Minnesanteckningar - Datateknik

Modul 1: C och Assembly

INPUTS		OUTPUTS					
A	B	AND	NAND	OR	NOR	EXOR	EXNOR
0	0	0	1	0	1	0	1
0	1	0	1	1	0	1	0
1	0	0	1	1	0	1	0
1	1	1	0	1	0	0	1

---

andi, **ori** och **xori** används när man lägger ihop register och heltal.

SLL = Shift left logical
SRL = Shift right logical
SRA = Shift right arithmetic. Shifts in the sign bit as the most significant bit. Dividing signed numbers.

LUI -- *Load upper immediate*

Description:	The immediate value is shifted left 16 bits and stored in the register. The lower 16 bits are zeroes.
Operation:	$t = (imm << 16); advance_pc (4);
Syntax:	lui $t, imm
Encoding:	`0011 11-- ---t tttt iiii iiii iiii iiii`

Pointers

Pointer ger adressen som är lagrad i variabeln pointer
*Pointer ger dig det som finns lagrat på adressen
& symbol is used in an expression for getting the memory address of a variable.

(5 << 1); // 0101 skiftas en bit åt vänster vilket blir 1010

Registerna $v0 och $v1 används som function return

Two’s complement

- Invert all bit of number X and add 1.

Ex. 3 in binary = 0011

Invert all bits

1100

Add 1

1101

A stack is a data structure where data items can be pushed and popped. The things that are pushed (added to the stack) will first be popped (removed from the stack) last. When a function call is performed, registers must be push because the called function may change these registers. Other “things” that can be stored on the stack are, for instance, argument values and local variables.

Modul 2 - I/O systems

Minns att t.ex. när lampor ska tändas så tänds detta genom att sätta specifika bitar till 1.
I detta fall börjar index på 0. Sekvensen 1011 tänder lampa 0, 1 och 3.
I assembly översätts detta till hex, dvs 0xb

För att konkatenera adresser kan man skriva lui på rad 1 och addi på rad 2.
addi kan användas på rad 2 men om immediate har MSB = 1 så tolkar addi detta som en subtraktion. Addi kör alltså sign extension på samtliga immediate värden.

---

volatile int* push_buttons = (**volatile int***) 0x8000abc0; // push_buttons pekar på adressen.

if((*push_buttons) & 8) // Om knapp 3 trycks ned ska samtliga lampor tändas. Därför kör man bitvis and med 8 (1000). Vilket gör att samtliga bitar blir 0 förutom den med index 3.

*LED = 0x3f; // eftersom att alla 6 lampor ska tändas då 3 är nere så skrivs 0x3F vilket binärt är 0011 1111

else *LED = 0x0; }

Tecknet ~ betyder bitvis invers

Genom att inkludera <pic32mx.h> kan man skriva namnet på registerna istället för adresserna.

---

volatile innebär:

När C-kompilatorn optimerar programmet så letar den efter kod som verkar läsa samma plats flera gånger i en loop eller liknande. Om kompilatorn finner sådan kod, gör den optimiseringar så att platsen endast läses en gång. Detta funkar för vanliga variabler men kommer förstöra kod som hanterar input/output. Volatile betyder att en plats kan ändras på ett sätt som kompilatorn inte kan se och stoppar optimiseringen.

On a PIC32 microprocessor, the timer TMR2 is controlled through the following 14 device-registers:

• T2CON, T2CONCLR, T2CONSET, T2CONINV • TMR2, TMR2CLR, TMR2SET, TMR2INV
• PR2, PR2CLR, PR2SET, PR2INV
• IEC0, IFS0

T1CON: Type A Timer Control Register

TxCON: Type B Timer Control Register

TMRx: Timer Register

PRx: Period Register

Direct Memory Access (DMA) enables a memory transfers to occur, without continuous intervention by the processor.

Prescaling

f / (periodregister * prescaling) = sekunder

1 sekund = 1 000 millisekunder

1ms = 0,001 s

Period registret beror på hur många bitar timern är.
En 16-bitars timer kan max anta PR av 65 535

Ett 8-bitars periodregister kan ha högst värdet 255

1, Ta reda på högsta värdet PR kan anta. Ex. 255
2, Bestäm vad timern ska räkna till. Ex 1000 för att få timer period ex 1 ms
3, Välj ett värde på PR som går jämnt ut med timer period. Ex 250 som PR och 1:4 →250/0,25 = 1000

I kontrollregistret så sätter man på timern och ställer in prescale och så

Period registret - Det är perioden, alltså hur lång tid det är mellan varje timeout

---

```
volatile int* control = (volatile int*) 0xffa0;

volatile int* period = (volatile int*) 0xffa2;

*control = (*control & ~0xc0) | (2 << 6);

*period = 250;
```

Tänd led 4 som finns på bit index 3

Adress 0xbf226000

led4:

```
lui     $t0, 0xbf22
ori     $t0, $t0, 0x6000 // load adress as usual

andi    $a0, $a0, 1      // makes sure that we only use bit with index 0.
Sll     $a0, $a0, 3      // shift the value to led 4 position

lw      $t1, 0($t0)      // load previous setting
andi    $t1, $t1, 0xfff7 // clear bit 3
or      $a0, $t1, $a0    // insert the bit for the led

sw      $a0, 0($t0)      // write to the memory mapped IO
jr      $ra              // return from function call
```

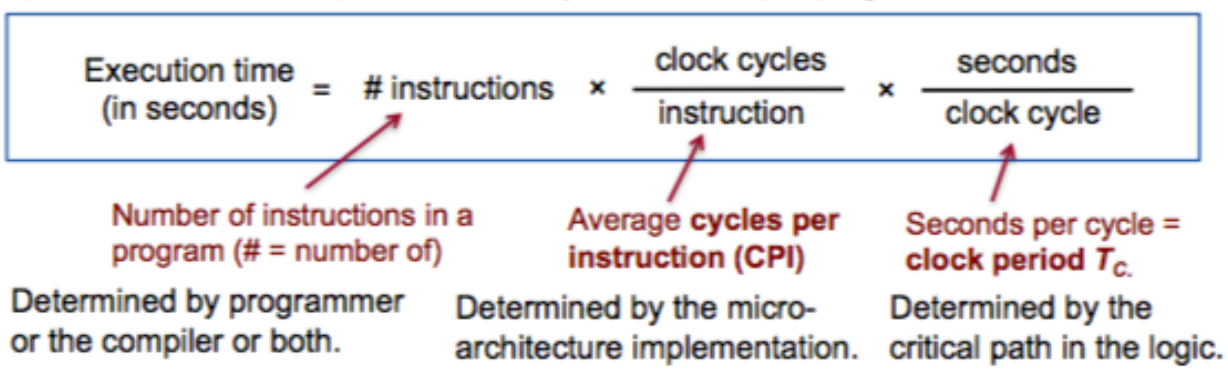
Modul 4 - Processor Design

MIPS Control Signal Summary

The single-cycle implementation of the MIPS processor uses the following control signals, which are grouped according to the execution activity that they affect.

Activity	Signal	Purpose
PC Update	Branch	Combined with a condition test boolean to enable loading the branch target address into the PC.
	Jump	Enables loading the jump target address into the PC (only appears in Figure 4.24 in Patterson and Hennessey).
Source Operand Fetch	ALUSrc	Selects the second source operand for the ALU (rt or sign-extended immediate field in Patterson and Hennessey).
ALU Operation	ALUOp	Either specifies the ALU operation to be performed or specifies that the operation should be determined from the function bits.
Memory Access	MemRead	Enables a memory read for load instructions.
	MemWrite	Enables a memory write for store instructions.
Register Write	RegWrite	Enables a write to one of the registers.
	RegDst	Determines how the destination register is specified (rt or rd in Patterson and Hennessey).
	MemtoReg	Determines where the value to be written comes from (ALU result or memory in Patterson and Hennessey).

Performance analysis



Forwarding används när värdet är beräknat innan nästkommande instruktion kommer till det steg där det är dags att använda värdet.

Stalling används när den andra instruktionens kommer innan värdet är beräknat i första instruktionen. (när forwarding inte fungerar) fetch is delayed.

Types of hazards

Data hazards = ex RAW (a **data hazard** occurs when an instruction reads a register that has not yet been written to.)

Control hazards = ex branch miss prediction

Branch delay slots

Since we have a pipelined processor with branch delay slots, the

instruction coming after the branch will always be executed, regardless if

the branch is taken or not.

Branch target adress (BTA)

BTA = PC + 4 + signext(imm) * 4

Modul 5 - Memory Hierarchies

Temporal locality

The processor is likely to access recently accessed addresses again.

Spatial Locality

The processor is likely to access addresses close to each other.

$$\text{Miss Rate} = \frac{\text{Number of misses}}{\text{Total number of memory accesses}}$$

$$\text{Hit Rate} = \frac{\text{Number of hits}}{\text{Total number of memory accesses}}$$

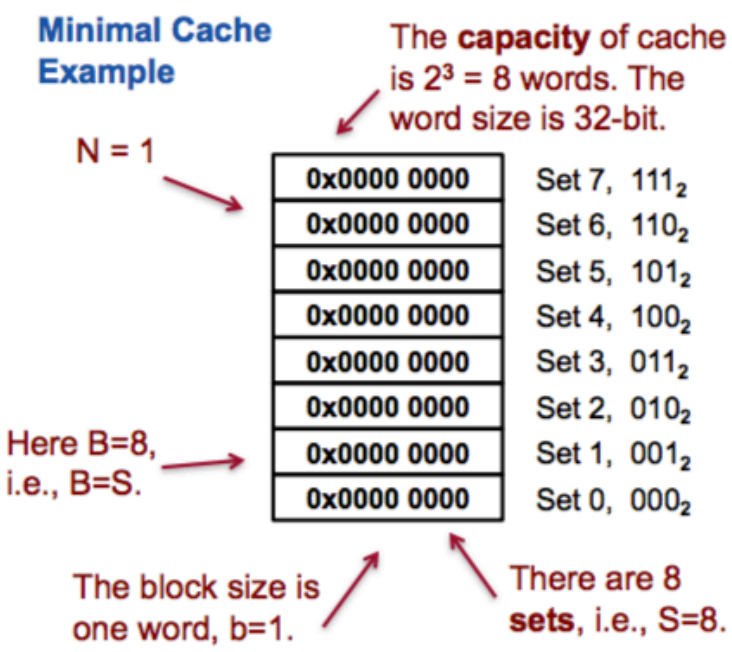
Capacity (C)
Number of words or bytes in the cache.

Number of Sets (S)
Each set holds one or more blocks of data. (Sometimes the term **row** is used instead of set)

Block size (b)
Number of words or bytes in a block.

Number of Blocks (B)
The total number of blocks.
Always: B >= S

Degree of associativity (N)
N = B/S



Set (S) kan ses som rader och blockstorleken (b) som kolumner.

Capacity C (total storlek i bytes) = B (antal blocks) * b (storlek/block)

Om N = 1 så är B = S

Men om du har flera ways, dvs N > 1. Då är B = S*N, för det finns S stycken set på varje ”way”. Och B är totala antalet blocks för ALLA ways, medans S är antalet blocks/way

---

The address bits can be divided into these three fields.

The ***byte offset*** determines which byte within the cache line (cache block) that is referenced. In this example, the byte offset field is 3 bits because the block size is 2^3 = 8 bytes.

The ***set* field** (also called the *index*) points to the row that should be accessed in the cache. In this case, there are 16 sets (16 rows), which means that the set field size is 4 bits ($2^4 = 16$)

If it is not a direct mapped cache, set field is calculated 4096/4/32 = 32 sets. C/N/B = x sets. Set field is the exponent, in this case 2^5 i.e. 5 bits.

The ***tag* field** of the address is used for checking if the correct cache line is actually located in the cache. The length of the tag field is the rest of the address. In this case the size of the tag field is 32 − 4 − 3 = 25 bits. Tag
checkar att det inte är annan data som ligger där. (32 kommer från antalet bitar processorn har. I detta fall 32).

Note that when a cache miss occurs, a whole cache block is always

fetched from the main memory.

When a cache miss occours, the three following parts are updated.

Valid bit sets to 1, tag is updated and the cache block will be updated.

N = B/S, dvs S = B/N

C/N/b = S

C/N = bytes in each set

Ordningen är tag, set, offset

När man inte räknar med direkt mappade cachar. Dela upp cachen per way.

Men då valid bits ska beräknas kom ihåg att det är en valid bits per rad. Dvs antal sets * N.

Ett cache block är 8 bytes

Cache coherence

A cache is used by the processor to get faster access to memory. The reason is locality, both temporal and spatial locality. That is, if the same address is accessed often or if addresses close to each other are accessed. The problem with cache coherence is a special problem that occurs in multicore systems with shared memory and separate caches for the different cores. The coherence problem can result in that different cores see different values for the same address.

Tillvägagångssätt för att beräkna cache hits i **instruktionscache**.

1, Beräkna all nödvändig information. Viktigast är set field och offset

2, Analysera den adress som programmet startar på.

3, När programmet startar är cachen tom, det vill säga resulterar i cache miss.

4, Nästa lagras då i samma set som den första försökte med. Dvs set field i basadressen.

5, PC ökar då med 4. Analysera den nya adressen och se om set har ändrats.

6, om det är samma set, resulterar det i en cache hit.

7, upprepa steg 5 till 6

Om programmet tillämpar spatial locality hämtas närliggande instruktioner. Dvs. Så många som får plats i varje set. Varje mips-instruktion är 4 bytes. Då hämtas det B/4 = x bytes. Det är därför det blir cache hit på de instruktioner som ligger i samma set.

Tillvägagångssätt för att beräkna cache hits i **datacache**.

Ex.

```
lui     $t1, 0x1001
lw      $t0, 0x8($t1)
lw      $t0, 0x4($t1)
lw      $t0, 0x20 ($t1)
lw      $t0, 0x4 ($t1)
```

Om man skriver till $t0, 0($t1) så letar vi i det minnet som finns på adressen som $t1 innehåller. Då cache:ar vi adressen som $t1 innehåller.

1, Analysera vilken adress som den första instruktionen använder. I detta fall hämtar första lw från 0x10010008

2, Då cache:ar vi den adressen i det set som adressen tillhör.

3, Eftersom cachen är tom blir det en cache miss och valid bit sätts till 1.

4, Därefter är det bara att titta vart nästa adress som vi läser ifrån ligger. I detta fall 0x10010004

Ex.
Set field = 6 bits
Offset = 4 bits

1, lw startar på 0x10010008 = bo 1000 och set field = 000 000 cache tom = miss

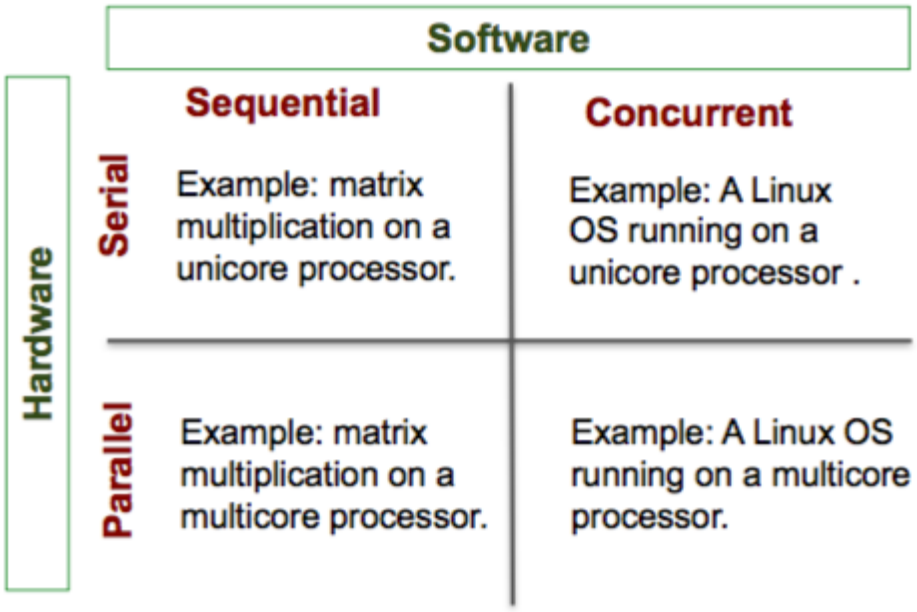
2, man läser nu från 0x10010004 = bo 0100 och set field 000 000 samma set cache hit

3, man läser från 0x10010020 = bo 0000 och set field 000 010. Nytt set resulterar i cache miss

4, man läser från 0x10010004 = bo 0100 och set field är gammalt, resulterar i hit

---

Modul 6 - Parallell processors and programs



Concurrency is about **handling** many things at the same time. Concurrency may be viewed from the **software** viewpointfrom the **hardware** viewpoint.

Weak scaling - Problem size increases proportionally to the number of cores.

Strong scaling - Problem size is fixed

$$Speedup = \frac{T_{before}}{T_{after}} = \frac{T_{before}}{\frac{T_{affected}}{N} + T_{unaffected}}$$

Assume that we perform 10 scalar integer additions, followed by one matrix addition, where matrices are 10x10. Assume additions take the same amount of time and that we can only parallelize the matrix addition.
Exercise A: What is the speedup with 10 processors?
Exercise B: What is the speedup with 40 processors?
Exercise C: What is the maximal speedup?

Solution A:
(10+10*10) / (10*10/10 + 10) = 5.5
Solution B:
(10+10*10) / (10*10/40 + 10) = 8.8
Solution C:
(10+10*10) / (10*10/N + 10) = 11 when N → infinity

Alternative solution for C
(10+10*10) / (10*10/100 + 10) = 10
if we assume that one add instruction cannot be parallelized

Main Classes of Parallelisms

Data-Level Parallelism (DLP)

Many data items can be processed at the same time.

Example – Sheep shearing

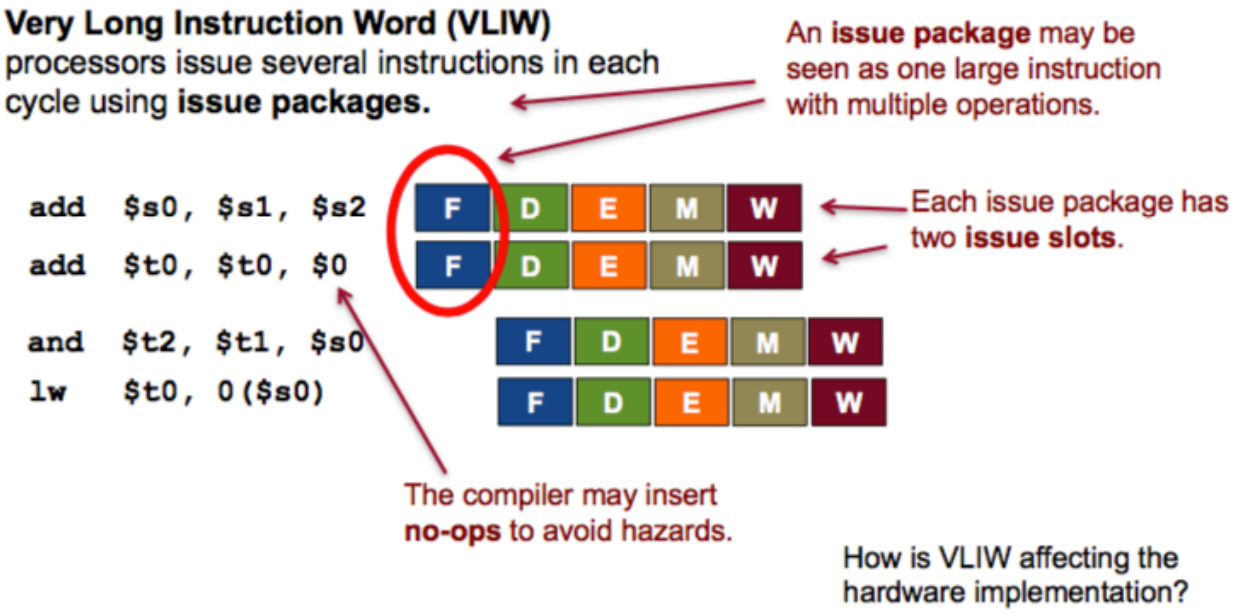
Assume that sheep are data items and the task for the farmer is to do sheep shearing (remove the wool). Data-level parallelism would be the same as using several farm hands to do the shearing.

Task-Level Parallelism (TLP)

Different tasks of work that can work in independently and in parallel

Example – Many tasks at the farm Assume that there are many different things that can be done on the farm (fix the barn, sheep shearing, feed the pigs etc.) Task-level parallelism would be to let the farm hands do the different tasks in parallel.

Instruction-Level Parallelism (ILP) may increase performance without involvement of the programmer. It may be implemented in a SISD, SIMD, and



MIMD computer.

Categorize the following keywords and concepts into one of the following two scenarios. Choose the scenario that fits best.

Keywords and concepts: SIMD, MIMD, hardware multithreading, VLIW, cache coher- ence, Advanced Vector Extension (AVX), MapReduce, and dynamic multiple issue.

- Scenario #1: A computer with a shared memory superscalar multicore (8 cores) processor with simultanious multithreading (SMT). The processor has separated L1 caches and a common L2 cache. The computer uses a modern Linux operating system.

- Scenario #2: A cluster system with 1000 computers connected by an Ethernet net- work. Each computer has an uniprocessor with static multiple issue and data-level parallelism at the instruction level. The processors have separated instruction and data caches. The cluster has software for massive big data computations.

Scenario #1 MIMD, Hardware multithreading, Cache coherence, dynamic multiple issue, AVX.

Scenario #2 SIMD, VLIW, MapReduce,

Superscalar

Det är basically att en processor kan dynamiskt hålla på med flera instruktioner samtidigt

Processorn kollar även vad det finns för beroenden och sen optimerar exekveringssekvensen av instruktionerna

VLIW är också ILP, fast då "paketeras" flera instruktioner i stora långa instruktioner

---

Simultaneous multithreading (SMT)

SMT är en blandning av hardware multithreading och superscalar Så den har flera threads där varje thread har förmågan att köra flera instruktioner per klockcykel.

Fine multithreadingär när den hoppar mellan threads varje klockcykel

---

2^0 = 1
2^1 = 2
2^2 = 4
2^3 = 8
2^4 = 16
2^5 = 32
2^6 = 64
2^7 = 128
2^8 = 256
2^9 = 512
2^10 = 1024
2^11 = 2048
2^12 = 4096
2^13 = 8192
2^14 = 16 384
2^15 = 32 768
2^16 = 65 536
2^17 = 131 072
2^18 = 262 144
2^19 = 524 288
2^20 = 1 048 576