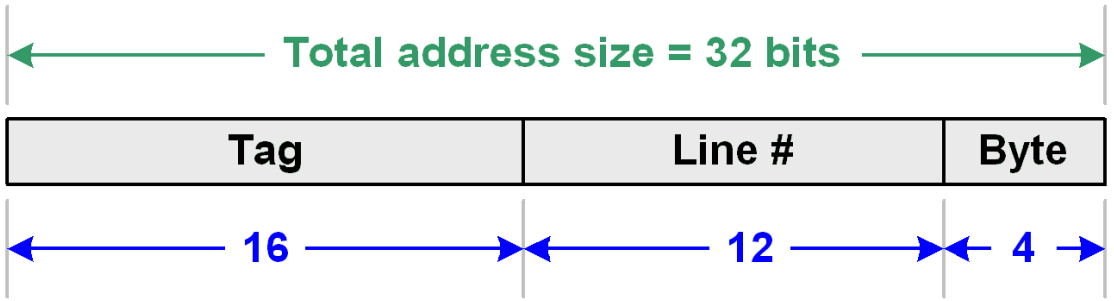
For example, let's assume that we have a direct-mapped cache with the following characteristics:

- **There are 16 data bytes stored in each cache line. This means 4 bits of the address are used for the byte number (because $2^4 = 16$)**
- **The system can access 4GB of memory. This means that a complete address is 32 bits long (because $2^{32} = 4GB$)**
- **There are 4096 lines in the cache.**

In this system, the memory address would be used by the cache subsystem as follows:



Direct-Mapped Cache



Notice that there is a new portion of the address called "Line #". The bits in this portion of the address tell the cache system which cache line number will hold the data. The cache can look directly at this particular line to see if it's tag matches. The line number is 12 bits long because the cache contains 4096 lines (and $2^{12} = 4096$).

The tag is whatever is left over in high-order part of the address after the space for "Byte" and "Line #" is used up. In our example, the "Byte" uses up 4 bits and the "Line #" uses up 12 bits. In this example, since the total address size is 32 bits, then tag is 32 – 12 – 4 = 16 bits long.

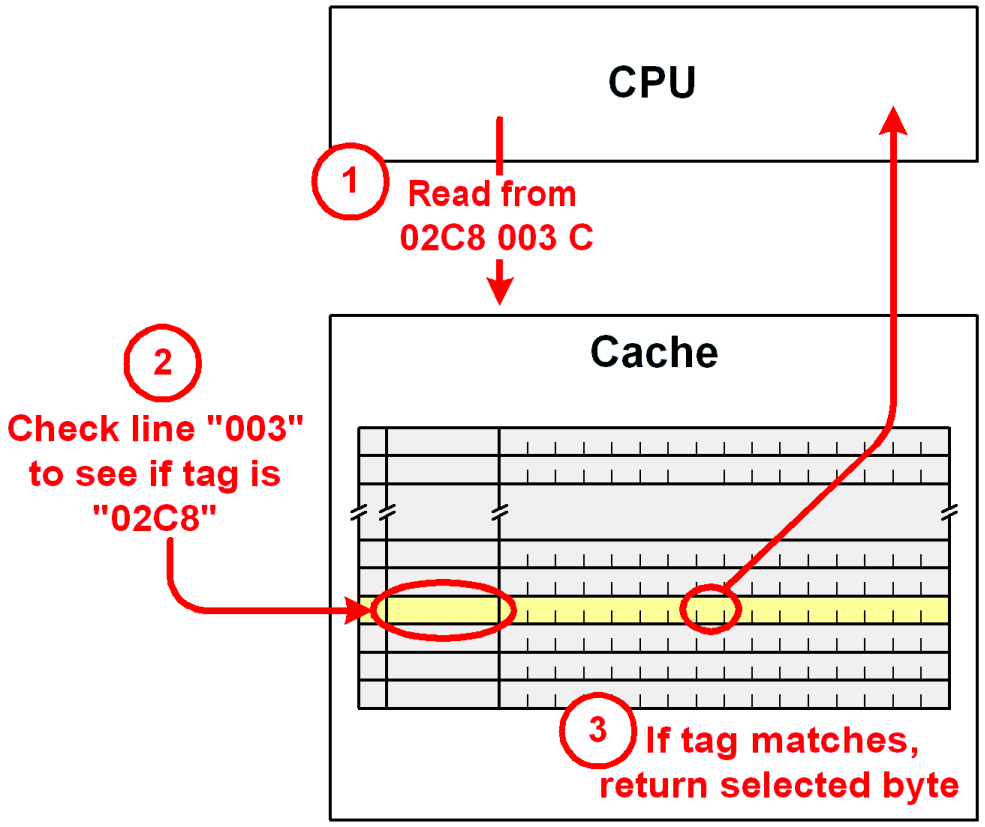
Using part of the address as the cache line number simplifies the cache considerably. Now, whenever the cache receives a memory read from the CPU, it no longer needs to search for a matching cache line – it simply goes directly to the given line number and checks to see if the tag matches.

Replacing a cache line is much simpler too – there's no longer any decision to make. If the selected cache line doesn't contain the correct data, then the new data is read from memory and it simply overwrites the old data in the selected line.

Direct-Mapped Cache Operation

A direct mapped cache subsystem works like this:

1. **The CPU issues a read for a certain memory address.**
2. **The Cache extracts the "line #" from the address and checks to see if that cache line matches the address's "tag" (the line must have the same tag and the "valid bit" must also be turned on)**
3. **If the line matches, the cache uses the "byte number" part of the address to locate the requested byte within the cache line, and then returns that byte to the CPU.**



If the selected line doesn't match, then the cache does the following:

4. **Issues a read to memory for 16 bytes to fill the chosen cache line. When the data arrives from memory it is written into the selected cache line along with the tag, and the "valid" bit is turned on.**
5. **Uses the "byte number" part of the address to locate the requested byte within the new cache line and returns that byte to the CPU.**

Direct-Mapped Cache

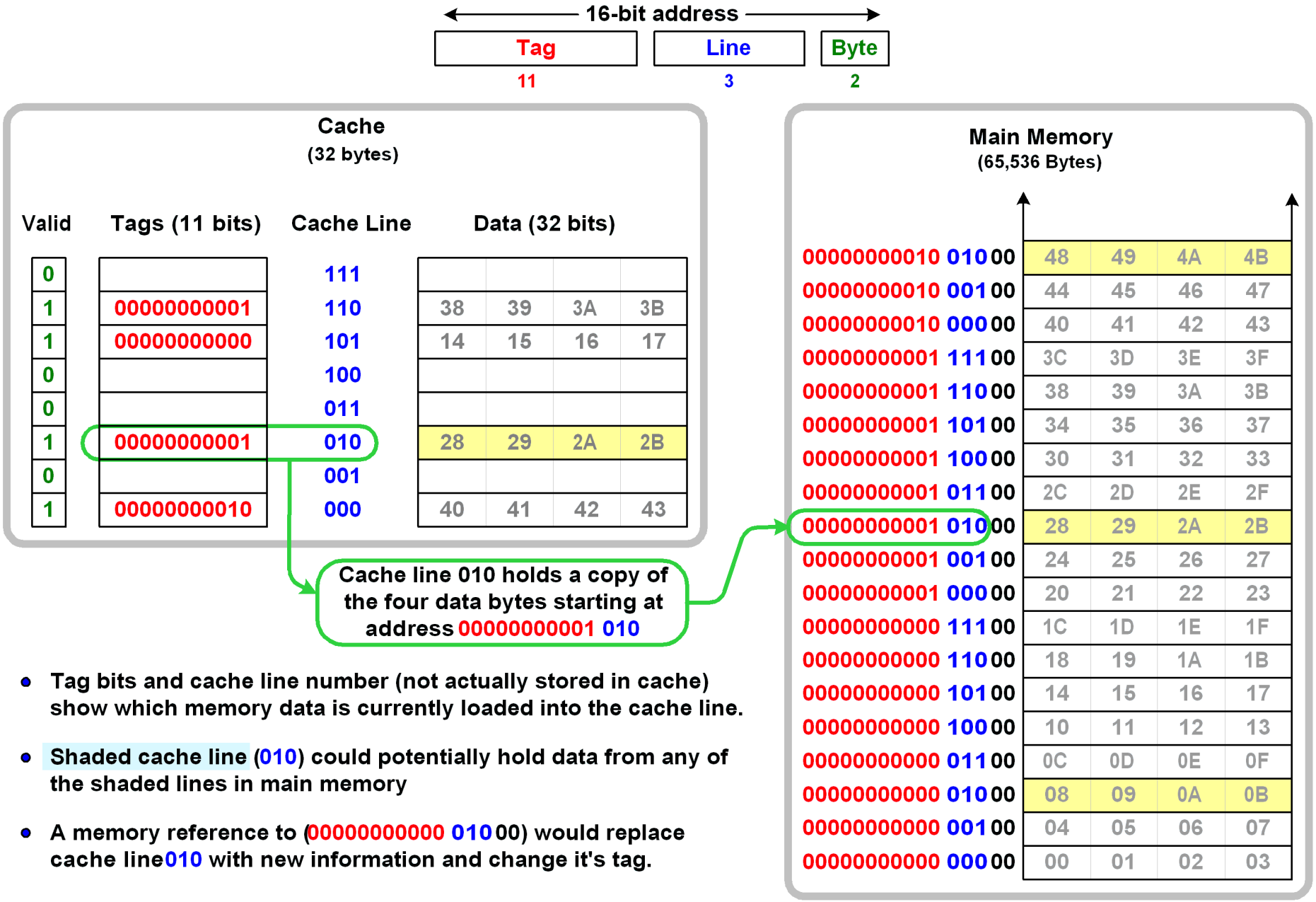
A direct-mapped cache is simple and fairly effective, but it's biggest disadvantage is that any given byte in memory can only ever be put into one cache line. This means that under certain conditions, if the program is alternately accessing two different addresses that happen to map to the same cache line, the cache can "thrash":

Read from address…	…accesses cache line	…with tag
02C8 003 2	003	02C8
3F75 003 7	003	3F75
02C8 003 3	003	02C8
3F75 003 8	003	3F75

Under circumstances like these (which, fortunately, don't occur too often), every read results in a cache miss and the data always has to be fetched from memory. This will dramatically slow the program down.

The next page shows an example using a very simple, small cache to illustrate how the mapping between memory addresses and cache lines works.

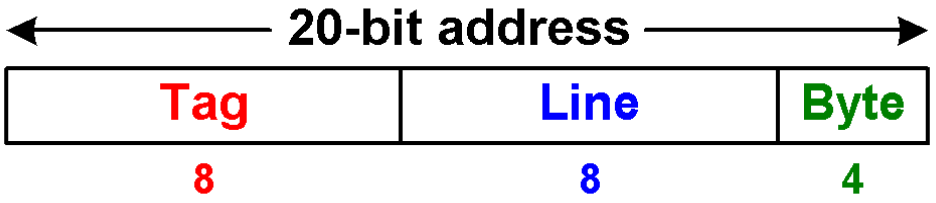
Direct-Mapped Cache using a 16-bit Address



- Tag bits and cache line number (not actually stored in cache) show which memory data is currently loaded into the cache line.
- Shaded cache line (010) could potentially hold data from any of the shaded lines in main memory
- A memory reference to (00000000000 010 00) would replace cache line010 with new information and change it's tag.

Exercise 1 – Direct-Mapped Cache

A system has a direct-mapped cache which interprets memory addresses as follows:



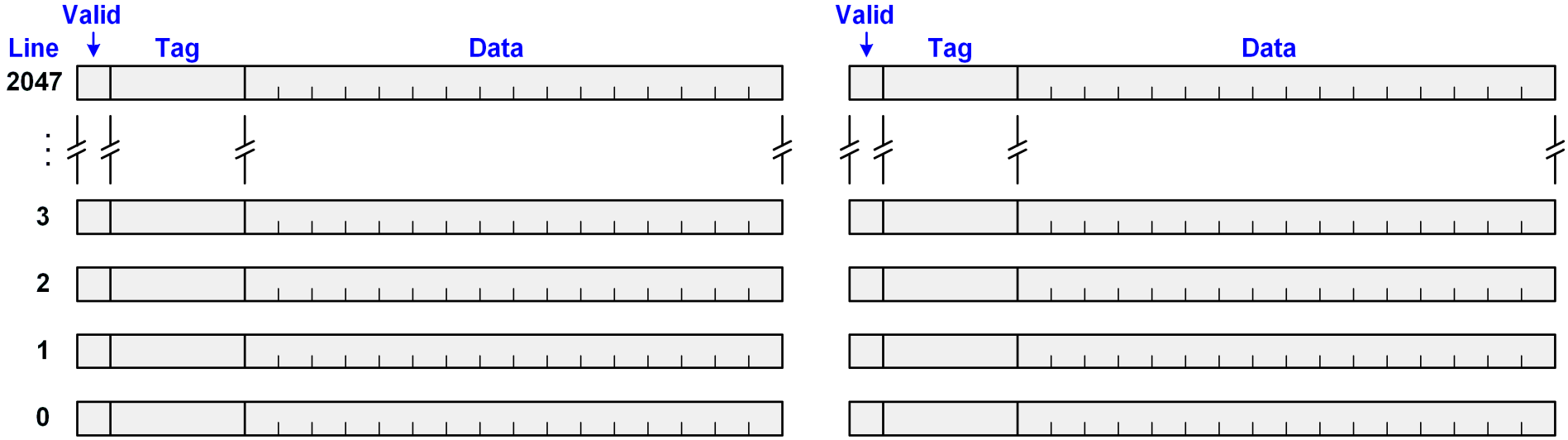
1. **How much main memory can the system access?**
   - **A – 256 bytes**
   - **B – 1K bytes**
   - **C – 64K bytes**
   - **D – 1M bytes**
   - **E – 16M bytes**
   - **F – 1G bytes**
   - **G – 4G bytes**
2. **How many bytes are there in each cache line?**
   - **A – 1 bytes**
   - **B – 2 bytes**
   - **C – 4 bytes**
   - **D – 8 bytes**
   - **E – 16 bytes**
   - **F – 32 bytes**
   - **G – 64 bytes**
3. **How many cache lines are there?**
   - **A – 4 lines**
   - **B – 8 lines**
   - **C – 16 lines**
   - **D – 256 lines**
   - **E – 1K lines**
   - **F – 64K lines**
   - **G – 64 bytes**
4. **How many bytes are in the cache?**
   - **A – 256 bytes**
   - **B – 1K bytes**
   - **C – 4K bytes**
   - **D – 8K bytes**
   - **E – 16K bytes**
   - **F – 64K lines**
   - **G – 1M bytes**

What cache line would data for hex address 1C5B4 be found in?

A – 1C **B – C5** **C – 5B** **D – B4** **E – 04** **F – none of these**

Set Associative Caches

A Set Associative cache is a compromise between the limitations of a direct mapped cache and the complexity of building a fully associative cache. A Set Associative cache still uses part of the address to determine which cache line to use, but it provides two or more entries in each cache line that can be used to store copies of data:



This example shows a 2-way Set Associative cache with 2048 cache lines, two entries per line, and 16 bytes per entry.

Set Associative caches can be built with 2, 4, or 8 entries per cache line (these would be called 2-way, 4-way, or 8-way Set Associative caches). The benefits start to fall off above 4 entries per cache line and so caches with more entries are rarely seen.

Set associative caches must solve the same problems as fully associative caches (searching multiple entries simultaneously, deciding which entry to replace on a cache miss), but since there are only a small number of entries per line these problems are much easier to solve.

2-Way Set Associative Cache Example

Cache
(64 bytes)

Cache Line	Valid	Tags (11 bits)	Data (32 bits)				Valid	Tags (11 bits)	Data (32 bits)			
111	0						0					
110	1	00000000001	38	39	3A	3B	1	00000000000	18	19	1A	1B
101	1	00000000000	14	15	16	17	0					
100	0						1	00000000001	30	31	32	33
011	0						0					
010	1	00000000001	28	29	2A	2B	1	00000000010	48	49	4A	4B
001	1	00000000010	44	45	46	47	0					
000	1	00000000000	00	01	02	03	1	00000000010	40	41	42	43

Main Memory
(65,536 Bytes)

00000000010 01000	48	49	4A	4B
00000000010 00100	44	45	46	47
00000000010 00000	40	41	42	43
00000000001 11100	3C	3D	3E	3F
00000000001 11000	38	39	3A	3B
00000000001 10100	34	35	36	37
00000000001 10000	30	31	32	33
00000000001 01100	2C	2D	2E	2F
00000000001 01000	28	29	2A	2B
00000000001 00100	24	25	26	27
00000000001 00000	20	21	22	23
00000000000 11100	1C	1D	1E	1F
00000000000 11000	18	19	1A	1B
00000000000 10100	14	15	16	17
00000000000 10000	10	11	12	13
00000000000 01100	0C	0D	0E	0F
00000000000 01000	08	09	0A	0B
00000000000 00100	04	05	06	07
00000000000 00000	00	01	02	03

The first set of data for cache line 010 holds a copy of the four data bytes starting at address 00000000001 010

- Each cache line can hold copies of data from two different areas
- Shaded cache lines (010) could potentially hold data from any of the shaded lines in main memory
- A memory reference to (00000000000 01000) would replace one of the two sets of data for cache line 010 with new information and change it's tag.

Cache Replacement Algorithms

In a Set Associative Cache, when a cache miss occurs the cache must decide which entry to replace when reading the new data from memory. There are many potential ways to make this decision, but the most common two are:

Least Recently Used (LRU)

Each memory read that accesses a cache entry is tracked. When an entry has to be replaced the one used least recently (i.e., the one that hasn't been accessed in the longest period of time) is chosen.

This method gives good results, but requires that the cache keep track of memory accesses and maintain an "LRU list" of entries to determine which entry is the lest recently used.

First In, First Out (FIFO)

This is a simpler algorithm that simply uses a round-robin approach. For example, in a 4-way set associative cache the cache will choose to replace entry 1 first, then entry 2, then entry 3, then 4, then back to entry 1 again.

Other cache replacement algorithms (such as "Least Frequently Used") also exist, but tend to be more complex to administer and are usually restricted to software-based caches (such as caching web pages).

LRU Algorithm Example

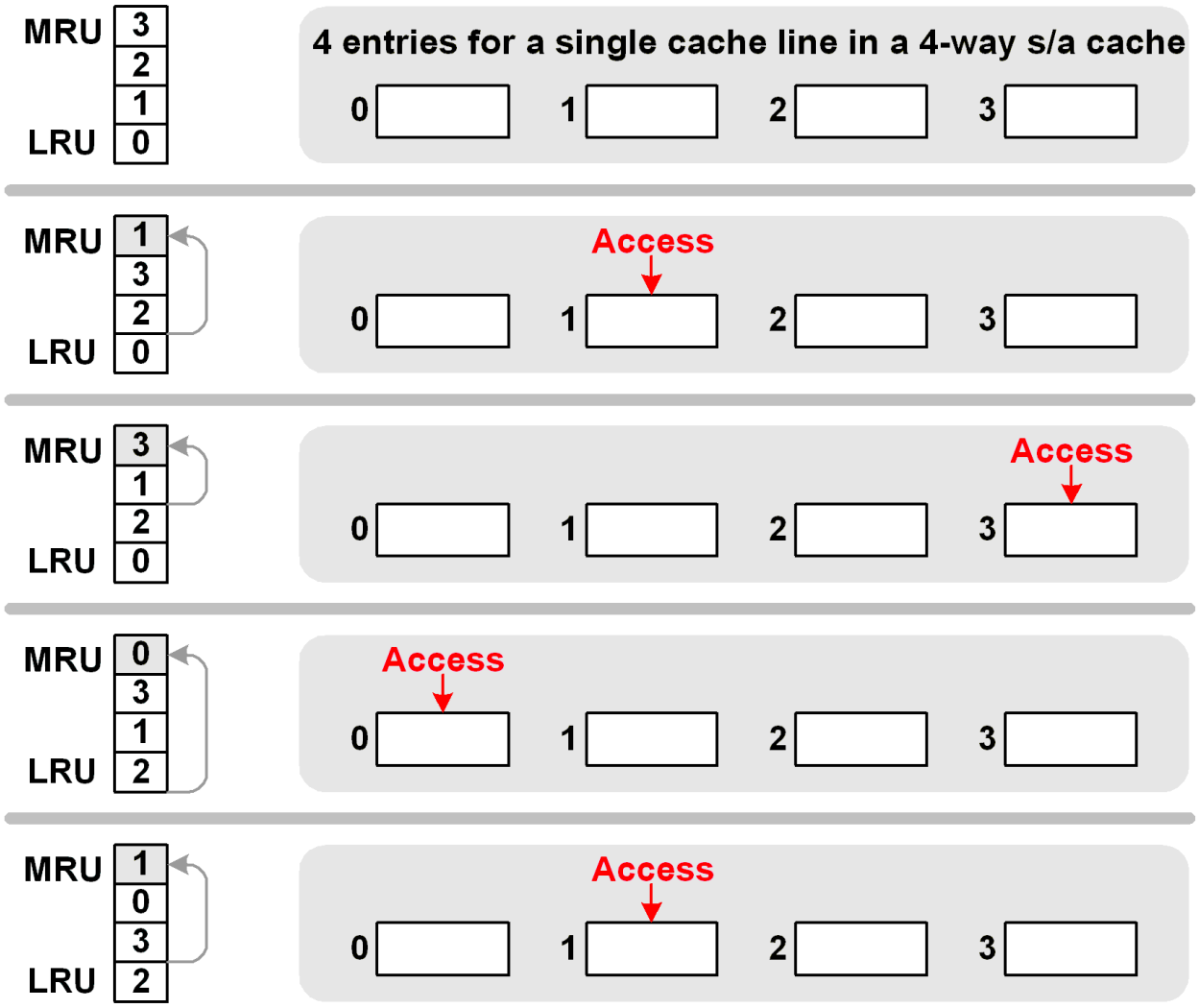
This example shows how the LRU algorithm works for a 4-way set associative cache. A list is kept for each cache line that shows the order in which entries on that line were used.

Each time a read is issued that is found in a given entry, that entry's number is moved to the Most Recently Used (MRU) end of the list.

After several accesses, the list shows that entry "2" is the Least Recently Used.

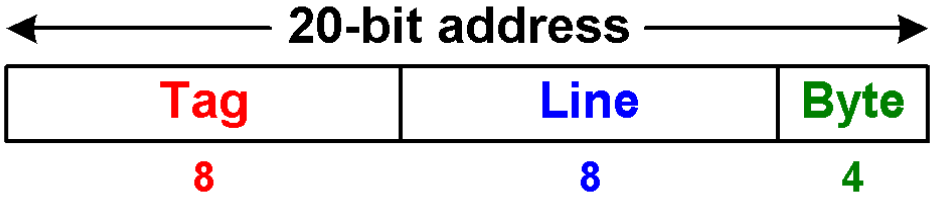
If a cache miss occurred for this cache line, the cache would choose to replace entry 2 because it's in the LRU position of the list.

When the cache reads new data into entry 2 it also moves entry 2's number to the MRU end of the list since that entry has just been accessed.



Exercise 2 – Set-Associative Cache

A system has a four-way set associative cache which interprets memory addresses as follows:



1. How much main memory can the system access?	2. How many bytes are there in each cache entry?	3. How many cache lines are there?	4. How many bytes are in the cache?
A – 256 bytes	A – 1 bytes	A – 4 lines	A – 256 bytes
B – 1K bytes	B – 2 bytes	B – 8 lines	B – 1K bytes
C – 64K bytes	C – 4 bytes	C – 16 lines	C – 4K bytes
D – 1M bytes	D – 8 bytes	D – 256 lines	D – 8K bytes
E – 16M bytes	E – 16 bytes	E – 1K lines	E – 16K bytes
F – 1G bytes	F – 32 bytes	F – 64K lines	F – 64K lines
G – 4G bytes	G – 64 bytes	G – 64 bytes	G – 1M bytes

What cache line would data for hex address 1C5B4 be found in?

A – 1C **B – C5** **C – 5B** **D – B4** **E – 04** **F – none of these**

Cache Write Policies

So far we've only discussed how caches handle read requests from the CPU. Write requests have their own special considerations.

Writes to memory store a copy of the written data in the cache, so a subsequent read from the same address will probably be able to read the data directly from cache. But if this is true, why write the data to memory at all?

There are a couple of reasons:

- **If the data is not written to memory immediately, then it will still have to be written later on when the cache entry is replaced. So the cache must have extra status information that marks this cache entry as "dirty" (i.e., needs to be written to memory) and it must do extra work when replacing such entries (before new information is read, the old information needs to be written).**
- **In multi-CPU systems, not writing the data to memory makes it more difficult to resolve cache coherency issues (this is discussed below).**

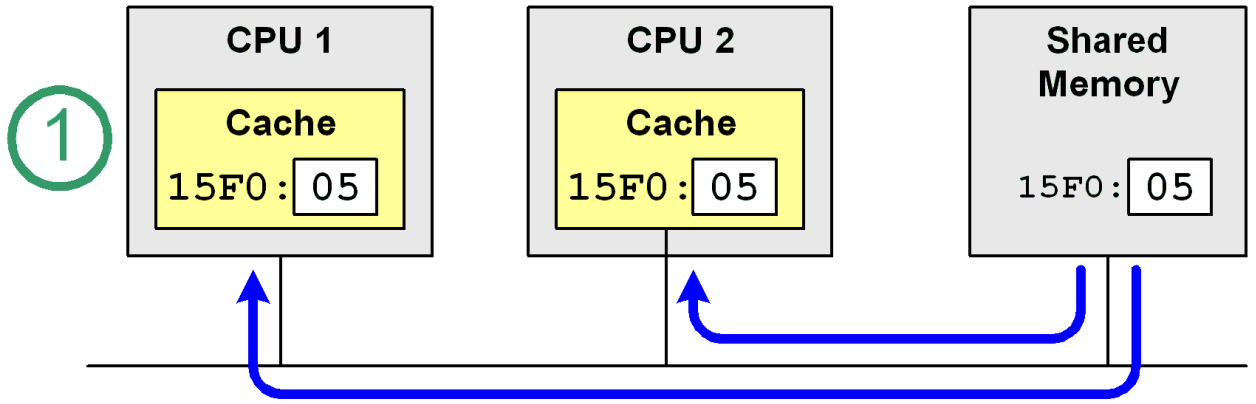
Caches that write data immediately to memory are said to use "write through", while caches that keep the data in cache until it needs to be replaced are said to use "write back" or "deferred write".

Another issue is how to handle writes to memory locations that are not contained in the cache. Some systems simply write the data and bypass the cache, while other systems write the data and read the affected cache line from memory (remember that the CPU may write one byte but that the cache lines contain groups of 4, 8 or more bytes). Caches which read in cache lines that have been written to are said to use "write allocation".

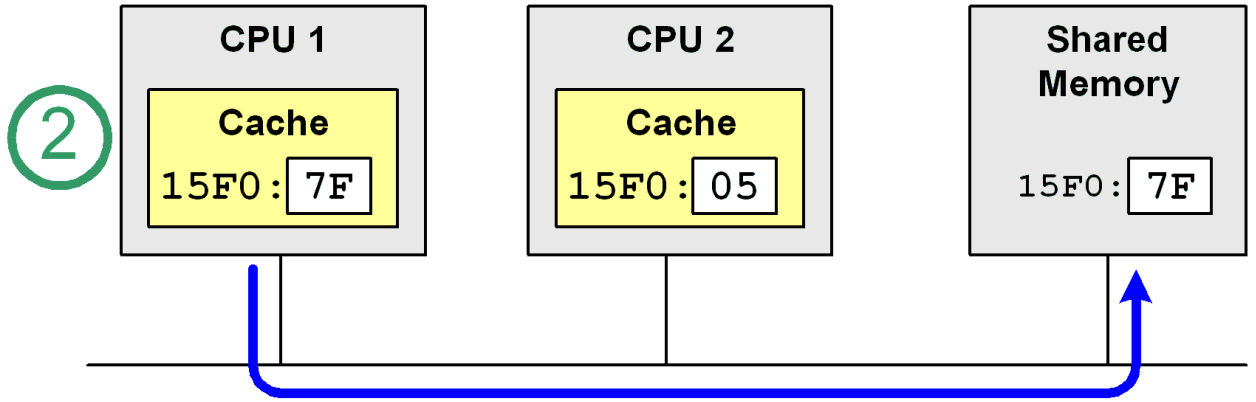
The Cache Coherency Problem

If a system has multiple CPUs and each CPU has it's own cache, then the caches in the different CPUs can get out of sync. This is known as the cache coherency problem.

1. **Both CPUs read data from memory address "15F0". Both CPU's caches now contain a copy of the memory data.**



2. **One CPU writes new information to memory address "15F0". This CPU's cache and memory contain the new data, but the other CPU's cache has stale data.**

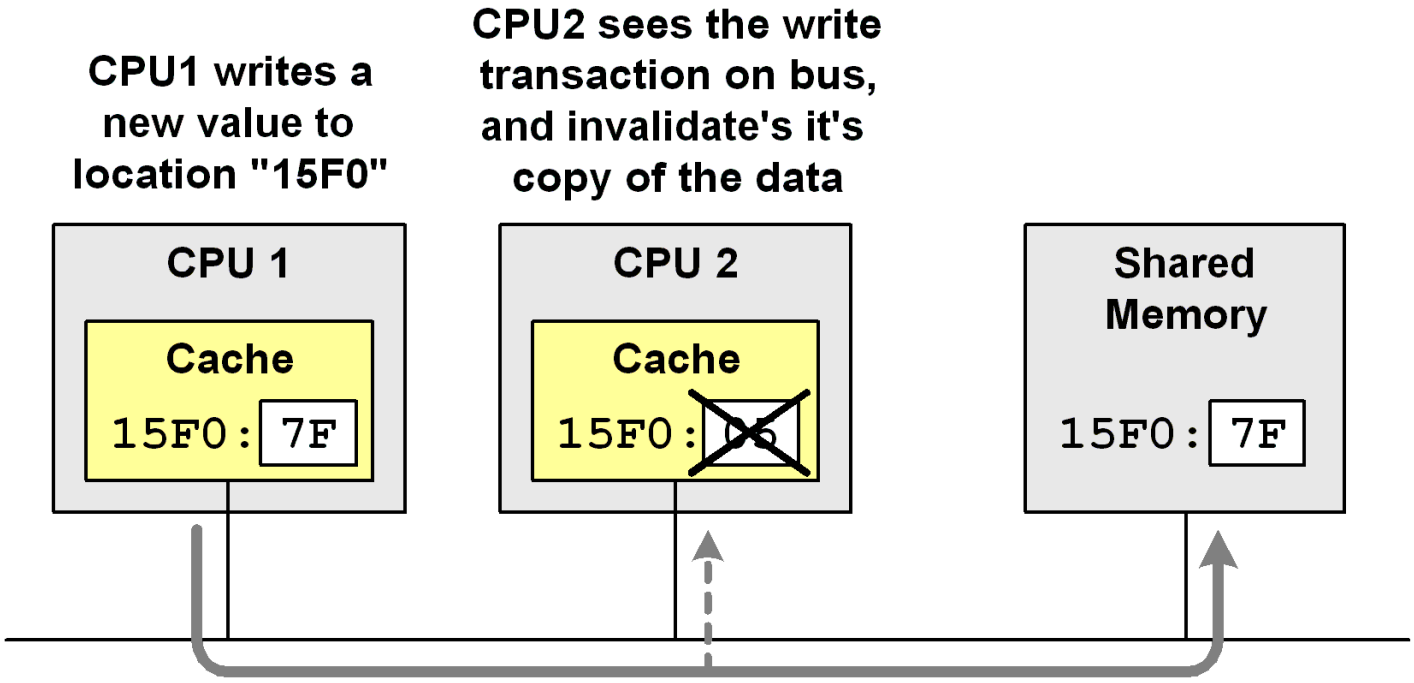


The same problem can occur with I/O devices that use DMA. If an I/O device writes data to memory addresses that are cached in the CPU, the CPU will not have a true copy of what's was put into memory by the I/O device.

Cache Coherency Solution 1

Memory Access Monitoring

One solution to the cache coherency problem is called "Memory Access Monitoring". In this solution, each CPU constantly watches the bus for memory write transactions. Any time a write is issued to memory for data which is in the CPU's cache, the CPU invalidates it's cache entry. The next time the CPU tries to read that byte it will be forced to get the data from the correct copy in memory:



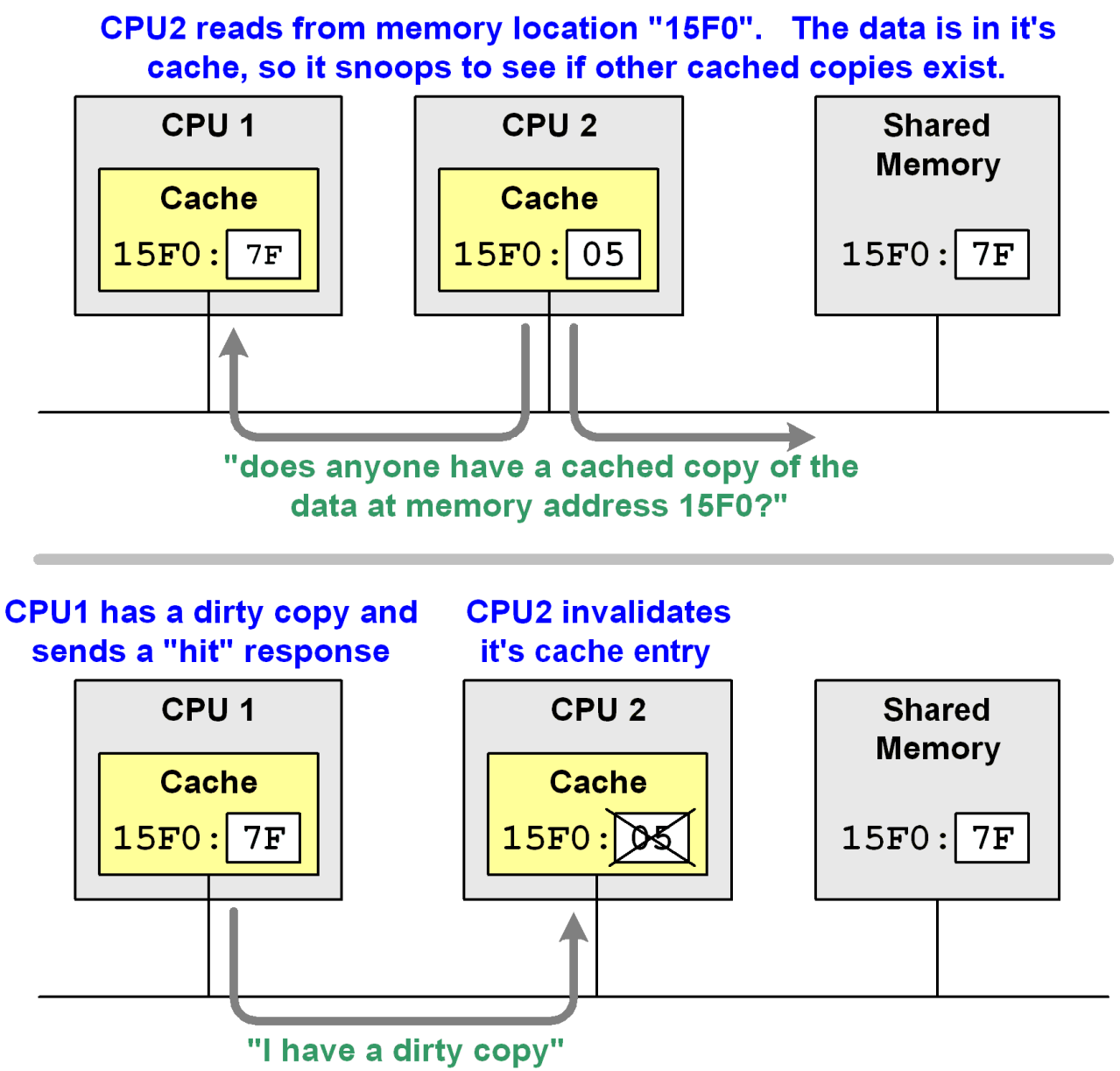
Cache Coherency Solution 2

Snooping

Another solution is called “Snooping”, which is used in Pentium and many other systems.

In this solution, any attempt to read from cached data generates a “snoop request” to other processors to see if they have a “dirty” copy of the data in their cache (i.e., a copy of data that has been modified since it was originally read from memory).

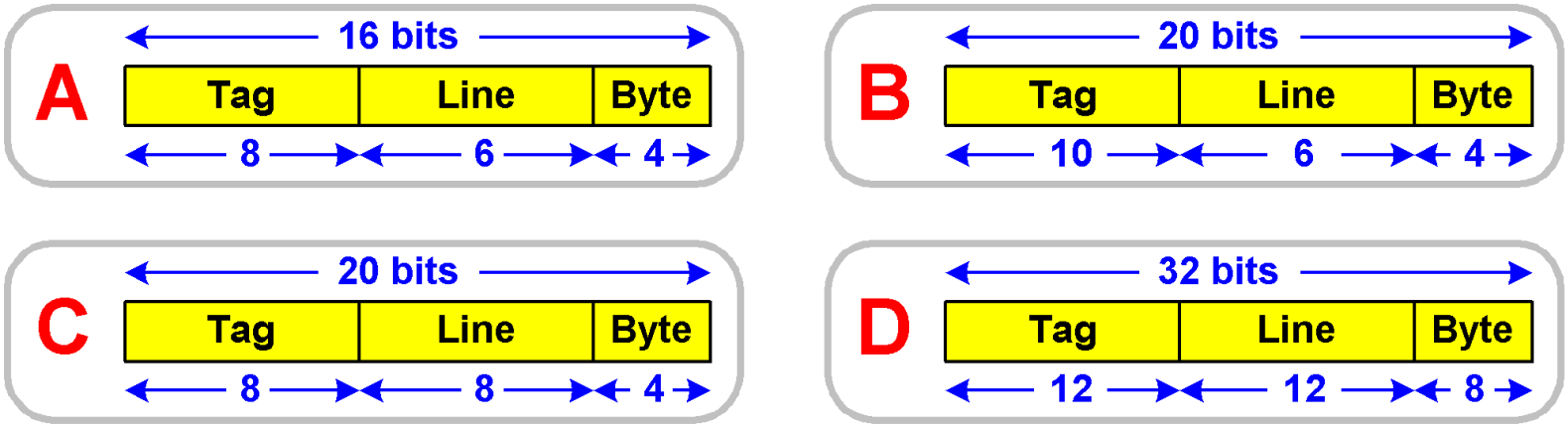
If the CPU doing the read receives a positive response, it invalidates it’s cached copy of the data, forcing it to read the correct data from memory.



Exercise 3 - Cache

A system with 1MB of addressable memory uses a direct-mapped cache with 256 lines and 16 bytes stored on each line.

1. **Which of the following diagrams shows how the cache interprets the memory addressees requested by the CPU?**



2. **How much data does the cache hold?**

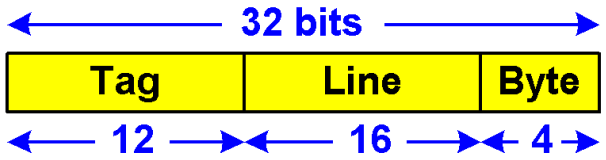
   (A) 1K bytes (B) 2K bytes (C) 4K bytes (D) 8K bytes

3. **Which cache line would data from memory address 0502F (hex) be located in?**

   (A) line 0 (B) line 2 (C) line 5 (D) line 8

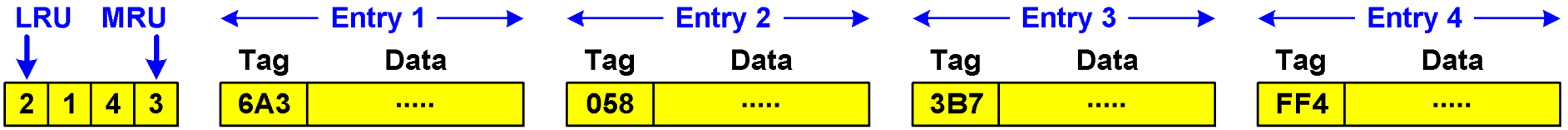
Exercise 4 – Cache

A 4-way set associative cache uses the portions of a memory address as shown at right. It receives a request to read from (hex) memory address:



073FF402

The cache line for this address is in the following state:



Which of the following occurs?

A **The data being requested resides in cache entry 4 and is returned to the CPU without having to wait for a read from memory.**

B **The data being requested is not in one of the cache entries. The cache reads the data from memory and places it into cache entry 2.**

C **The data being requested is not in one of the cache entries. The cache reads the data from memory and places it into cache entry 3.**

D **The data being requested is not in one of the cache entries. The cache reads the data from memory and places it into cache entry 4.**

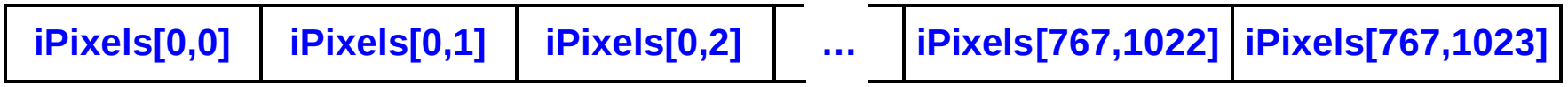
Effective Programming with Caches

Multidimensional Arrays

To maximize performance it's important to understand how data is laid out in your program and how best to access it. A common example of this is the use of multidimensional arrays. Consider a program which includes the following data declaration:

```
int     iPixels[768,1024];
```

In most languages, the data appears in memory in this order:



Now consider these two different ways of accessing the array:

```
for (iRow=0;  iRow<768, iRow++)
    for (iCol=0;  iCol<1023, iCol++)
        if (iPixels[iRow,iCol] == 0)…
```

Accesses [0,0] [0,1] [0,2]…

Good!

```
for (iCol=0;  iCol<768, iCol++)
    for (iRow=0;  iRow<1023, iRow++)
        if (iPixels[iRow,iCol] == 0)…
```

Accesses [0,0] [1,0] [2,0]…

Bad!

In the left-hand example, accessing element [0,0] will cause the cache to read in an entire cache line. If the cache lines contain 32 bytes each, then accessing element [0,0] will load array items [0,0] through [0,7] into the cache and the next 7 accesses will be at high speed.

In the right-hand example, accessing the element [0,0] will not load in element [1,0]. *Every* access will result in a very slow memory read. This will take several times longer to run than the example on the right!

Effective Programming with Caches

Arrays of Structures

Structured data can cause similar discontiguous access to memory data. Consider the following two examples:

```
struct {
  int     iKey;
  int     iData;
  char    cData[60];
} sTbl[1000];

  for( iNo=0;  iNo<1000;  iNo++)
  {
        if (sTbl[iNo].iKey == iFind)
            sTbl[iNo].iData = …;
  }
```

Index keys occur every 64 bytes

Bad!

```
    int     iKey[1000];
struct {
    int     iData;
    char    cData[60];
} sTbl[1000];

    for( iNo=0;  iNo<1000;  iNo++)
    {
        if (iKey[iNo] == iFind)
            sTbl[iNo].iData = …;
    }
```

Index keys are adjacent to each other

Good!

The example on the right takes the best advantage of cache while it is searching for the data and will run much faster than the example on the left.

These code examples are intended to show the effects of data layout, but don't forget that your choice of algorithm usually has a much bigger performance impact. For example a hash table lookup can be hundreds of times faster than the sequential key search shown above!

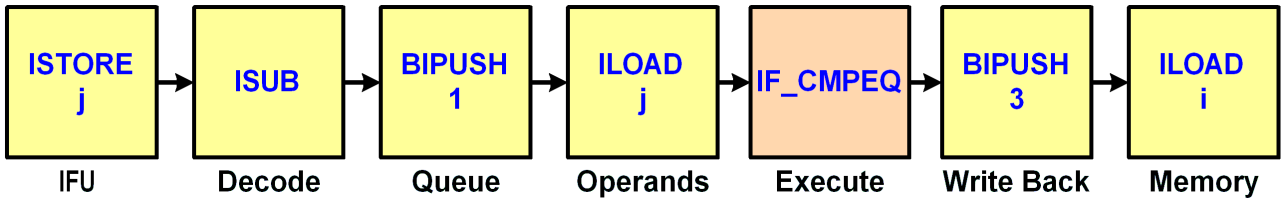
Branch Instructions

Branch instructions transfer control to a different sequence of ISA instructions. We saw examples of branch instructions in the IJVM instruction set:

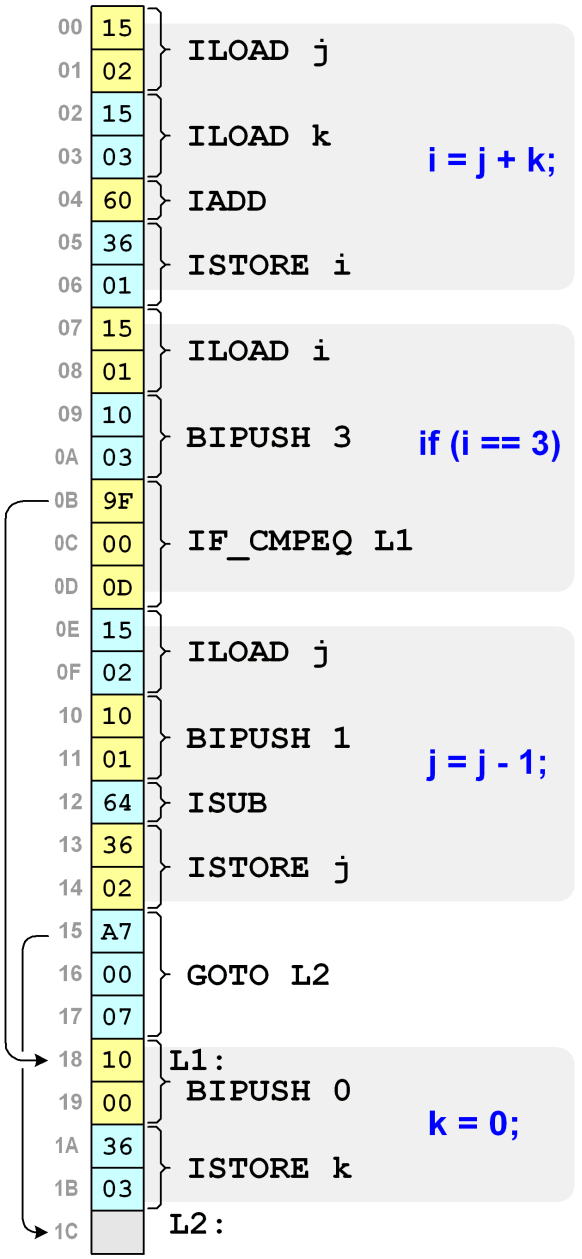
GOTO – An unconditional branch (always branches)

IF_CMPEQ – A conditional branch (branches only if top two items on the stack are equal)

The problem with branches is that they cause the pipeline to be "flushed". For example, by the time the IF_CMPEQ instruction at right is executed, a typical pipeline might look something like this:



If the execute stage determines that the IF_CMPEQ actually does branch, then the following instructions should not be executed. They will have to be removed ("flushed") from the pipeline, and the new sequence of instructions at label "L1:" will have to be fed into the pipeline from the beginning. This will waste at least several clock cycles.



Branch Prediction

To avoid pipeline flushes, most modern CPUs use one or more forms of "Branch Prediction" in order to try to get the right instructions into the pipeline as quickly as possible. In general, there are three forms of Branch Prediction:

Static Branch Prediction

Static branch prediction assumes that certain types of branch instructions are more likely to take a certain course of action (i.e., are more likely to branch, or, for other instructions, more likely to not branch).

Branch Hints

Some ISAs allow "hints" to be encoded into the branch instruction to suggest that it is likely to branch (or not to branch). This may be done by using special opcodes or by including an extra bit in the branch instruction.

Dynamic Branch Prediction

Dynamic branch prediction watches the actual behaviour of branch instructions as the program executes. Based on the observed pattern of branches / no branches, logic attempts to guess whether the next execution of the instruction will actually branch or not.

Branch prediction logic is located early in the pipeline. If the logic (using any of these techniques) indicates that the instruction is likely to branch, then the pipeline is flushed and starts to be filled with instructions at the branch target address. Because this happens earlier in the pipeline, there is less lost time than if the flush had to wait until the branch instruction reached the execution stage.

Static Branch Prediction

Static branch prediction uses the type of instruction to decide whether the branch is likely to be taken or not. The most obvious use of this is for unconditional branches. Most pipelined machines detect unconditional branches at an early pipeline stage and flush/reload the pipeline as quickly as possible in order to eliminate the performance penalty.

The UltraSPARC system eliminates the disadvantage of even an early pipeline flush by always executing the instruction following an unconditional branch. This instruction is called the "delay slot".

A good example of a conditional branch instruction suitable for static branch prediction is the Pentium "LOOP" instruction. This special-purpose instruction is used at the bottom of a loop to control how many times the loop is executed. It decrements the CX register by one and branches if the register is not zero.

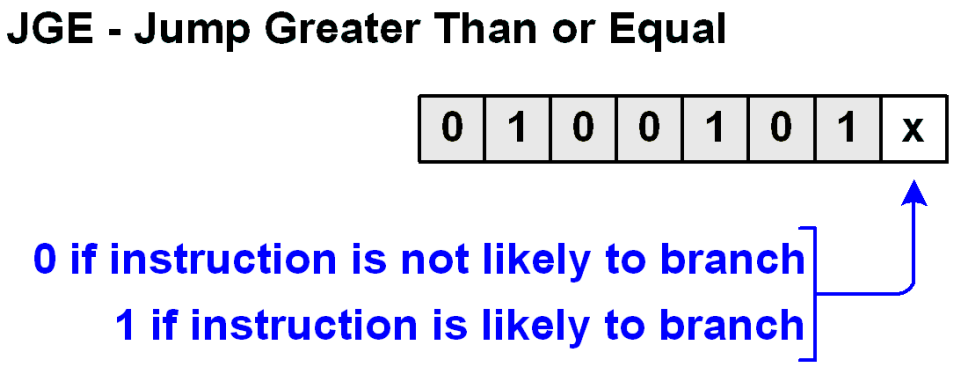
```
        MOV     CX,10
A10:
        (instructions to be
         executed in the body
         of the loop)

        LOOP    A10
```

Since this instruction usually branches (except on the last iteration of the loop), statically predicting that the instruction will branch will result in a correct prediction most of the time.

Another useful static branch prediction technique is to base the prediction on whether the conditional jump instruction jumps ahead to an instruction at a higher memory address, or jumps back to an instruction at a lower memory address.

Branch Hints

Branch hints are a special type of static branch prediction. An ISA that uses branch hints typically reserves a bit in the instruction to indicate whether the instruction is likely to branch or not. For example, the opcode for a "JGE" ("Jump Greater Than or Equal") instruction with a branch hint may look like this:



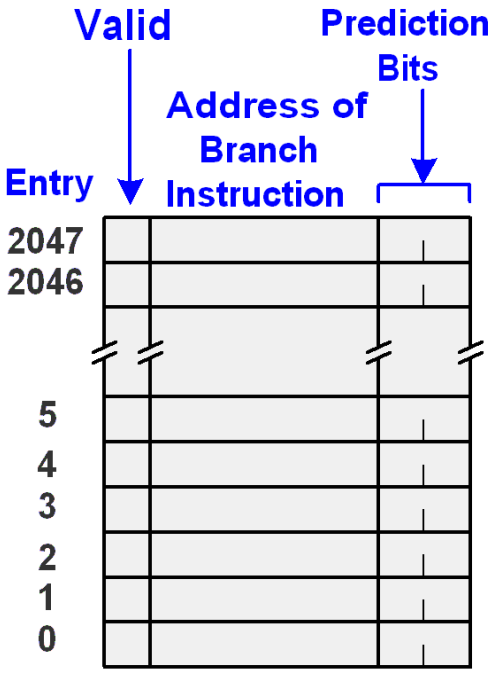
The idea is that the compiler sets the hint bit based on it's evaluation of whether the branch instruction is likely to branch or not. In theory, the compiler can analyze the code and make a prediction that's better than a fixed "likely to branch" or "not likely to branch" based on a given instruction type.

As the program runs, the pipeline treats the prediction bit like a static branch prediction – the bit is checked early in the pipeline, and if it says the instruction is likely to branch then the pipeline is flushed and refilled with the instructions at the target address.

Dynamic Branch Prediction

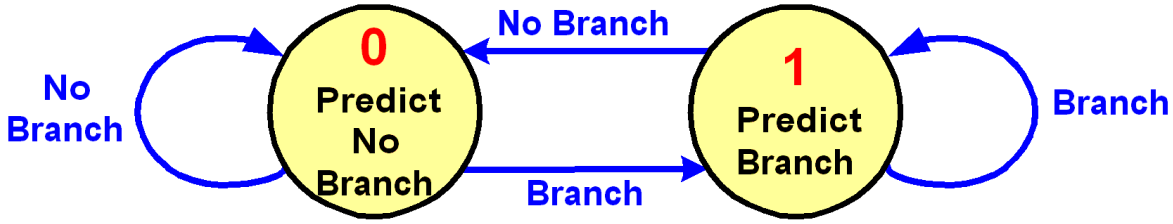
Static branch prediction techniques have limitations, so many modern systems include logic to observe the actual run-time behaviour of branch instructions and try to predict whether they will branch or not on the next execution.

There are many techniques for doing this, but they all basically involve storing a table that tracks the history of branch instructions. This table operates in much the same manner as a cache, except that the information stored isn't a copy of what's in memory, but rather it stores "Prediction Bits" which are updated each time the branch instruction is executed.



A simple branch prediction scheme uses a single prediction bit which simply records whether the instruction did or did not branch the last time it was executed (i.e., 0 = branched, 1 = didn't branch). When the machine is about to execute the instruction again, it finds the instruction's table entry and predicts that it will branch the same way this time that it did the last time.

This behaviour can be shown with a finite state machine diagram:



Dynamic Branch Prediction

Tracing Branch Prediction

Let's track the execution of the branch prediction logic using our example of the Pentium LOOP instruction (which is actually a conditional branch) shown at the right. This instruction will be executed five times – the first four times the instruction will branch and the fifth time it will not:

```
        MOV     CX,5
A10:
        :
        :
        LOOP    A10
```

#	Actual Result	Mispredicted?	Branch Prediction State
			0 – No Branch
1	Branch	Yes	
			1 – Branch
2	Branch	No	
			1 – Branch
3	Branch	No	
			1 – Branch
4	Branch	No	
			1 – Branch
5	No Branch	Yes	
			0 – No Branch

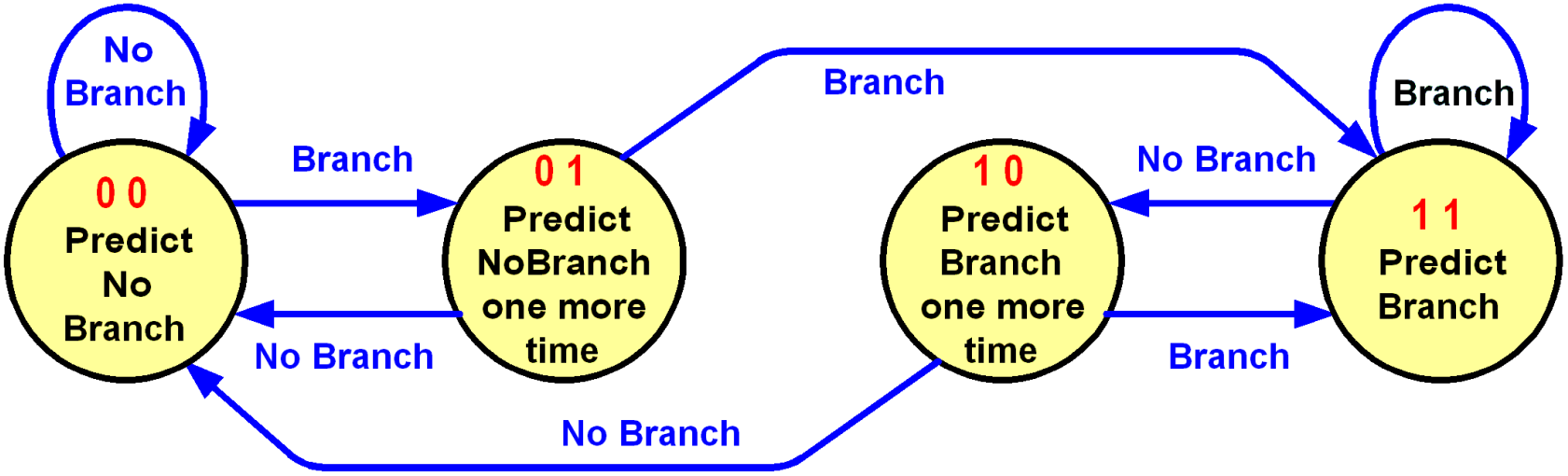
With this simple branch prediction technique, we've correctly predicted 3 of the 5 branches – (only slightly better then if we just tossed a coin). Notice that we have left the prediction state as "No Branch" – the next time we enter this loop we will mispredict the first branch again.

Dynamic Branch Prediction

Multiple Prediction Bits

When an instruction *usually* branches (or doesn't branch), a single prediction bit doesn't give very good results because it alwa ys generates two mispredictions – one for the unexpected branch and the second because the unexpected branch changed the branch prediction bit state.

To avoid this problem we can use two branch prediction bits and give the instruction a "second chance":



For our example, this finite state machine will only generate one misprediction per complete loop.

Many CPUs use even more prediction bits and sophisticated algorithms to try to detect more complex branch patterns and eliminate as many mispredictions as possible.

Dynamic Branch Prediction

Multiple Branch Prediction Bits

Let's look again at our sample loop and see how it works on a machine with two branch prediction bits using the Finite State Machine described on the previous page.

```
        MOV     CX,5
A10:
          :
          :
        LOOP    A10
```

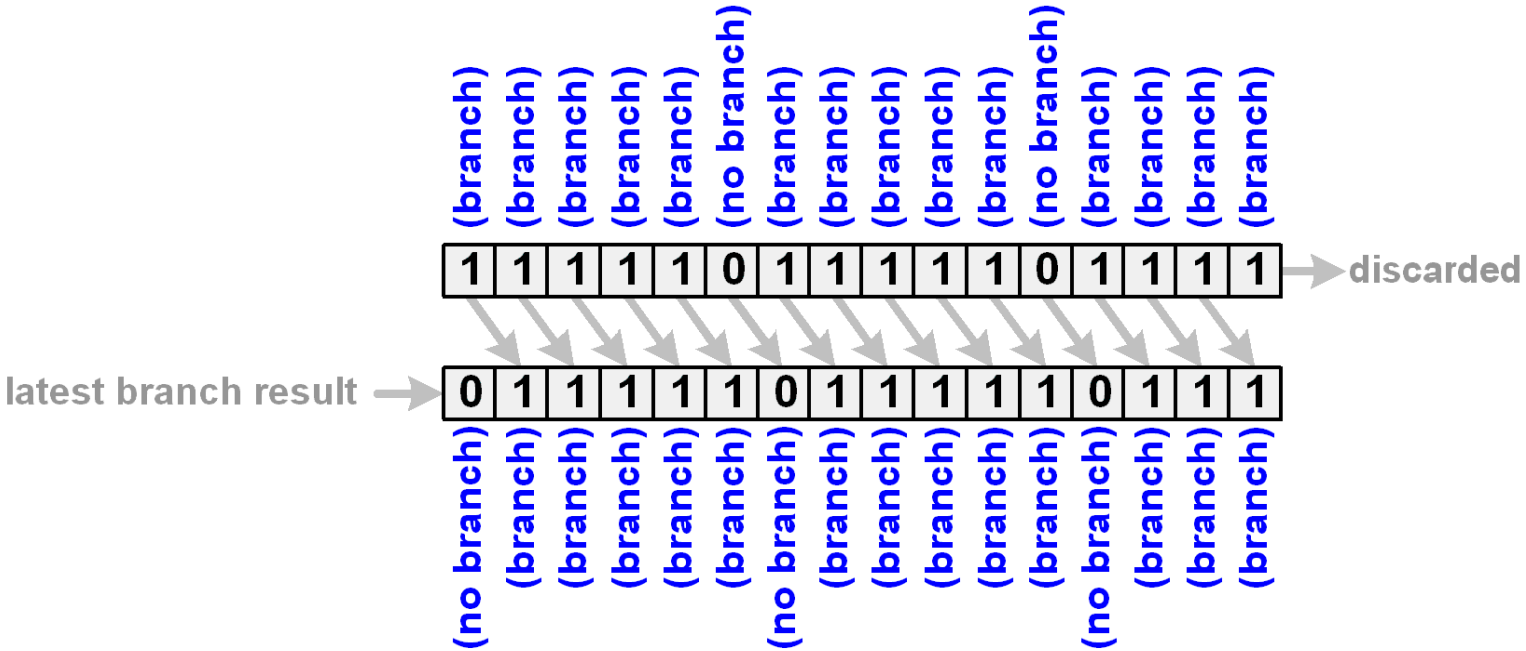
#	Actual Result	Mispredicted?	Branch Prediction State
			10 – Branch once more
1	Branch	No	
			11 – Branch
2	Branch	No	
			11 – Branch
3	Branch	No	
			11 – Branch
4	Branch	No	
			11 – Branch
5	No Branch	Yes	
			10 – Branch once more

Each time the loop is executed, the last iteration of the loop is mispredicted. Note that the branch prediction bits for the LOOP instruction are left in a "Branch once more" prediction state so that the next time the loop is executed the first branch is correctly predicted.

Dynamic Branch Prediction

Shift Registers

Another technique is the use of a Branch Prediction Shift Register. The idea here is to have a series of bits which tracks the actual history of branches vs. no branches for each instruction:



By noting when the branches occur, the branch prediction logic can anticipate when the next "no branch" occurrence will happen. This type of branch prediction is more complex, but it can eliminate all mispredictions for loops that always have the same number of iterations.

Programming to Eliminate Branches

Reducing the number of branches you program executes will improve it's speed on modern CPU architectures. Here are a couple of examples of how this can be done:

Loop Unrolling

Reduces the number of loop tests and branch instructions executed

Instead of:

```
for (i = 0;  i < 50;  i++)
    tbl[i] = 0;
```

Use:

```
for (i = 0;  i < 50;  )
{
    tbl[i++]=0;
    tbl[i++]=0;
    tbl[i++]=0;
    tbl[i++]=0;
    tbl[i++]=0;
}
```

Code Hoisting

…eliminates conditional branch from the body of the loop

Instead of:

```
for (i = 0;  i < max;  i++)
    if (no > 0) tbl[i] = val/no; else tbl[i] = 0;
```

Use:

```
if (no > 0) ival = val/no; else ival = 0;
for (i = 0;  i < max;  i++)  tbl[i] = ival;
```

Some compilers will do these kinds of optimizations for you automatically as they compile your code. It pays to know which optimizations your compiler can and can't do.

Exercise 5 – Branch Prediction

1. **Branch prediction is important in modern computer architectures because:**

   A – it prevents the CPU from executing the wrong instructions in a program

   B – it avoids having to flush and re-fill the pipeline

   C – it allows more instructions to be put into the pipeline at once

   D – all of the above

2. **Branch prediction techniques used in modern architectures include:**

   A – static branch prediction

   B – branch prediction hints

   C – dynamic branch prediction

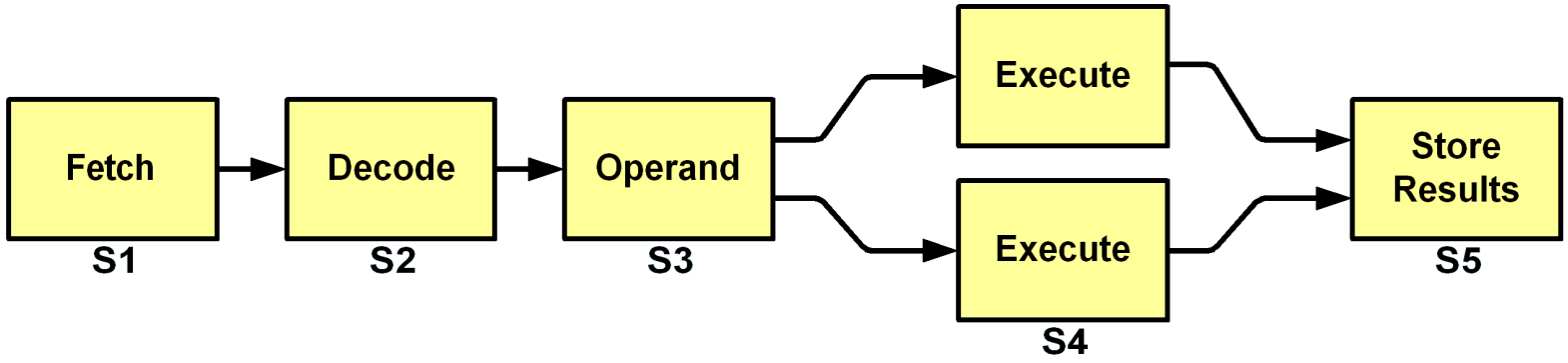
   D – A and B above

   E – B and C above

   F – all of the above

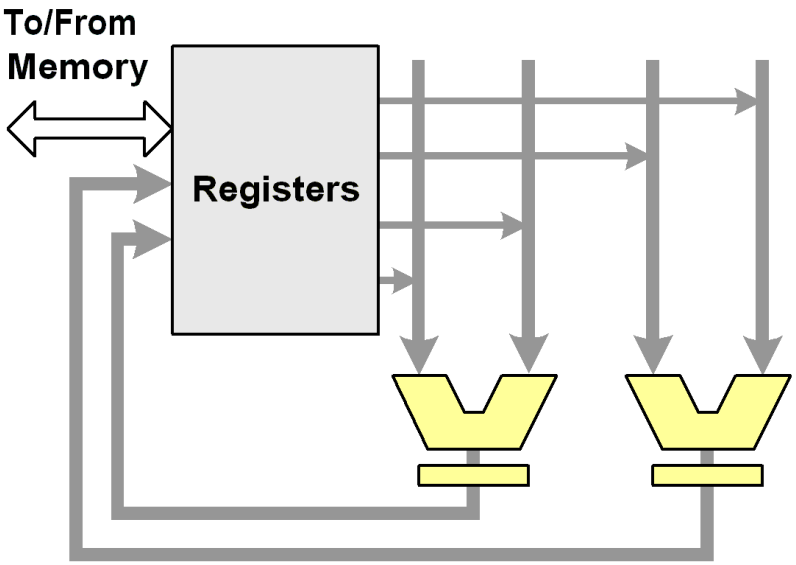
Superscalar Execution

We saw how the pipeline in the MIC-3 architecture improved performance by breaking up the work of instruction execution into small tasks and doing them in parallel. Another performance improvement technique is to actually replicate the entire execution stage so that two or more instructions can execute at the same time. Such a system could have a pipeline that might look something like this:



In order for this pipeline to be successful, the other stages (Fetch, Decode, etc.) would have to be faster than the Execute stages in order to keep them supplied with work.

The data path for this type of machine might look something like the diagram to the right:



Instruction Dependencies

Introducing a second execution unit is not without it's pitfalls. The most important problem that needs to be solved is how to deal with instruction dependencies.

To illustrate instruction dependencies, we'll use a hypothetical system which has eight registers named R0 through R7. This system can execute simple arithmetic instructions which we will write in a format like:

R0 = R1 + R2 **(Add R0 and R2 and store the result in R2)**

Note that in this example the R0 register is being written to, and the R0 and R1 registers are being read.

Now imagine that we're executing the following two instructions at the same time:

R0 = R1 + R2

R5 = R0 * R3

Do you see the problem? The second instruction will start using the value in the R0 register as part of it's calculation while the first instruction is still executing. This means that the first instruction hasn't stored it's result in R0 yet. Because of this, the second instruction will use the old R0 value and end up with the wrong answer.

This situation is known as a "Read After Write", or "RAW" dependency. A superscalar architecture must be able to detect this type of situation and stall the second instruction until the first instruction has finished it's work.

Instruction Dependencies

There are several types of instruction dependencies which a superscalar architecture must be able to detect and handle:

Type of dependency	Acronym	Example
Read After Write	RAW	R0 = R1 + R2 R5 = R0 * R3
Write After Read	WAR	R0 = R1 + R2 R2 = R5 – R6
Write After Write	WAW	R0 = R1 + R2 R0 = R3 + R4

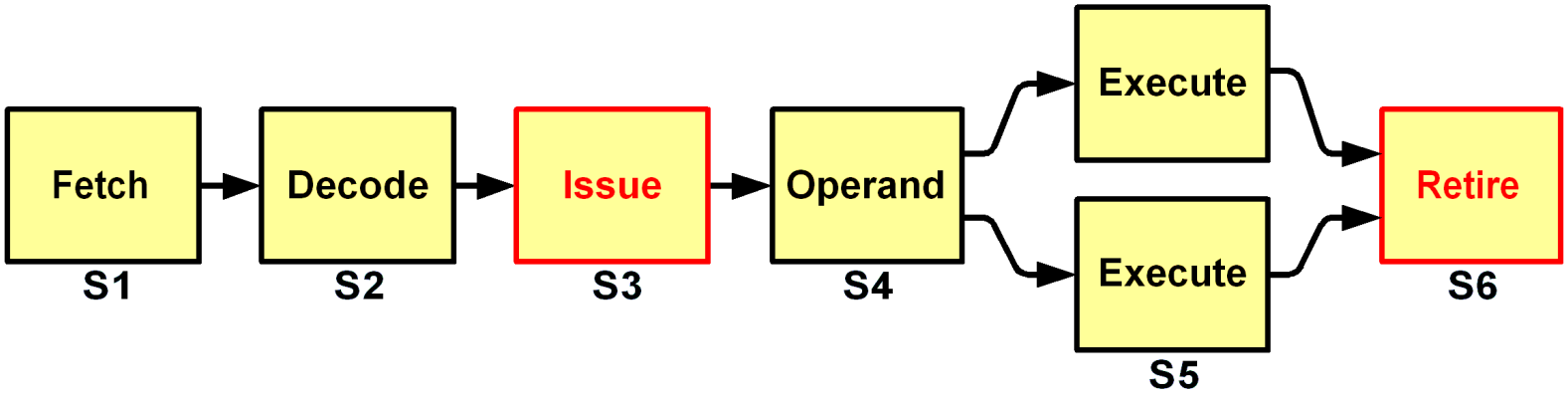
Note that the last (WAW) example isn't very likely – there's no need to execute the first instruction if the second instruction is simply going to overwrite it's result. WAW dependencies usually occur with at least intermediate instruction (which also introduces other dependencies), as in:

```
R0 = R1 + R2
R5 = R0 – R2
R0 = R3 + R4
```

Our dependency examples are overly simple because real programs usually have other instructions like branches intermixed with simple arithmetic ones – but we're going to ignore those so we can concentrate on how to solve dependency problems.

Dealing With Dependencies

When a dependency is detected, the pipeline has to stall the dependent instruction until it's safe for it to execute. This requires another pipeline stage to detect the dependency and decide when the instruction can be "issued" (start executing):



This new stage is quite complex – it has to do several things:

- **Detect dependencies between instructions**
- **Provide a place to "hold" dependent instructions while they're waiting for dependencies to be cleared**
- **Send instructions to the next pipeline stage when it's safe to do so.**

We've also renamed the final stage to "Retire" – this is to reflect the fact that when an instruction is retired, there is some extra bookkeeping work to do in order to track the dependencies.

Detecting Dependencies

Dependencies are detected using a "scoreboard" to track which registers are being read and written.

Read Counts

R0	R1	R2	R3	R4	R5	R6	R7
1	0	2	0	1	0	0	3

Write Counts

R0	R1	R2	R3	R4	R5	R6	R7
0	1	0	1	0	1	0	0

The scoreboard works as follows:

- **There is a "read" and a "write" counter for every register (the "write counts" are actually just single bits since it's never valid to have two instructions writing to the same register at the same time).**
- **As instructions are issued, the "read" and "write" counters are incremented for the registers that this instruction is reading and writing.**
- **As instructions are retired, the "read" and "write" counters are decremented for the registers that the instruction read or wrote.**

The Issue stage can detect dependencies for the instruction it's about to issue by checking the scoreboard for the following conditions:

- **This instruction reads from a register whose "write count" is nonzero (RAW dependency)**
- **This instruction writes to a register whose "read count" is nonzero (WAR dependency)**
- **This instruction writes to a register whose "write count" is nonzero (WAW dependency)**

Register Renaming

We can eliminate Write-after dependencies by choosing a different output register to store the result in. But there are two problems with just arbitrarily changing one register to another like this:

R0 = R1 + R2	**→**	**R0 = R1 + R2**
R1 = R3 + R4	**→**	**R5 = R3 + R4**

1. **The program might be using register R5 to hold other data it needs.**
2. **Later instructions will look in register R1 for the result of the 2nd instruction.**

But we can still change the output register without affecting the execution of the program. The way we do it is to have a bank of "secret" registers where all of the information is actually stored. There are more "secret" registers than the registers used by the program, so that the eight registers R0 through R7 used by the program may be changed to one of sixteen "secret" registers named S0 through S15.

When a program instruction specifies a specific register name such as "R0" or "R5", the decode unit changes the instruction to use the "secret" register instead. Here's an example of how this might work:

R0 = R1 + R2	**→**	**S0 = S1 + S2**
R1 = R3 + R4	**→**	**S8 = S3 + S4**
R6 = R1 * R2	**→**	**S6 = S8 * S2**

The decode unit is converting registers specified by the program to the "secret" registers actually used by the hardware. When it detects the Write-After-Read dependency in the second instruction, it uses a different secret register to store the result. Subsequent instructions that depend on the result are also change to use the different secret register.

Register Renaming

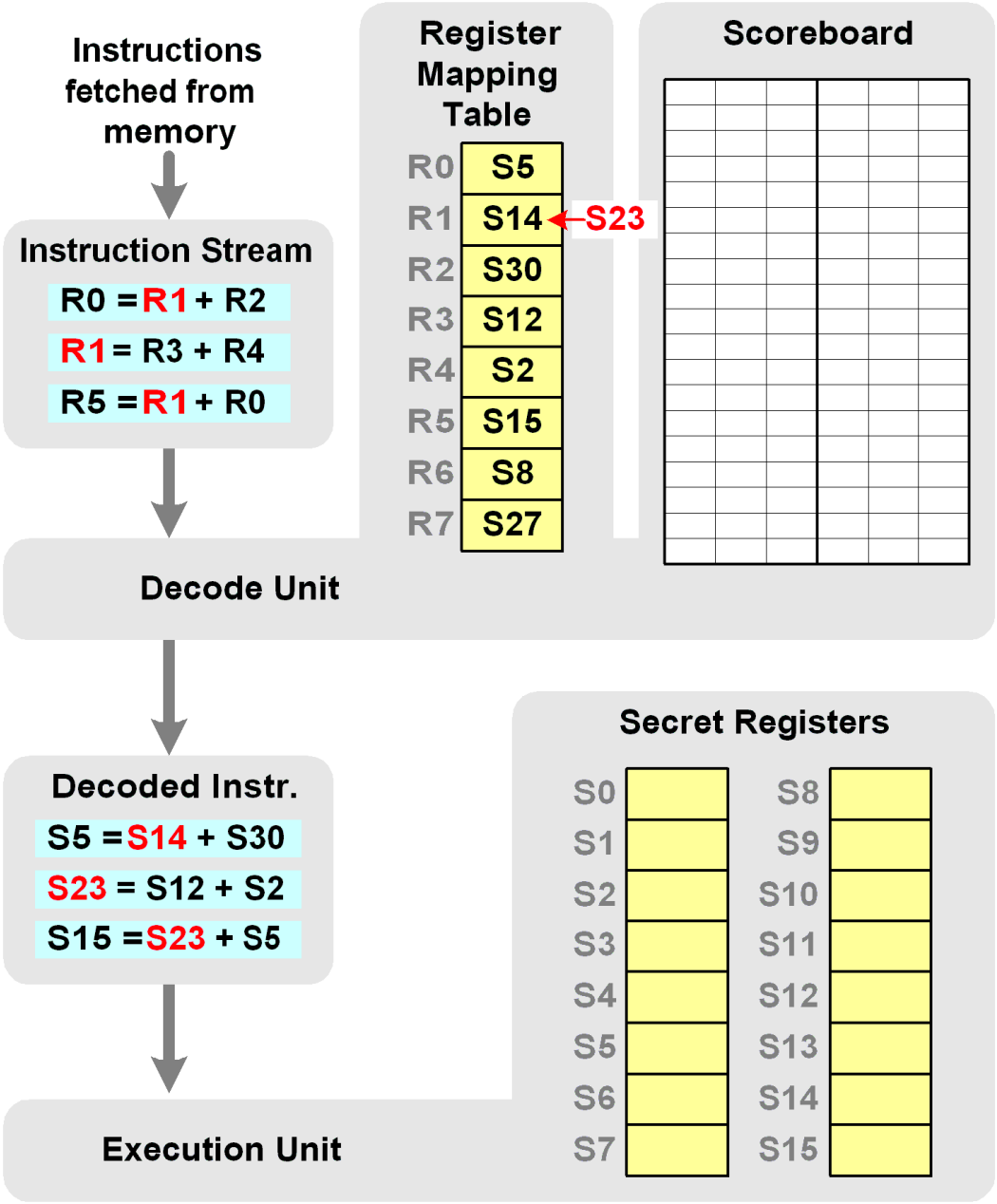
A system which uses register renaming has a mapping table which controls what the registers specified by the instruction are changed to.

As instructions are decoded, the registers are changed from the what was specified by the instruction to the "secret" register indicated in the mapping table.

When the decode unit discovers a Write-After dependency (by checking the scoreboard), it changes the mapping table for the affected register to an new, unused "secret" register.

That instruction and all subsequent instructions that specify the register will then be decoded to use the new "secret" register.

By the time the instructions reach the execution unit, all Write-After dependencies have been eliminated.



Instruction Reordering

Another way to avoid stalling instructions is to allow them to be issued and retired in a different order than they appear in the instruction stream. For example, these instructions:

```
R0 = R1 + R2
R3 = R4 * R5
```

…have no dependencies between them. If the first instruction is stalled due to a dependency on some previous operation, there really isn't any reason we can't go ahead and start executing the second instruction anyway. The same applies to retiring instructions.

We will go through the scoreboard example again, this time using register renaming and instruction reordering, to see how these techniques improve performance. As we do so, note the following:

- **The register names in the scoreboard actually refer to the secret registers, not the registers in the instruction stream.**
- **This time the example shows the same instructions as before, but with the registers renamed to use the appropriate "secret" registers:**

1	R3 = R0 * R1		S3 = S0 * S1
2	R4 = R0 + R2		S4 = S0 + S2
3	R5 = R0 + R1		S5 = S0 + S1
4	R6 = R1 + R4		S6 = S1 + S4
5	R7 = R1 * R2	→	S7 = S1 * S2
6	R1 = R0 – R2		S8 = S0 – S2
7	R3 = R3 * R1		S3 = S3 * S8
8	R1 = R4 + R4		S9 = S4 + S4

Scoreboard Example with Renaming / Reordering

					Regs being read										Regs being written									
Cy	#	Decoded	Iss	Ret	0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9
1	1 2	S3 = S0 * S1 S4 = S0 + S2	1 2		1 2	1 1	 1											1 1	 1					
2	3 4	S5 = S0 + S1 S6 = S1 + S4	3 –		3 3	2 2	1 1											1 1	1 1	1 1				
3	5 6	S7 = S1 * S2 S8 = S0 – S2	5 6	 2	3 4 3	3 3 3	2 3 2											1 1 1	1 1 0	1 1 1		1 1 1	 1 1	
4	7 8	S3 = S3 * S8 S9 = S4 + S4	4 – 8	 1 3	3 3 3 2 1	4 4 4 3 2	2 2 2 2 2		1 1 3 3 3									1 1 1 0		1 1 1 1 0	1 1 1 1 1	1 1 1 1 1	1 1 1 1 1	 1 1 1
5				6	0	2	1		3							1					1	1	0	1
6			7	 4 5 8		2 1 0	1 1 0	1 1 1 1	3 2 2 0				1 1 1 1			1 1 1 0		1 1 1			1 0	1 1 0		1 1 1 0
7								1					1					1						
8								1					1					1						
9				7				0					0					0						

Renaming / Reordering

We can now see that the eight instructions completed in 9 clock cycles. What's even more interesting is that five instructions were issued within 3 clock cycles – so we are actually getting some benefit from the parallel execution capability of our superscalar architecture.

It should be evident that register renaming and instruction reordering are key requirements for building a high performance superscalar machine.

Instruction reordering comes at a price – it complicates the reporting and handling of exceptions. Exceptions occur when a program instruction causes an error (such as divide-by-zero). If instructions have been reordered, then when an exception is raised some instructions that follow the exception-causing one may already have stored their results:

R0 = 0
R5 = R1 / R0 ← **causes an exception**
R6 = R2 + R1 ← **may complete before exception is reported**

There are two ways to deal with this:

- **Force instructions to retire in order, even though they may be issued in order. This has a performance penalty, but is the easiest solution and is quite common.**
- **Note that exceptions are "not precise". In other words, the program can not depend on an exception being reported on the instruction that causes it. This makes it more difficult for software that needs to deal with exceptions – but since it gives better performance it is a technique that is used on several RISC processors.**

Exercise 6 – Detecting Dependencies

The scoreboard in a CPU currently has the following information:

Read Counts

R0	R1	R2	R3	R4	R5	R6	R7
1	0	2	0	1	0	0	3

Write Counts

R0	R1	R2	R3	R4	R5	R6	R7
0	1	0	1	0	1	0	0

For each of the instructions below, is there:

A **A Read-After-Write (RAW) dependency?**

B **A Write-After-Write (WAW) dependency?**

C **A Write-After-Read (WAR) dependency?**

D **No dependency**

(Assume that the scoreboard starts with exactly the same counts for each instruction)

1. **R0 = R2 + R6**
2. **R3 = R7 – R0**
3. **R6 = R4 * R2**
4. **R1 = R5 / R6**

Exercise 7 – Register Renaming

The purpose of register renaming is to:

A – eliminate "Write after read" dependencies

B – eliminate "Write after write" dependencies

C – eliminate "Read after write" dependencies

D – eliminate "Read after read" dependencies

E – A and B above

F – A, B and C above

G – all of the above

H – none of the above

Speculative Execution

Speculative execution is one of the last performance enhancement techniques to be introduced. It attempts to extend some of the gains in pipelining, out-of-order execution, and register renaming across "control block" boundaries in the code. Control block boundaries are points at which the code executes branch instructions.

The basic idea behind speculative execution is that when a conditional branch occurs, one of two possible code paths will be executed. Rather than waiting until the branch actually occurs, the system can start to "speculatively execute" some of the instructions following the branch. This is especially useful if these are lengthy instructions like memory reads.

If it turns out that the speculatively executed instructions actually do need to be executed, then the system has gained a head start. If they don't need to be executed, the system can discard the results.

There are two problems with speculative execution:

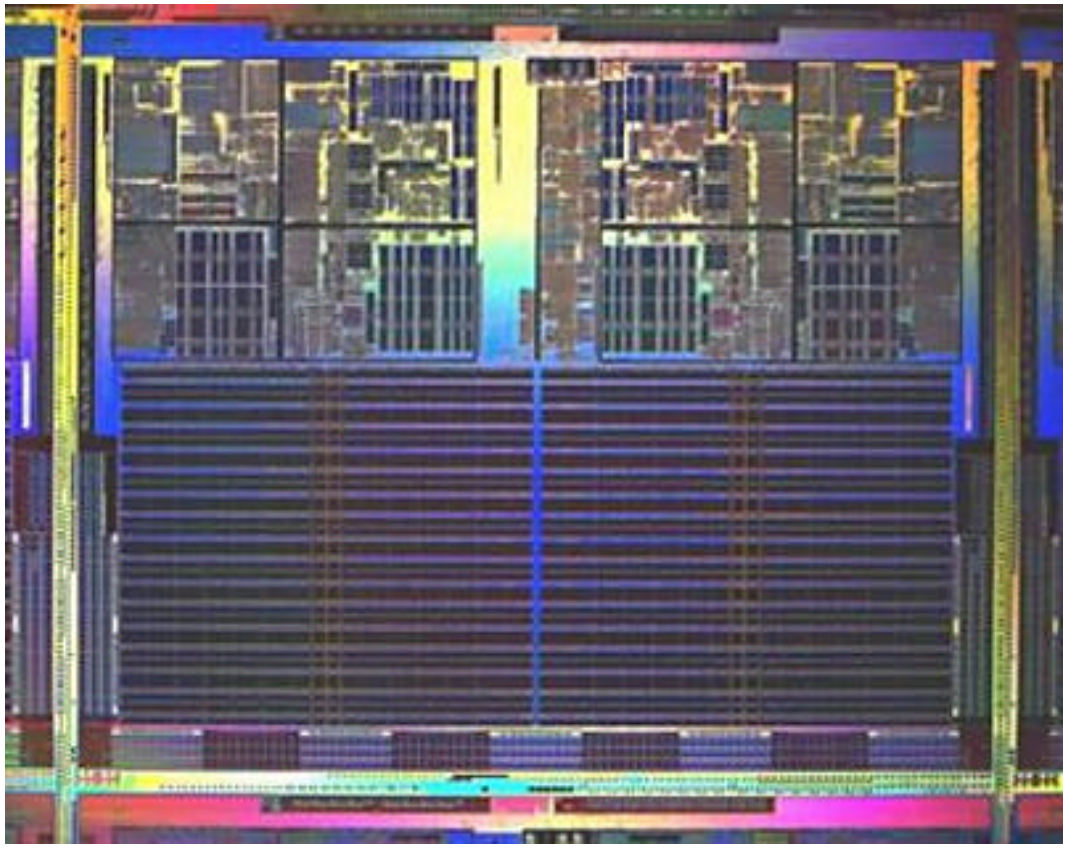
- **If a speculatively executed instruction incurs a big performance penalty (i.e., a cache-miss memory read), then it can actually slow down the system.**
- **If a speculatively executed instruction that should not have been executed causes an exception (i.e., divide-by-zero), then it's difficult to get everything back under control.**

Many systems that support speculative execution have special instructions like "speculative memory read" which the compiler can use to preload data it thinks may be needed. These types of instructions are used to get an early start on reading data from memory, but if the memory address is invalid (because the instruction shouldn't have been executed) then the instruction simply doesn't do anything.

End of the line for Instruction-Level Parallelism?

Most of these techniques have been aimed at improving the performance of a single stream of instructions, but they require more and more transistors for smaller and smaller performance gains. It looks like the CPU manufacturers have run out of tricks for improving the single instruction stream performance and are turning instead to multi-core architectures.

A multi-core CPU is one that has two or more complete CPU units on the same piece of silicon. Each CPU contains the registers, processing units, branch prediction, scorecard, etc. etc. Depending on the design, the chip may contain one combined cache for both CPUs, or it may contain a dedicated cache for each CPU.



The picture at right shows the dual-core AMD Opteron CPU chip. The two complete CPUs and their caches are easily visible.

A chip with two CPUs has the potential to double the performance of an equivalent chip with only one CPU, but in practice this is very dependent on the design of the cache and the bus interconnects on the chip to ensure that the two streams of instructions don't interfere with each other.

Multi-Threading

A multi-core CPU chip will not improve the performance of an ordinary program running a single instruction stream. Multi-core chips only benefit when there are two or more independent programs running at the same time, or when running a single program which has been written to have multiple threads.

In a multi-threaded program, you use a special routine to call one of your subroutines. Normally the subroutine would only return when it finishes executing, but the special routine creates a new "thread" and your main program then continues to execute at the same time as the subroutine. Here's an example program:

```
#include "stdio.h"
#include "process.h"
#include "windows.h"

static void Thread(void * pArgList)
{
        int i;

        for( i=0;  i<10;  i++ )
        {
                printf( "Thread - %d\n", i );
                Sleep( 500 );
        }
}
```

```
main (void)
{
        int          i;

        _beginthread( Thread, 0, NULL );

        for( i=0;  i<10; i++ )
        {
                printf( "Main program - %d\n", i );
                Sleep( 500 );
        }
}
```

Synchronizing Multiple Threads

You must take great care when writing a multithreaded program, because two or more instruction streams are using the same shared memory and can access the same data at the same time. This can lead to unexpected results which are very hard to recreate and debug.

The key requirement for multithreaded programs is the ability for the various threads to synchronize their operations when accessing shared data. Operating systems such as Windows provide library routines that make this possible. Some of the capabilities of these libraries include:

- **Creating "flag" or "mutex" data objects which can be set and tested in a way that is thread-safe. One thread can use such a data object to signal another thread that it is using shared data or that it has finished performing some task.**
- **Providing a way to wait for one or more of these data objects to be signaled.**
- **Providing a way to define a "critical section" of instructions is guaranteed to be executed all at once without any instructions from another thread executing at the same time.**

A good understanding of synchronization issues is essential when writing multithreaded code.

Sample Microarchitectures

We don't have time here to go into the details of the sample microarchitectures, but here is a summary of the differences:

8051

This is a very simple microarchitecture which has a degree of sophistication that is very similar to the MIC microarchitectures in the textbook.

UltraSPARC

The UltraSPARC is a classic RISC microarchitecture which needs almost nothing to decode the instructions. Most of the processing steps are the ones we've discussed, such as execution units, branch prediction, superscalar execution, and so on.

Pentium

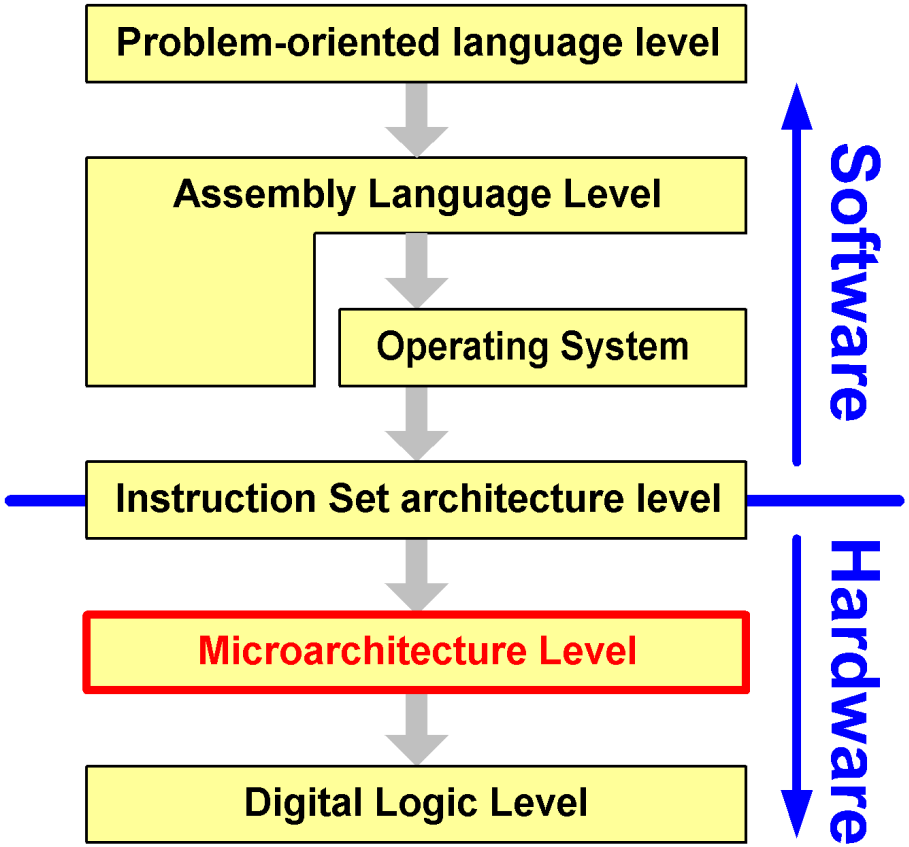
The Pentium microarchitecture has a major job to do that the UltraSPARC does not – convert the very irregular IA-32 instruction stream into a series of easily executed microinstructions. So while the Pentium has a "back end" that is very similar to the UltraSPARC, it also has a very complex "front end" whose job is to do the IA-32 instruction decoding.

In particular, the Pentium-4 used what Intel called the "NetBurst" microarchitecture which had very a deep pipeline (almost 2 dozen stages). Some have suggested this was a marketing decision to make CPUs with artificially high clock speeds.

Pages 312 thru 326 of the textbook have more information on the microarchitectures of these systems.

The Microarchitecture Level

This completes our exploration of the microarchitecture level. This level and the levels below it, are about the "how" of a computer system. They are the implementation of a computer system. Strictly speaking, computers are a "black box" and a programmer does not need to know anything about the implementation in order to use one. But as we've seen, the hardware architecture can have a large impact on performance, and an understanding of it can help you to write better programs.



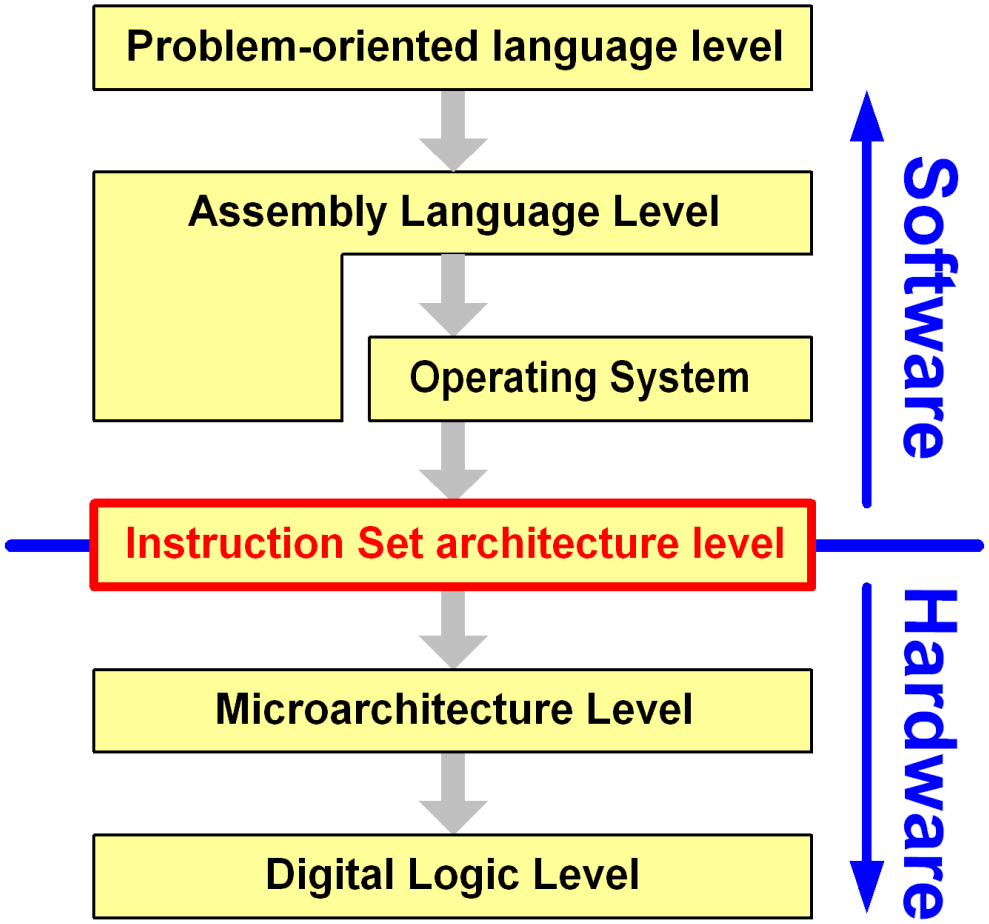
The Instruction Set Architecture Level

We'll now start our study of the Instruction Set Architecture (ISA) level of computers. This level is especially significant because it is the boundary between hardware and software.

When you buy a computer chip, it is designed to execute programs written at the ISA level. It's not possible to change the operation of the chip without replacing the chip itself (as Intel realized when they were forced to recall millions of Pentium chips with a faulty floating-point unit).

But software is loaded into memory each time it's run (typically from magnetic disk), and it can be changed simply by replacing the files that it's loaded from.

ISA-level programs can be produced by compilers / interpreters or by humans writing programs using Assembly Language. In either case, the program must conform to the specifications of the ISA for the computer that it will run on.



In this section of the course we'll describe what makes up the ISA level of the machine and show examples of the ISA level for several systems.

The Instruction Set Architecture Level

Here are some important points about the ISA level:

- **It describes what the computer hardware is supposed to do (the specification), not how this is accomplished (the implementation). It's possible to build several computers with the same ISA level, each of which accomplishes it's results using quite different techniques. Examples: Intel, AMD and Transmeta implementations of the X86 architecture, or different versions of the MIC microarchitecture that can all execute the same JVM instructions.**
- **ISA levels for different computers from different chip manufacturers (Intel, Sun, IBM) are usually different. This means that an ISA-level program that runs on one line of chips (UltraSPARC, for example) will not run on a different line (Pentium). Programs can be run on different ISA levels only if:**
  - **The program is written in a high level language such as C or Basic and is recompiled to produce an ISA translation for the machine you want to run it on, or**
  - **The program is written in a language that can be interpreted (ie, Java) and an interpreter exists for the ISA environment you want to run it on.**
- **Once a computer system with an ISA level has started to be accepted in the marketplace, backward compatibility becomes the overriding concern when introducing new generations of the computer system. Backward compatibility means that new features can be added to the ISA level, but all of the previous features must be retained.**
- **Some architectures (Java, UltraSPARC) have formal definitions of the ISA level because the manufacturers want to encourage people to build compatible systems, others (Pentium) do not (for exactly the opposite reason).**

Properties of the ISA Level

These are the main properties that comprise the ISA level of a particular computer system. Each of these will be described in more detail later:

- **Memory model – how memory is addressed and accessed by the system**
- **Registers – the number and type of registers and how they can be used.**
- **Data Types – what native data formats are supported by ISA instructions**
- **Instructions – what instructions the machine can execute, and what the effects are for each**
- **Interrupts and exceptions – how the machine controls and handles interrupts and exceptions**
- **Operating modes – which operating modes are available and the characteristics of each**

The documentation for a particular microprocessor may define some of the following characteristics for the system which are not part of the ISA level of the machine:

- **Performance – how long it takes to execute each instruction**
- **Implementation-specific features – such as the result returned by the CPUID instruction**

Properties of the ISA Level

Memory Model

The memory model describes how the CPU accesses memory. It includes the following characteristics:

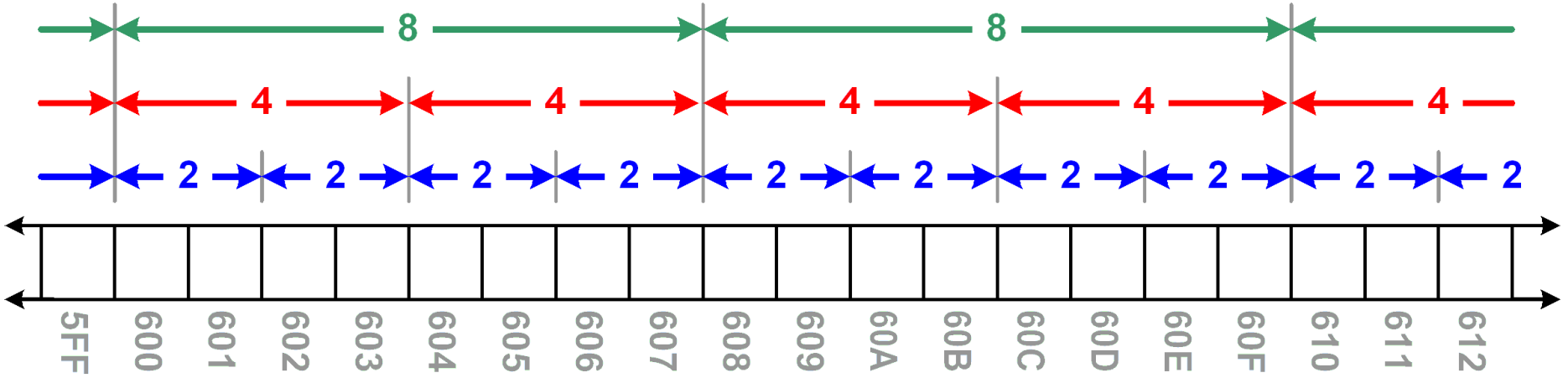
- **Word size (how many bits are stored at each address). 8 bits (1 byte) per address is common, but others are possible (as in the 32-bit word oriented data area of the JVM memory). Memories that store one byte per address are said to be byte-addressable.**
- **Memory capacity (how many addresses are supported by the architecture - the theoretical maximum, not the actual installed memory). It's quite common for an architecture to support a certain theoretical maximum, and various implementations to support a smaller amount of memory. For example the UltraSPARC architecture supports a memory space of 2^{64} addresses, but current chips only allow up to about 2^{48} bytes of actual memory to be physically connected. As technology improves newer chips can support larger memories and still be compatible with the original architecture.**
- **Memory alignment (whether data items that span multiple addresses must be aligned on particular address boundaries – see below).**
- **Big- vs. Little-Endian byte ordering (this was discussed earlier). Many ISAs are switchable between the two modes.**
- **Memory semantics for reads and writes. In most systems data written to memory can be immediately read back, but some systems require a specific memory synchronization instruction to be used to guarantee that all previous writes have completed.**
- **How the address space is used (see below)**
- **Virtual memory support (beyond the scope of this course)**

Properties of the ISA Level

Memory Model – Alignment

When a data spans more than one address (for example, a 32-bit word stored in a byte-addressable memory), some ISAs require the data to be aligned on a natural boundary. Aligned data always starts at an address that is a multiple of the word size. For example, a 32-bit number (which is 4 bytes long) would have to start on an address that is a multiple of 4.

This diagram shows the proper alignment for 2-, 4- and 8-byte words:

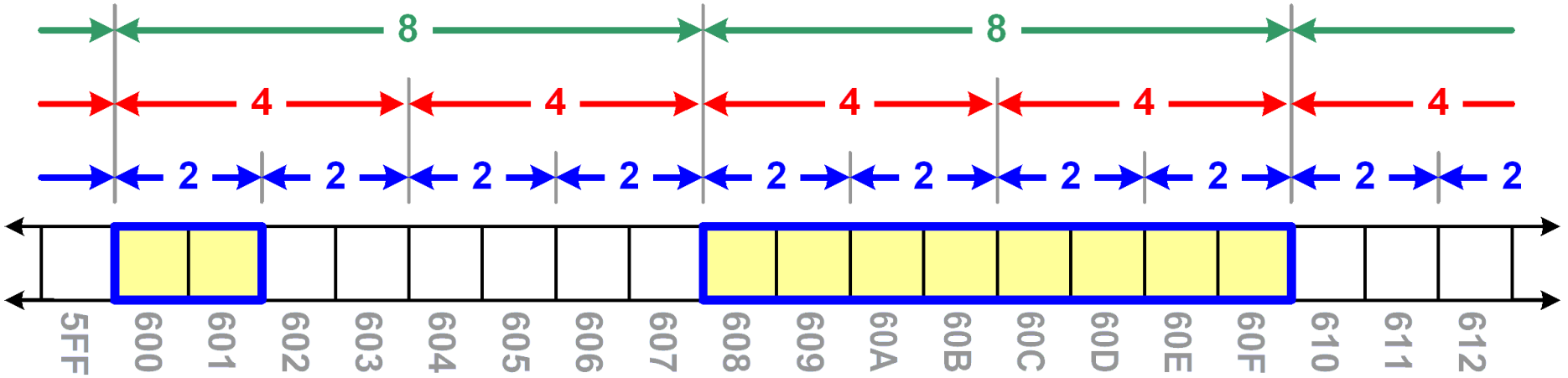


The reason that memory alignment is important is that modern CPUs access multi-byte words in memory. For example, imagine that the program tries to access a 4-byte word stored at addresses 606, 607, 608 and 609 in the diagram above. In order to fulfill this request, the CPU must actually read two words (one at addresses 604-607, and the second one at 608-60B) and then pick out the required bytes from them. If the ISA requires the words to be aligned, then the designers don't have to worry about adding the extra logic to do this, and the system performs faster.

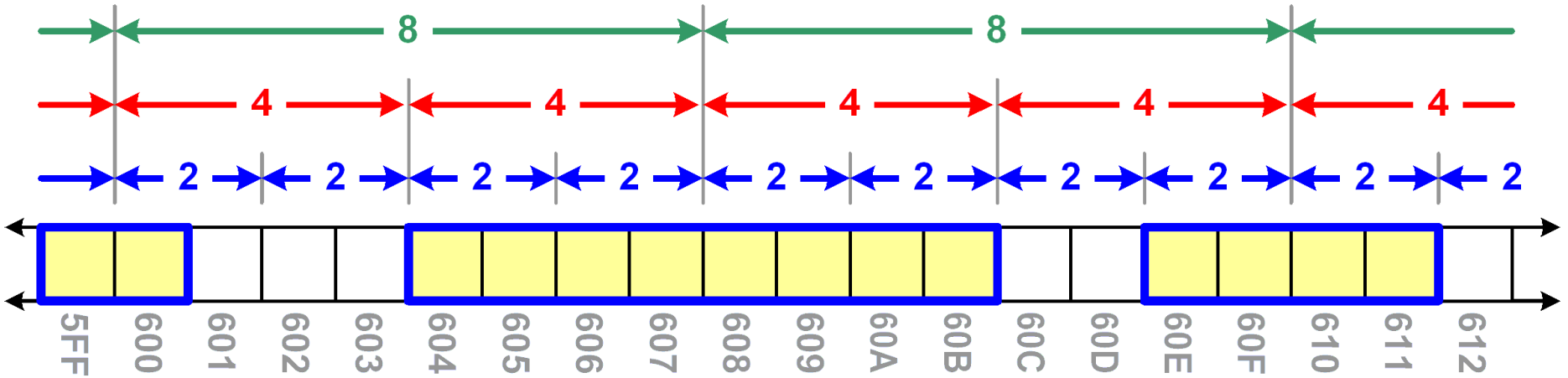
Properties of the ISA Level

Memory Model – Alignment

Here are some examples of memory words that are properly aligned:



And here are some that are improperly aligned:



Properties of the ISA Level

Memory Alignment and Data Declarations

When you define data in your program you should always try to do so in a way that keeps the data aligned on it's proper boundaries so as to optimize performance. Here's an example of the wrong and right way to define data in a structure:

BAD:

```
struct
{                           // Stored at:
    char    cAge;           //   byte 0
    short   sAcct;          //   bytes 1-2
    long    lBalance;       //   bytes 3-6
} myStruct;
```

GOOD:

```
struct
{                           // Stored at:
    long    lBalance;       //   bytes 0-3
    short   sAcct;          //   bytes 4-5
    char    cAge;           //   byte 6
} myStruct;
```

In the declaration on the left, the short (2-byte) and long (4-byte) variables are <u>not</u> stored at addresses that are a multiple of their size. By changing the order of the declaration, you can arrange the data so that each item is properly aligned. The general rule for doing this is:

Declare the data that takes the most storage first

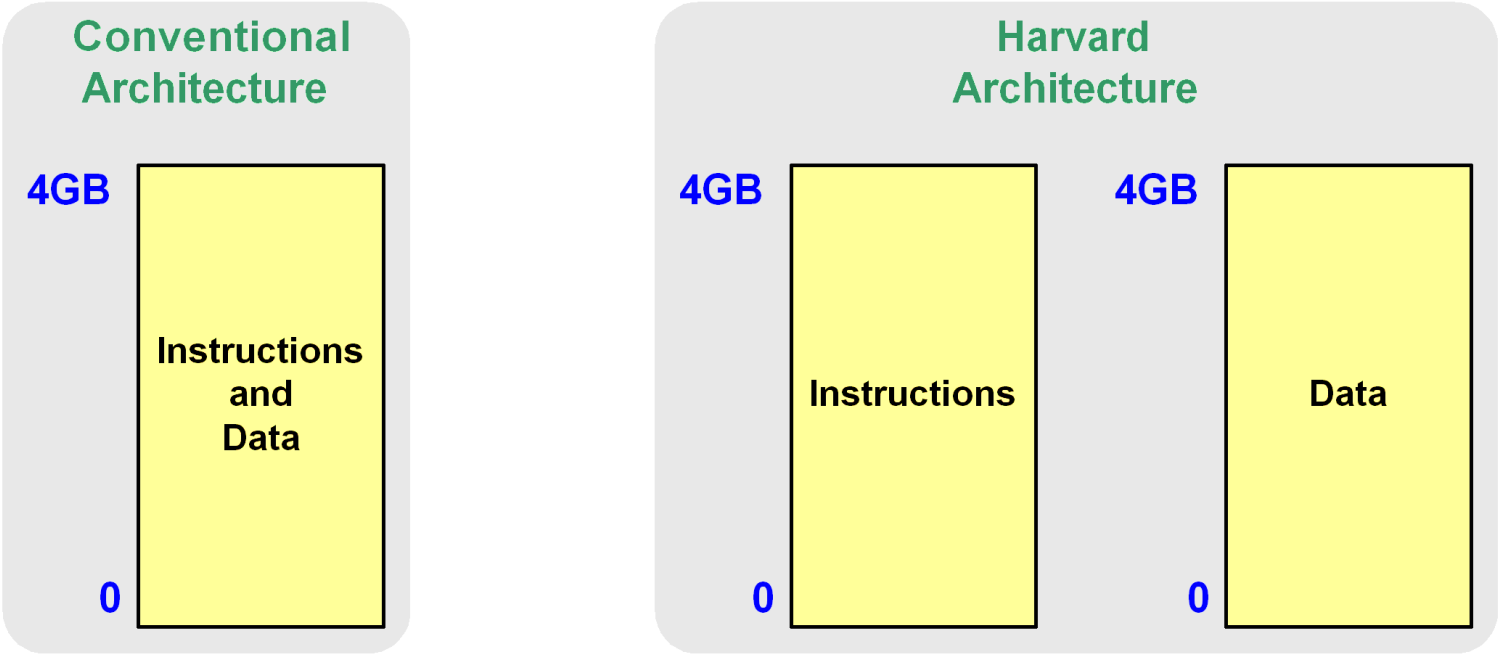
This is most important in structure definitions, since the compiler has no flexibility to rearrange the data, but it's a good idea to be in the habit of doing this for all data declarations.

But remember that if you're declaring a structure that has to match an existing data layout, you can't arbitrarily change it's order!

Properties of the ISA Level

Memory Model – Address Space

Most computers have a single address space which stores both instructions and data, but it is possible to separate these into two separate memories, each with their own set of addresses. Such a memory organization is called the "Harvard Architecture":



The Java Virtual Machine is an example of an ISA that uses the Harvard Architecture.

Harvard Architecture machines have twice as much memory capacity for a given address size, and they also have twice the memory bandwidth – but they suffer from a "partitioning" problem (instructions can't take advantages of unused data memory if all the instruction addresses are used up). Most modern systems compromise by using a split cache and a unified memory – that's an implementation feature and not an architectural feature.

Properties of the ISA Level

Memory Model – Segmented vs. Flat Address Space

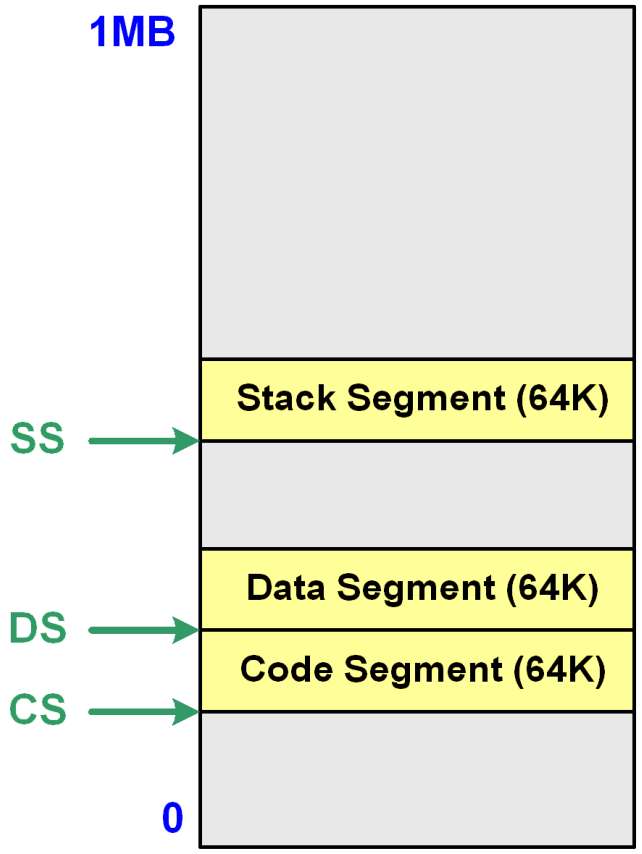
Most computers have a simple "flat" address space with addresses that range from 0 to the maximum address. But it's also possible to use a segmented address space.

The diagram at right shows an example of the segmented address model used in the original Intel 8088. The 8088 could support 1MB of memory, but the registers were only 16 bits long, so the largest memory address they could store was 65535.

To get around this limitation, the 8088 had a several segment registers which pointed to the base of memory areas that held different parts of the program. The most common registers were CS (Code Segment), DS (Data Segment) and SS (Stack Segment).

When the 8088 fetched instructions, it added the value of the IP (Instruction Pointer, similar to PC) register to the base address in the CS register to computer the actual memory address.

The result was that the program had access to a set of 64K "windows" into the 1MB of physical memory.



Modern Pentium machines can still use this same segmented addressing mode, but because they have 32-bit registers it's simpler to use flat addressing.

Properties of the ISA Level

Operating Modes

Modern CPUs can typically execute in at least two different modes of operation. These different modes are sometimes called privilege levels. The names of these levels may vary from system to system, but common names are:

User Mode

Typical application programs operate in user mode. In this mode the CPU makes a basic set of ISA features available for the program to use.

Kernel Mode

The operating system normally runs in kernel mode. In this mode the CPU makes the full set of ISA features available. This typically includes:

- **Ability to perform input / output instructions**
- **Ability to set memory protections**
- **Ability to control the interrupt and exception handling features of the system**

Some systems have additional modes – for example the Pentium II has four different privilege levels.

Having two or more modes is essential if the operating system is to protect the computer from buggy or malicious programs. It allows the operating system to configure memory and I/O devices so that one program cannot affect the memory or files used by another program.

Properties of the ISA Level

Registers

As we've seen, registers are storage areas within the CPU used to hold the data that's being operated on. Data must be moved from memory to the registers before it can be manipulated.

Virtually all CPUs have a set of special-purpose registers that have a specific defined function at the ISA level. These typically include:

The Program Counter (PC) Register

This register contains the memory address of the next ISA instruction to be executed. It's known as the IP (Instruction Pointer) register in IA-32 (Pentium & compatible) CPUs.

The Stack Pointer (SP) Register

This register contains the memory address of the last word pushed onto the stack.

The Frame Pointer (FP) Register

This register contains the memory address of the area that holds the local variables for the current routine (this is typically an address on the stack). It's known as the BP (Base Pointer) register in IA-32 (Pentium & compatible) CPUs, and the LV (Local Variable Pointer) in the Java Virtual Machine.

The Flags or Status Register

This register contains a series of bits that show the state of the last arithmetic or Boolean instruction as described on the next page.

When operating in kernel mode there are usually other special registers that are also available to control the state of the machine.

Properties of the ISA Level

Registers – The Flags Register

The Flags register contains bits which indicate the status of the last arithmetic or Boolean instruction. Virtually all CPUs include the following status bits in this register:

- **N (Negative) – set to TRUE when the last result was less than zero**
- **Z (Zero) – set to TRUE when the last result was zero**
- **V (oVerflow) – set to TRUE when the last result had a signed overflow**
- **C (Carry) – set to TRUE when the last result had an unsigned overflow**

There are often other status bits as well.

This register usually contains other bits that the program can set to control the operation of the CPU. These bits often include:

- **Enable Interrupts – set to TRUE to enable interrupts**
- **Interrupt Priority Level – Indicates what interrupt levels can interrupt the machine**
- **Trace – set to TRUE to enable debugging features**

The specific bits and their positions are usually quite different from one CPU family to another.

The flags register bits which control the state of the machine are usually read-only unless the system is operating in Kernel mode.

Properties of the ISA Level

Registers – Accumulator vs. General-Purpose Registers

Early computers had a single register called the "Accumulator" which was used as the input and output for all program instructions. In , for example, a program to calculate A = B + C might use ISA instructions like this:

LOAD	**B**	**Move B from memory to accumulator**
ADD	**C**	**Add C from memory into accumulator**
STORE	**A**	**Move result from accumulator to A in memory**

A single register forces lots of memory accesses, and since memory is much slower than modern CPUs, most current systems have anywhere from several to dozens of registers. Registers that are not used for a particular purpose (as described above) are known as general-purpose registers.

The names of general-purpose registers vary from machine to machine. For example, many systems use register numbers such as "R0", "R1", "R2", and so on. The Intel Pentium and compatible systems use names starting with AX, BX, CX, etc., with variants such as EAX, AX, or AL to refer to 32-bit, 16-bit, or 8-bit portions of the register.

Although general-purpose registers can usually be used interchangeably, some architectures reserve specific registers for certain functions. For example, the AX register in a Pentium is always used as the operand in integer Multiply and Divide instructions.

Even in systems whose registers are completely interchangeable, the operating system and languages often have a convention for register usage so that different programs use the same registers to pass parameters, return results, and so on.

Many systems have a third set of floating-point registers to handle floating point arithmetic.

Exercise 8 – Registers

Which registers are used in a typical Instruction Set Architecture for the following purposes:

1. **Holds information about the last arithmetic or logical operation**
2. **Contains the address of the next instruction to be executed**
3. **Contains the address of the local variables for the current procedure**

A – Stack Pointer register

B – Frame Pointer, BP or LV register

C – Status or Flags register

D – Microinstruction register

E – Program Counter or IP register

Sample ISA Levels – the Intel Pentium 4

The Instruction Set Architecture used by Intel chips since the 80386 is known as the "IA-32" architecture ("Intel Architecture 32-bit"). Essentially the same ISA, was used in earlier Pentium chips, as well as in clone CPUs such as the AMD Athlon.

The IA-32 architecture is upward compatible from the original 8088 chip used in the original IBM PC, because it includes the subset 16-bit instruction used in that chip. The biggest architectural changes to the IA-32 architecture have been:

- **The addition of vector arithmetic instruction sets: MMX, SSE and SSE2**
- **The addition of special addressing modes to extend beyond 4GB of memory**

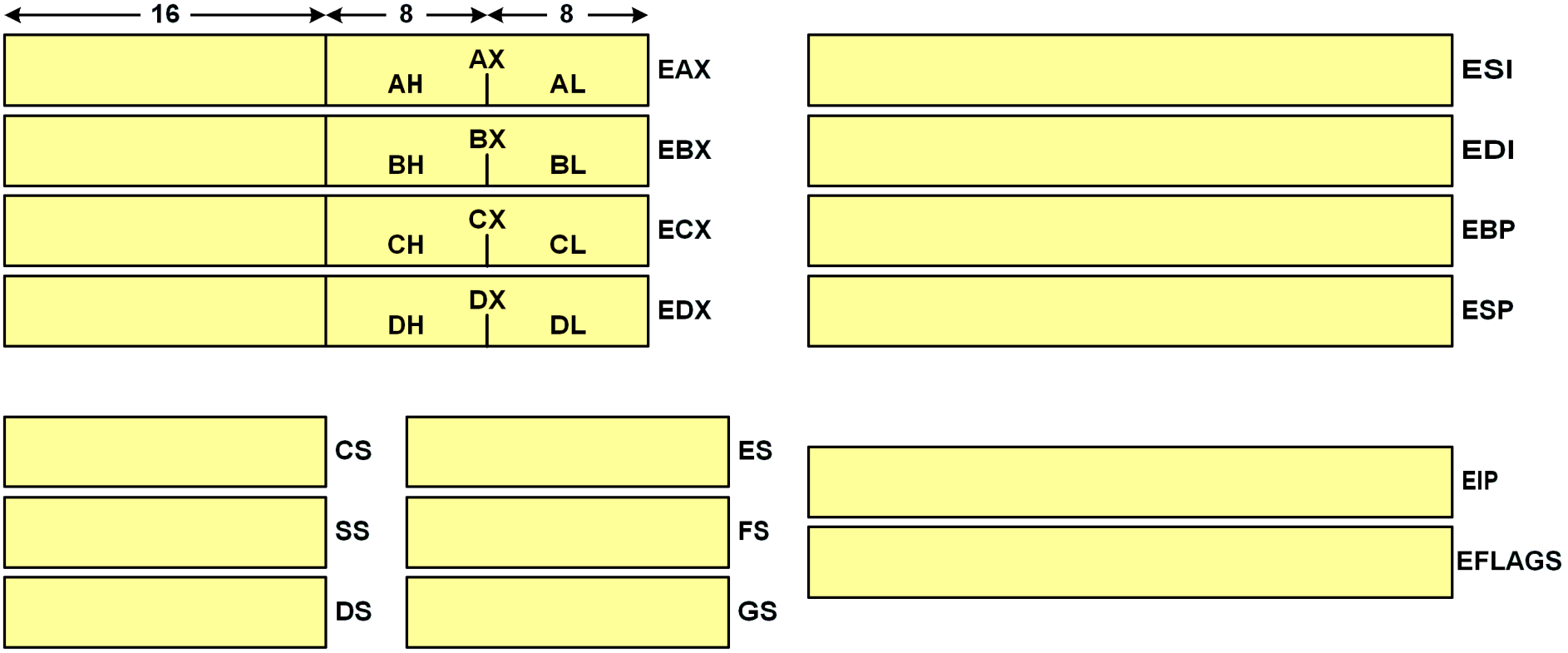
A Pentium system can operate in one of three modes:

- **Real Mode – in this mode the chip operates exactly like a 16-bit 8088 chip with all of the 32-bit features turned off. When a Pentium chip is first turned on it comes up in this mode, and it would be capable of running Microsoft DOS V1.0 (if you could find it on media that can be read by a modern system)**
- **Protected Mode – in this mode the 32-bit features are turned on, including virtual memory and security features such as user/kernel mode switching. Modern operating systems switch the processor into this mode during their startup sequence.**
- **Virtual 8086 Mode – the operating system can put the chip into this mode when running a DOS application program to more accurately simulate a 16-bit environment. This mode is not very important today since few people run 16-bit programs.**

Sample ISA Levels – the Intel Pentium 4

Registers

The following diagram shows the registers that are available in ISA level for the Pentium 4:



EAX, EBX, ECX and EDX are general-purpose registers

ESI, EDI, EBP, ESP, EIP and EFLAGS are special-purpose registers

CS, SS, DS, ES, FS, and GS are segment registers

Sample ISA Levels – the Intel Pentium 4

Registers

The original 8088 used 16-bit registers, but in the Pentium these have been extended to 32-bits and the letter "E" (for "extended") has been added to the front of all the register names.

The general-purpose registers (EAX, EBX, ECX, EDX) contain 32 bits each, but the program can access the low-order 16 bits as AX, BX, CX, and DX. The program can also access the low order 2 bytes as AL or AH, BL or BH, CL or CH, and DL or DH. Although these are general-purpose registers, many instructions require specific registers. For example, multiply and divide always use the EAX and EDX registers, while loops use the CX register.

The special-purpose registers are:

- **ESI and EDI – these are the "Source Index" and "Destination Index" registers which are used as memory pointers for string and other instructions.**
- **EBP – this is the "Base Pointer" – it's equivalent to the "FP" (Frame Pointer) register which points to the local variables for the current procedure.**
- **ESP – this is the "Stack Pointer" register that points to the top word on the stack.**
- **EIP – this is the "Instruction Pointer" – it's equivalent to the "PC" (Program Counter) register which points to the next instruction to be executed.**
- **EFlags – this is the flags register which holds status flags from the last operation.**

The Pentium also has a separate set of floating-point registers (not shown).

Sample ISA Levels – the Intel Pentium 4

Memory

The Pentium supports a 4GB address space using 32-bit addresses. Words are stored in Little-Endian format and do not need to be aligned on natural boundaries.

Addressing is done through the use of segments which are controlled by the segment registers:

- **CS – Code Segment.**
- **DS – Data Segment**
- **SS – Stack Segment**
- **ES, FS, GS – extra segments**

Each segment register points to a segment table which controls segment's starting address and length in memory. This scheme is a throwback to the original segmented architecture used in the 8088 CPU.

Segmented addressing is awkward to use, so modern operating systems us a "flat" addressing model by setting all of the segments to start at address zero and have a length of 4GB. This way any segment can be used to access any part of memory, and the segmented nature of the system is effectively bypassed.

Sample ISA Levels – the UltraSPARC III

The UltraSPARC is one of the first commercial RISC architectures, and it's based very heavily on the research done at Berkely in the 1980's. Like the IA-32 architecture, UltraSPARC has evolved over time. The original UltraSPARC was a 32-bit architecture, but with the UltraSPARC II the architecture was been extended to 64 bits. It was very easy to make these improvements since the 64-bit extensions were anticipated when the original architecture was designed.

As in most RISC designs, the UltraSPARC III uses a Load/Store architecture, so most instructions can only operate on data stored in the registers. Therefore the UltraSPARC III provides 32 registers that a routine can use:

Register	Convention	Function
R0	G0	Hardwired to 0. Stores into it are ignored
R1 – R7	G1 – G7	Holds global variables
R8 – R13	O0 – R5	Parameters being passed to a called procedure
R14	SP	Stack Pointer
R15	O7	Scratch register for temporary use
R16 – R23	L0 – L7	Holds local variables for the current procedure
R24 – R29	I0 – I5	Holds parameters passed from the calling procedure
R30	FP	Frame Pointer to local variables stored in memory
R31	I7	Holds return address for the current procedure

These are the registers that any procedure can use. (There is also a separate set of 32 floating-point registers). But the UltraSPARC architecture is designed so that each procedure can have it's own set of registers, and there are in fact many more registers than this.

Sample ISA Levels – the UltraSPARC III

Register Windows

The UltraSPARC architecture has two sets of registers:

Global Registers

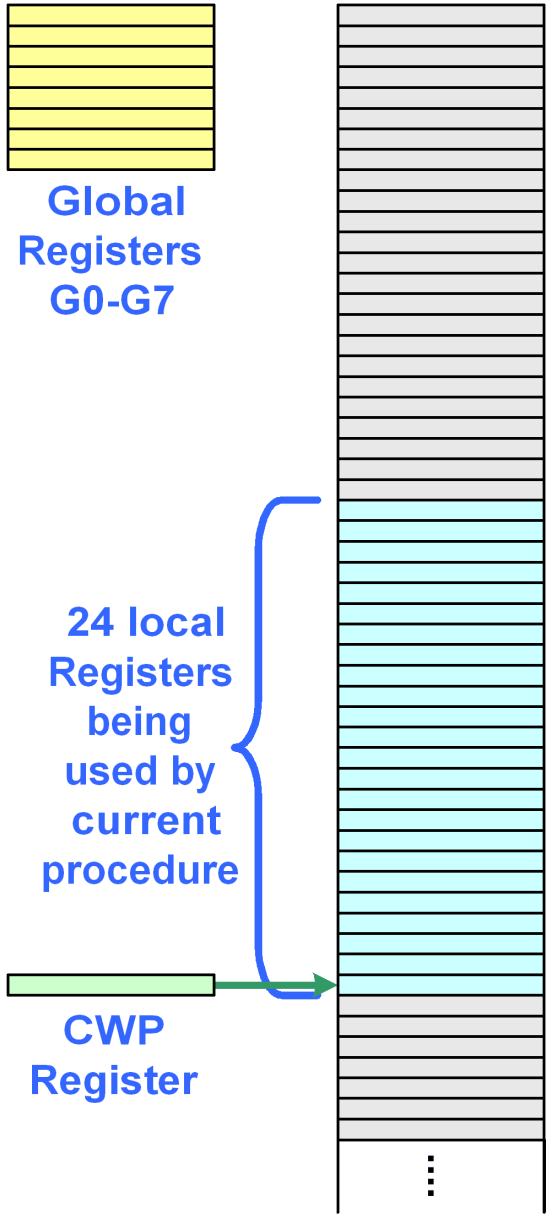
There are eight global registers called G0 through G7 which are shared by all programs.

Local Registers.

There are 24 registers that are different for each procedure. They are allocated from a much larger set of registers. A special control register called CWP (Current Window Pointer) identifies which 24 registers of the register bank are being used by the current procedure.

Each time a procedure call instruction is issued, the CWP register is changed to that a new set of local registers becomes active. When the called procedure issues a return instruction, the CWP register is reset back to it's original value.

Procedure calls and returns don't activate a completely different set of 24 registers – there is overlap between the calling procedure's registers and the called procedure's registers – this allows the calling procedure to pass data in the registers to the called procedure as shown on the next page.



Sample ISA Levels – the UltraSPARC III

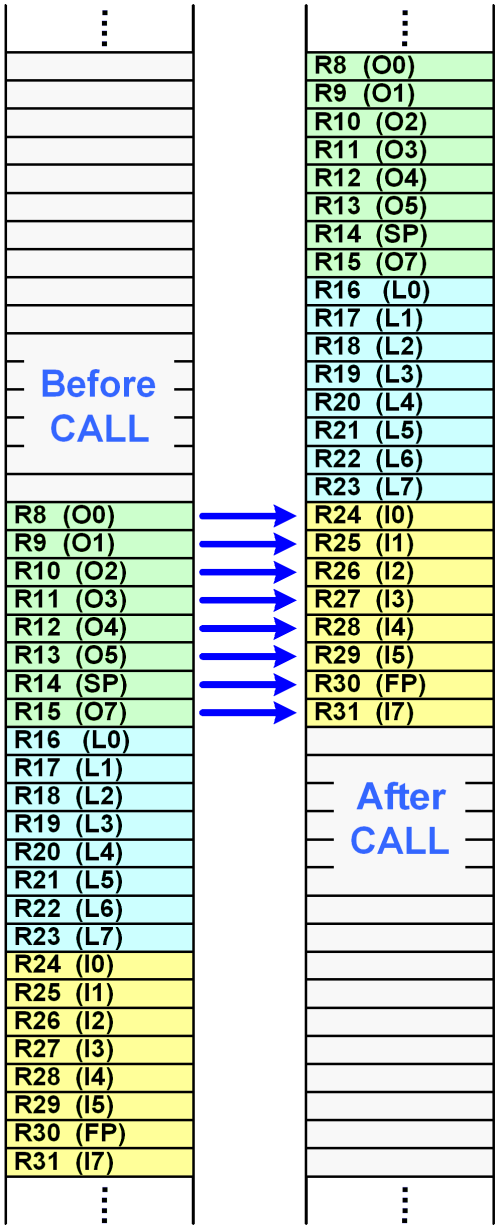
Register Windows

Then a subroutine call is issued, the CWP register is changed and a new set of local registers becomes active. But there are 8 registers that overlap between the calling and called procedures. These are the Input and Output parameter registers.

To pass information to a subroutine, the calling procedure simply puts information into the output registers (R8 – R15) and called the subroutine. The same information appears in the called procedure's input registers (R24 – R31). Thus, the UltraSPARC eliminates the need for routines to pass information through memory.

The total number of UltraSPARC registers is finite, and if calls are too deeply nested then some information must be written to memory to make room for new registers. In many ways this bank of registers is like the 64 "top of stack" registers in the picoJava II CPU.

The idea of using register windows was an early attempt in RISC architectures to eliminate the need for memory accesses, but it introduces a lot of complexity to the architecture. Whether the benefits it brings are worth the trouble is a matter for debate.



Sample ISA Levels – the UltraSPARC III

Memory

The UltraSPARC II is a 64-bit architecture, and it supports 64-bit addressing. This provides for potentially 2^{64}, or 18,446,744,073,709,551,616 bytes (18 Exabytes) of memory – over a billion times more memory than most current systems.

(An Exabyte is 1000 Petabytes, a Petabyte is 1000 Terabytes, and a Terabyte is 1000 Gigabytes)

The CPU can access data stored in memory in Little-Endian or Big-Endian format depending on the setting of certain control registers. Data must be properly aligned on natural boundaries.

In practice it's not possible to connect the theoretical maximum amount of memory to an UltraSPARC system, and the actual implementation only supports 2^{44} bytes (about 16 terabytes) of memory. However future chips will be able to support more memory without any changes to the Instruction Set Architecture.

In the past, running out of addressing space has been a recurring problem with computer architectures. The computer industry ran into this problem in the 1980's when the 64KByte memory limitations of 16-bit computers became a problem. The same thing is happening with high-end 32-bit computers today because of their 4GB addressing limitation.

The leap to 64-bit computing is a substantial one – it will certainly be at least a few decades before 18 Exabytes of memory becomes an issue.

Sample ISA Levels – the 8051

The 8051 is a very old design which was intended for use as an embedded processor with most of the hardware required to build a complete working system included as part of the CPU chip itself. Because of this, it has a much more functionality than more recent processors. Despite this, the 8051 is still widely used in embedded applications.

Some of the features of the 8051 include:

- **On-chip RAM memory for data and ROM memory to hold programs**
- **Different versions of the chip are available with different memory sizes**
- **Switchable banks of registers for fast handling of interrupts**
- **Built-in timers and I/O ports**
- **Special memory addresses to streamline handling of individual bits**

These features support the design goal of delivering a cheap but effective processor for embedded applications.

Sample ISA Levels – the 8051

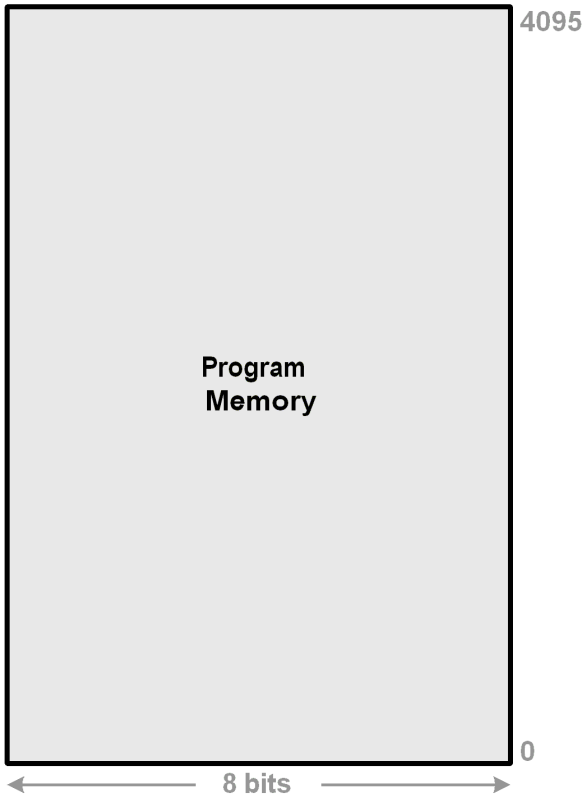
Memory

The 8051 uses the Harvard Architecture – it has two totally separate address spaces for instructions and data as shown in the diagram at right.

Program instructions are held in ROM which stores 8 bits per cell and has 4096 cells. (It's also possible to connect an external chip to provide more program storage).

Data is held in a separate area with 8 bits per cell and a total of 128 cells. The data memory has several unique features:

- **The first 32 bytes of the data memory hold four banks of register values. Unlike most other systems, the 8051 uses the data memory to hold register values. This isn't as big a problem as it is in other architectures because in the 8051 the data memory is on the CPU chip and is just as fast as the CPU is.**
- **Another 16 bytes of data memory is reserved for use with special bit-oriented instructions. These instructions can AND, OR, or test each of the 128 bits of these 16 bytes as if they were individual bits by specifying a bit number from 0 through 127. This makes the 8051 very efficient at handling bit values such as "button is pressed" or "turn light on".**

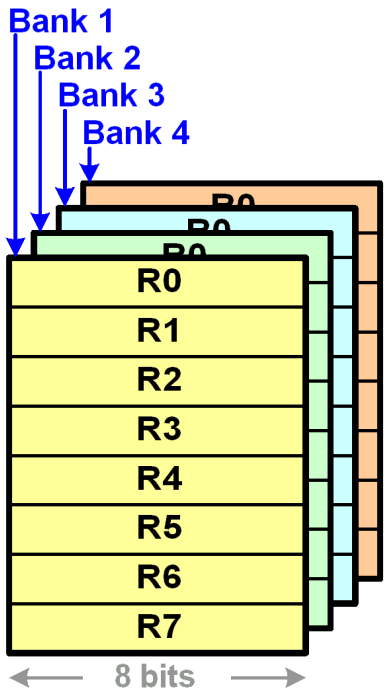


Sample ISA Levels – the 8051

General Registers

The 8051 has eight general-purpose 8-bit registers whose storage is held in memory. These registers can be accessed directly by many program instructions which contain a 3-bit register number.

Like the UltraSPARC, the 8051 actually has multiple copies of each of the registers (but of course the 8051 was using this idea long before the UltraSPARC was around). There are four complete "banks" of registers, and any bank can be switched to become "active" by changing a 2-bit field in the Program Status Word.



Having multiple register banks makes it possible for the 8051 to respond to interrupts very quickly, because it doesn't have to save the current register values when the interrupt handler starts running and then restore them when it exits. Being able to respond quickly to interrupts is an important capability for a system which can be used in real-time applications.

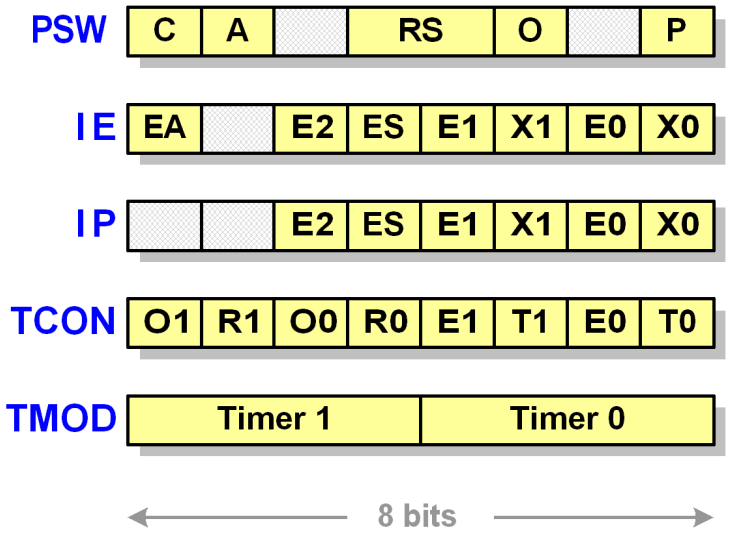
Another interesting feature of the 8051 is that the register values are actually held in the data memory. For example, if the program puts a number into R0 of bank 1, the same number can be read back out of memory at data address 0.

In addition to the general registers, the 8051 also has a special "Accumulator" register which is the target of many arithmetic instructions. For example, the instruction "ADD R5" tells the machine to add the contents of R5 to the accumulator registers.

Sample ISA Levels – the 8051

Special Registers

In addition to the general-purpose registers, the 8051 also has five special-purpose registers:



PSW (Program Status Word) contains the following bits which generally show the status of the last ALU result:

- **C – Carry bit (unsigned number is too large)**
- **A – Auxilliary Carry bit (carry from bit 3 → 4)**
- **RS – which register set is active**
- **O – Overflow bit (signed number is too large)**
- **P – Parity bit (set to the even parity of the result)**

IE (Interrupt Enable) register:

- **EA – Disables all interrupts if zero**
- **E0 / E1 / E2 – Disables interrupts from timers 0, 1, or 2 if zero**
- **ES – disables serial interrupts if zero**
- **X0 / X1 disables external interrupts if zero**

IP (Interrupt Priority) Sets the priority for various interrupts to "Low" or "High"

TCON (Timer Control) turns timers on or off. The "O0" and "O1" bits are turned on by the timers when their value overflows.

TMOD (Timer Mode) controls how high the timers count and whether they wrap around or not

Exercise 9 – Instruction Set Architecture

The ISA level of a computer system includes the following characteristics:

A – memory size, and organization, and access requirements

B – register sizes and purposes

C – operating modes (such as user or kernel)

D – the number of pipeline stages

E – clock speed

F – A, B and C above

G – all of the above

H – none of the above

Key Concepts

- Cache lines, tags and byte numbers
- Cache organizations: Fully associative, direct mapped, set associative
- How the cache uses an address to find data
- Cache replacement algorithms
- Branch prediction bits and finite state machines
- Superscalar execution and dependencies
- Using a scoreboard to detect dependencies between instructions
- Using register renaming to eliminate some types of dependencies
- Differences and similarities between sample microarchitectures
- Characteristics of the ISA level:
  - Memory model
  - Registers
  - Data Types
  - Instructions
  - Interrupt and exception handling
  - Operating modes

What's Next

- **Look at Review Questions for this week**
- **Sign on to WebCT and do Module 8 – "Intel IA64 Architecture"**
- **Study for Quiz 6, which includes:**
  - **Week 10 lecture**
  - **WebCT module 8**
- **Complete Assignment 2 for handing in next week**
- **Pre-read the material for Week 10**