Final Study Report: Servo Drive Architecture

CIS-655 ADVANCED COMPUTER ARCHITECTURE

PROF. MOHAMMED ABDALLAH

12/8/2021

Anthony Redamonti
SYRACUSE UNIVERSITY

Introduction:

The following processor/memory architecture is called the “Servo Drive Architecture.” It is intended to work in a servo drive which will control the amount of current applied to a servo motor. Embedded systems are common in the motion control industry, as systems must be compact and be able to efficiently perform a list of system specific functions.

Processor Architecture:

The servo drive architecture will use a 32-bit ARM processor. The processor will feature internal SRAM to be used with its A/D convertor. SRAM allows for faster memory access than DRAM and does not need to be periodically refreshed. An A/D convertor is an essential feature as some motion control masters command motion by transmitting data via an analog signal. The A/D convertor will convert the 0-10V DC signal to signed 32-bit commands. Also, there is a DAC to send an analog output from the drive to downstream devices. The output voltage range of the DAC is 0-5V DC. The CPU also includes an ALU that can perform a wide range of trigonometric operations using floating-point registers. The calculations needed for motion control applications require these trigonometric operations as well as 16-bit floating-point precision.

The ARM processor also features seven DMA channels which have access to the memory bus. DMA allows for overall improved system performance as memory transfers can be done without the use of the CPU. DMA is essential to maintain the real-time performance of the system. There is a real-time clock (RTC) used to program interrupts and a watchdog timer used to handle interrupts, exceptions, or system faults.

CPU Registers:

There are 32 general purpose registers. The registers are physically adjacent to the CPU to maximize performance and are 32-bits in size. Also, the upper 16 bits or lower 16 bits can be individually addressed. For example, $t1 is temporary register 1. Its upper 16 bits can be addressed as $th1 and the lower 16 bits as $tl1. The zero register ($zero) always stores the value zero to improve program efficiency.

Included are 6 floating-point registers, $f0-$f5, which can store decimal numbers in floating-point format. Because there are floating-point registers, the opcodes for trigonometry are more practical.

Table 1: General Purpose Registers

Register Number	Register Name	Description
0	$zero	The zero register. Stores the value zero. Reduces number of steps when performing calculations.
1	$at	Reserved for assembler
2	$v0	Store result of function call
3	$v1	Store result of function call
4	$a0	Store argument for function call
5	$a1	Store argument for function call
6	$a2	Store argument for function call
7	$a3	Store argument for function call
8	$t0	Holds a temporary value

9	$t1	Holds a temporary value
10	$t2	Holds a temporary value
11	$t3	Holds a temporary value
12	$t4	Holds a temporary value
13	$t5	Holds a temporary value
14	$f0	Holds a floating-point value. Up to 12-bit resolution.
15	$f1	Holds a floating-point value. Up to 12-bit resolution.
16	$f2	Holds a floating-point value. Up to 12-bit resolution.
17	$f3	Holds a floating-point value. Up to 12-bit resolution.
18	$f4	Holds a floating-point value. Up to 12-bit resolution.
19	$f5	Holds a floating-point value. Up to 12-bit resolution.
20	$s0	Holds content to use later.
21	$s1	Holds content to use later.
22	$s2	Holds content to use later.
23	$s3	Holds content to use later.
24	$s4	Holds content to use later.
25	$s5	Holds content to use later.
26	$k0	Kernel register zero
27	$k1	Kernel register one.
28	$gp	Global Pointer
29	$sp	Stack Pointer
30	$fp	Frame Pointer
31	$ra	Return Address

Instruction Set Architecture (ISA)

The instruction set architecture consists of 3 different instruction types: r-type, v-type, and j-type. R-type are register operations that use registers as arguments. V-type are register operations using registers and a signed value as arguments. J-type are jump operations that alter the control path of a program (jump to a memory address).

Below is a table of the opcodes. There are 53 in total. Most of them deal with complex mathematical operations needed to calculate a multi-axes trajectory (motion control path). Each is 32-bits in size.

For example, a common function of a servo drive is to perform two-dimensional path planning. The following calculation is repeatedly used to calculate the orientation of the positions of the axes in motion.

```
// use the atan2 function to accurately retrieve the current angle
//(orientation of the current position)
initialAngleRadians = atan2(deltaPosition[1], deltaPosition[0]);
```

The atan2 function is routinely performed, so it is given an opcode: 22. The same logic was applied when choosing the other opcodes. All the instructions listed in the ISA are functions that are routinely performed by a servo drive.

Table 2: Servo Drive ISA (53 Instructions)

Opcode	Name	Action	Opcode Bitfields				
readw	Read word	Rt=*(*int)(offset + Rs)	000000	Rs	Rt	signed 16-bit offset	
writew	Write word	*(*int)(offset + Rs) = Rt	000001	Rs	Rt	signed 16-bit offset	
add	Add reg	Rt = Rs + Rd	000010	Rs	Rt	Rd	00000000000
addv	Add val	Rt = Rs + (signed val)	000011	Rs	Rt	signed 16-bit value	
sub	Subtract reg	Rt = Rs – Rd	000100	Rs	Rt	Rd	00000000000
mult	Multiply reg	Rt = Rs * Rd	000101	Rs	Rt	Rd	00000000000
multv	Multiply val	Rt = Rs * (signed val)	000110	Rs	Rt	signed 16-bit value	
div	Divide reg	Rt = Rs/Rd	000111	Rs	Rt	Rd	00000000000
divrv	Divide reg val	Rt = Rs/val	001000	Rs	Rt	signed 16-bit value	
divvr	Divide val reg	Rt = val/Rs	001001	Rs	Rt	signed 16-bit value	
and	And reg	Rt = Rs & Rd	001010	Rs	Rt	Rd	00000000000
or	Or reg	Rt = Rs|Rd	001011	Rs	Rt	Rd	00000000000
xor	Xor reg	Rt = Rs^Rd	001100	Rs	Rt	Rd	00000000000
nand	Nand reg	Rt = ~(Rs & Rd)	001101	Rs	Rt	Rd	00000000000
nor	Nor reg	Rt = ~(Rs|Rd)	001110	Rs	Rt	Rd	00000000000
not	Not reg	Rt = ~(Rs)	001111	Rs	Rt	0x0000	
andv	And val	Rt = Rs & val	010000	Rs	Rt	signed 16-bit value	
orv	Or val	Rt = Rs|val	010001	Rs	Rt	signed 16-bit value	
xorv	Xor val	Rt = Rs^val	010010	Rs	Rt	signed 16-bit value	
nandv	Nand val	Rt = ~(Rs & val)	010011	Rs	Rt	signed 16-bit value	
setlt	Set less than	Rt = 1 if Rs < Rd. Else 0.	010100	Rs	Rt	Rd	00000000000
setltv	Set less than val	Rt = 1 if Rs < val. Else 0.	010101	Rs	Rt	signed 16-bit value	
atan2	Arctan2	Rt = arctan2(Rs/Rd)	010110	Rs	Rt	Rd	00000000000

rand	Random	Rt = Rand(Rs, Rd)	010111	Rs	Rt	Rd	00000000000
log	Logarithmic	$R_t = Log_{R_s}(R_d)$	011000	Rs	Rt	Rd	00000000000
pow	Power	$R_t = R_s^{R_d}$	011001	Rs	Rt	Rd	00000000000
hypot	Hypotenuse	$R_t = \sqrt{R_s^2 + R_d^2}$	011010	Rs	Rt	Rd	00000000000
shiftr	Shift Right	Rt = Rs >> value	011011	Rs	Rt	unsigned 16-bit value	
shiftl	Shift Left	Rt = Rs << value	011100	Rs	Rt	unsigned 16-bit value	
sin	Sine	Rt = sin(Rs)	011101	Rs	Rt	0x00	0x00
cos	Cosine	Rt = cos(Rs)	011101	Rs	Rt	0x00	0x01
tan	Tangent	Rt = tan(Rs)	011101	Rs	Rt	0x00	0x02
csc	Cosecant	Rt = csc(Rs)	011101	Rs	Rt	0x00	0x03
sec	Secant	Rt = sec(Rs)	011101	Rs	Rt	0x00	0x04
cot	Cotangent	Rt = cot(Rs)	011101	Rs	Rt	0x00	0x05
sinh	Hyperbolic Sine	Rt = sinh(Rs)	011101	Rs	Rt	0x00	0x06
cosh	Hyperb. Cosine	Rt = cosh(Rs)	011101	Rs	Rt	0x00	0x07
tanh	Hyperb. Tangent	Rt = tanh(Rs)	011101	Rs	Rt	0x00	0x08
asin	Arcsine	Rt = arcsin(Rs)	011101	Rs	Rt	0x00	0x09
acos	Arccosine	Rt = arccos(Rs)	011101	Rs	Rt	0x00	0x0a
atan	Arctan	Rt = arctan(Rs)	011101	Rs	Rt	0x00	0x0b
sqrt	Square Root	$R_t = \sqrt{R_s}$	011101	Rs	Rt	0x00	0x0c
flr	Floor	$R_t = \lfloor R_s \rfloor$	011101	Rs	Rt	0x00	0x0d
ceil	Ceiling	$R_t = \lceil R_s \rceil$	011101	Rs	Rt	0x00	0x0e
trunc	Truncate	Rt = trunc(Rs)	011101	Rs	Rt	0x00	0x0f
rnd	Round	$R_t \approx R_s$	011101	Rs	Rt	0x00	0x10
abs	Absolute Value	Rt = abs(Rs)	011101	Rs	Rt	0x00	0x11
deg	Degrees	Rt = Rs * (180/π)	011101	Rs	Rt	0x00	0x12
rad	Radians	Rt = Rs * (π/180)	011101	Rs	Rt	0x00	0x13
fact	Factorial	Rt = Rs!	011101	Rs	Rt	0x00	0x14
jump	Jump	Jump to address at (Rs + offset)	011110	Rs	00000	signed 16-bit offset	
jlink	Jump and Link	Jump and link to address at (Rs + offset)	011111	Rs	00000	signed 16-bit offset	
bequal	Branch if equal	If rs == rt, jump to PC + signed 16-bit offset	100000	Rs	Rt	signed 16-bit offset	

<u>Readw : Opcode = 000000</u>

Read a word of data from memory and store it in the register, Rt. The memory address is calculated as "Rs + address offset".

Bits	26-31	21-25	16-20	0-15
Meaning	Opcode	Rt	Rt	Address Offset

<u>Writew: Opcode = 000001</u>

Write a word of data stored in Rt to a memory address. The memory address is calculated as "Rs + address offset".

Bits	26-31	21-25	16-20	0-15
Meaning	Opcode	Rs	Rt	Address Offset

Add: Opcode = 000010

Add two registers together.

$$R_t = R_s + R_d$$

Bits	26-31	21-25	16-20	11-15	0-10
Meaning	Opcode	Rs	Rt	Rd	00000000000

Addv: Opcode = 000011

Add a signed integer value to a register.

$$R_t = R_s + Value$$

Bits	26-31	21-25	16-20	0-15
Meaning	Opcode	Rs	Rt	Signed value

Sub: Opcode = 000100

Subtract two registers.

$$R_t = R_s - R_d$$

Bits	26-31	21-25	16-20	11-15	0-10
Meaning	Opcode	Rs	Rt	Rd	00000000000

Mult: Opcode = 000101

Multiply two registers.

$$R_t = R_s * R_d$$

Bits	26-31	21-25	16-20	11-15	0-10
Meaning	Opcode	Rs	Rt	Rd	00000000000

Multv: Opcode = 000110

Multiply a register by a signed 16-bit value.

$$R_t = R_s * Value$$

Bits	26-31	21-25	16-20	0-15

Meaning	Opcode	Rs	Rt	Signed value

Div: Opcode = 000111

Divide two registers. Compatible with floating point and non-floating-point registers.

$$R_t = \frac{R_s}{R_d}$$

Bits	26-31	21-25	16-20	11-15	0-10
Meaning	Opcode	Rs	Rt	Rd	00000000000

Divrv: Opcode = 001000

Divide a register by a 16-bit value. Compatible with floating point and non-floating-point registers.

$$R_t = \frac{R_s}{Value}$$

Bits	26-31	21-25	16-20	0-15
Meaning	Opcode	Rs	Rt	Signed value

Divvr: Opcode = 001001

Divide a 16-bit value by a register. Compatible with floating point and non-floating-point registers.

$$R_t = \frac{Value}{R_s}$$

Bits	26-31	21-25	16-20	0-15
Meaning	Opcode	Rs	Rt	Signed value

And: Opcode = 001010

Bitwise AND operation.

$$R_t = R_s \,\&\, R_d$$

Bits	26-31	21-25	16-20	11-15	0-10
Meaning	Opcode	Rs	Rt	Rd	00000000000

Or: Opcode = 001011

Bitwise OR operation.

$$R_t = R_s \mid R_d$$

Bits	26-31	21-25	16-20	11-15	0-10
Meaning	Opcode	Rs	Rt	Rd	00000000000

Xor: Opcode = 001100

Bitwise XOR operation.

$$R_t = R_s \wedge R_d$$

Bits	26-31	21-25	16-20	11-15	0-10
Meaning	Opcode	Rs	Rt	Rd	00000000000

Nand: Opcode = 001101

Bitwise NAND operation.

$$R_t = \sim(R_s \,\&\, R_d)$$

Bits	26-31	21-25	16-20	11-15	0-10
Meaning	Opcode	Rs	Rt	Rd	00000000000

Nor: Opcode = 001110

Bitwise NOR operation.

$$R_t = \sim(R_s \mid R_d)$$

Bits	26-31	21-25	16-20	11-15	0-10
Meaning	Opcode	Rs	Rt	Rd	00000000000

Not: Opcode = 001111

Bitwise NOT operation.

$$R_t = \sim(R_s)$$

Bits	26-31	21-25	16-20	0-15
Meaning	Opcode	Rs	Rt	0x0000

Andv: Opcode = 010000

Bitwise AND operation with a register and a 16-bit value.

$$R_t = R_s \,\&\, Value$$

Bits	26-31	21-25	16-20	0-15
Meaning	Opcode	Rs	Rt	Signed value

Orv: Opcode = 010001

Bitwise OR operation with a register and a 16-bit value.

$$R_t = R_s \mid Value$$

Bits	26-31	21-25	16-20	0-15
Meaning	Opcode	Rs	Rt	Signed value

<u>Xorv: Opcode = 010010</u>

Bitwise XOR operation with a register and a 16-bit value.

$$R_t = R_s \ \hat{}\ Value$$

Bits	26-31	21-25	16-20	0-15
Meaning	Opcode	Rs	Rt	Signed value

<u>Nandv: Opcode = 010011</u>

Bitwise NAND operation with a register and a 16-bit value.

$$R_t = \sim(R_s \,\&\, Value)$$

Bits	26-31	21-25	16-20	0-15
Meaning	Opcode	Rs	Rt	Signed value

<u>Setlt: Opcode = 010100</u>

Set Rt to 1 if Rs is less than Rd. Set Rt to 0 otherwise.

$$R_t = 1\ if\ R_s < R_d. Else, R_t = 0.$$

Bits	26-31	21-25	16-20	11-15	0-10
Meaning	Opcode	Rs	Rt	Rd	00000000000

<u>Setltv: Opcode = 010101</u>

Set Rt to 1 if Rs is less than the 16-bit value. Set Rt to 0 otherwise.

$$R_t = 1\ if\ R_s < Value. Else, R_t = 0.$$

Bits	26-31	21-25	16-20	0-15
Meaning	Opcode	Rs	Rt	Signed value

<u>Atan2: Opcode = 010110</u>

Rt equals the atan2 of Rs over Rd.

$$R_t = atan2\left(\frac{R_s}{R_d}\right)$$

Bits	26-31	21-25	16-20	11-15	0-10
Meaning	Opcode	Rs	Rt	Rd	00000000000

Rand: Opcode = 010111

Rt will be a random number between Rs and Rd (inclusive).

$$R_s \leq R_t = random\,number \leq R_d$$

Bits	26-31	21-25	16-20	11-15	0-10
Meaning	Opcode	Rs	Rt	Rd	00000000000

Log: Opcode = 011000

Rt is equal to the logarithm base Rs of Rd.

$$R_t = \log_{R_s} R_d$$

Bits	26-31	21-25	16-20	11-15	0-10
Meaning	Opcode	Rs	Rt	Rd	00000000000

Pow: Opcode = 011001

Rt is equal to Rs raised to the power of Rd.

$$R_t = R_s^{R_d}$$

Bits	26-31	21-25	16-20	11-15	0-10
Meaning	Opcode	Rs	Rt	Rd	00000000000

Hypot: Opcode = 011010

Rt is the hypotenuse of a right triangle where the adjacent sides are of length Rs and Rd.

$$R_t = \sqrt{R_s^2 + R_d^2}$$

Bits	26-31	21-25	16-20	11-15	0-10
Meaning	Opcode	Rs	Rt	Rd	00000000000

Shiftr: Opcode = 011011

Rt is equal to Rs >> Rd.

$$R_t = R_s \gg Value$$

Bits	26-31	21-25	16-20	0-15
Meaning	Opcode	Rs	Rt	Signed value

Shiftl: Opcode = 011100

Rt is equal to Rs << Rd.

$$R_t = R_s \ll Value$$

Bits	26-31	21-25	16-20	0-15
Meaning	Opcode	Rs	Rt	Signed value

<u>Sin: Opcode = 011101, Function Code = 0x00</u>

Rt is equal to the sine of Rs.

$$R_t = \sin(R_s)$$

Bits	26-31	21-25	16-20	8-15	0-7
Meaning	Opcode	Rs	Rt	0x00	Function Code

<u>Cos: Opcode = 011101, Function Code = 0x01</u>

Rt is equal to the cosine of Rs.

$$R_t = \cos(R_s)$$

Bits	26-31	21-25	16-20	8-15	0-7
Meaning	Opcode	Rs	Rt	0x00	Function Code

<u>Tan: Opcode = 011101, Function Code = 0x02</u>

Rt is equal to the tangent of Rs.

$$R_t = \tan(R_s)$$

Bits	26-31	21-25	16-20	8-15	0-7
Meaning	Opcode	Rs	Rt	0x00	Function Code

<u>Csc: Opcode = 011101, Function Code = 0x03</u>

Rt is equal to the cosecant of Rs.

$$R_t = \csc(R_s)$$

Bits	26-31	21-25	16-20	8-15	0-7
Meaning	Opcode	Rs	Rt	0x00	Function Code

<u>Sec: Opcode = 011101, Function Code = 0x04</u>

Rt is equal to the secant of Rs.

$$R_t = \sec(R_s)$$

Bits	26-31	21-25	16-20	8-15	0-7
Meaning	Opcode	Rs	Rt	0x00	Function Code

Cot: Opcode = 011101, Function Code = 0x05

Rt is equal to the cotangent of Rs.

$$R_t = \cot(R_s)$$

Bits	26-31	21-25	16-20	8-15	0-7
Meaning	Opcode	Rs	Rt	0x00	Function Code

Sinh: Opcode = 011101, Function Code = 0x06

Rt is equal to the hyperbolic sine of Rs.

$$R_t = \sinh(R_s)$$

Bits	26-31	21-25	16-20	8-15	0-7
Meaning	Opcode	Rs	Rt	0x00	Function Code

Cosh: Opcode = 011101, Function Code = 0x07

Rt is equal to the hyperbolic cosine of Rs.

$$R_t = \cosh(R_s)$$

Bits	26-31	21-25	16-20	8-15	0-7
Meaning	Opcode	Rs	Rt	0x00	Function Code

Tanh: Opcode = 011101, Function Code = 0x08

Rt is equal to the hyperbolic tangent of Rs.

$$R_t = \tanh(R_s)$$

Bits	26-31	21-25	16-20	8-15	0-7
Meaning	Opcode	Rs	Rt	0x00	Function Code

Asin: Opcode = 011101, Function Code = 0x09

Rt is equal to the arcsine of Rs.

$$R_t = \mathrm{asin}(R_s)$$

Bits	26-31	21-25	16-20	8-15	0-7
Meaning	Opcode	Rs	Rt	0x00	Function Code

Acos: Opcode = 011101, Function Code = 0x0a

Rt is equal to the arccosine of Rs.

$$R_t = \text{acos}(R_s)$$

Bits	26-31	21-25	16-20	8-15	0-7
Meaning	Opcode	Rs	Rt	0x00	Function Code

<u>Atan: Opcode = 011101, Function Code = 0x0b</u>

Rt is equal to the arctangent of Rs.

$$R_t = \text{atan}(R_s)$$

Bits	26-31	21-25	16-20	8-15	0-7
Meaning	Opcode	Rs	Rt	0x00	Function Code

<u>Sqrt: Opcode = 011101, Function Code = 0x0c</u>

Rt is equal to the square root of Rs.

$$R_t = \sqrt{R_s}$$

Bits	26-31	21-25	16-20	8-15	0-7
Meaning	Opcode	Rs	Rt	0x00	Function Code

<u>Flr: Opcode = 011101, Function Code = 0x0d</u>

Rt is equal to the floor of Rs.

$$R_t = \lfloor R_s \rfloor$$

Bits	26-31	21-25	16-20	8-15	0-7
Meaning	Opcode	Rs	Rt	0x00	Function Code

<u>Ceil: Opcode = 011101, Function Code = 0x0e</u>

Rt is equal to the ceiling of Rs.

$$R_t = \lceil R_s \rceil$$

Bits	26-31	21-25	16-20	8-15	0-7
Meaning	Opcode	Rs	Rt	0x00	Function Code

<u>Trunc: Opcode = 011101, Function Code = 0x0f</u>

Rt is equal to the truncate of Rs.

$$R_t = \text{trunc}(R_s)$$

Bits	26-31	21-25	16-20	8-15	0-7
Meaning	Opcode	Rs	Rt	0x00	Function Code

<u>Rnd: Opcode = 011101, Function Code = 0x10</u>

Rt is equal to the rounded value of Rs.

$$R_t \approx R_s$$

Bits	26-31	21-25	16-20	8-15	0-7
Meaning	Opcode	Rs	Rt	0x00	Function Code

<u>Abs: Opcode = 011101, Function Code = 0x11</u>

Rt is equal to the absolute value of Rs.

$$R_t = \text{abs}(R_s)$$

Bits	26-31	21-25	16-20	8-15	0-7
Meaning	Opcode	Rs	Rt	0x00	Function Code

<u>Deg: Opcode = 011101, Function Code = 0x12</u>

Rs is in radians. Rt equals the conversion of Rs to degrees.

$$R_t = R_s * \left(\frac{180}{\pi}\right)$$

Bits	26-31	21-25	16-20	8-15	0-7
Meaning	Opcode	Rs	Rt	0x00	Function Code

<u>Rad: Opcode = 011101, Function Code = 0x13</u>

Rs is in degrees. Rt equals the conversion of Rs to radians.

$$R_t = R_s * \left(\frac{\pi}{180}\right)$$

Bits	26-31	21-25	16-20	8-15	0-7
Meaning	Opcode	Rs	Rt	0x00	Function Code

<u>Fact: Opcode = 011101, Function Code = 0x14</u>

Rt equals the factorial of Rs.

$$R_t = R_s!$$

Bits	26-31	21-25	16-20	8-15	0-7
Meaning	Opcode	Rs	Rt	0x00	Function Code

<u>Jump: Opcode = 011110</u>

Jump to address at Rs + offset. The offset is a signed 16-bit integer.

Bits	26-31	21-25	16-20	0-15
Meaning	Opcode	Rs	00000	Signed 16-bit offset

Jlink: Opcode = 011111

Jump and link to address at Rs + offset. The offset is a signed 16-bit integer.

Bits	26-31	21-25	16-20	0-15
Meaning	Opcode	Rs	00000	Signed 16-bit offset

Bequal: Opcode = 100000

Jump to address at PC + offset if Rs == Rt. The offset is a signed 16-bit integer.

Bits	26-31	21-25	16-20	0-15
Meaning	Opcode	Rs	Rt	Signed 16-bit offset

1.3 Processor Architecture and Characteristics

The Servo Drive processor architecture is multicore, meaning that there are multiple processing units executing instructions simultaneously. The reason for a multicore system in this application is that there are multiple axes controlled with one processor. All axes must be handled in real-time simultaneously. The processor architecture also supports multithreading. Because there are multiple axes being commanded in real-time, threads must share the resources of the CPU to update downstream devices with feedback information. Data is sent to peripheral devices (serial port, CAN message bus, EtherCAT message bus) through the SPI interface. The number of cores is directly correlated with the number of axes supported by the servo drive. There should be 2 cores per axis.

Even though the servo-drive architecture supports real-time applications, the number of threads will be limited by the requirements of the application. Therefore, a coarse-grained multithreading approach is ideal for this architecture since it requires fewer threads to keep the CPU busy than a fine-grained multithreading approach.

The ARM processor uses the five-stage pipelining model: Fetch (F), Decode (D), Execute (X), Memory Access (M), and Writeback (W). If there are structural hazards (i.e., two instructions require the same hardware recourse), the CPU will add stall cycles to wait for the recourse to become available. If there are data hazards (data dependencies), the CPU will stall since there is no forwarding unit. If there are control hazards, there is a two-bit branch prediction buffer to allow a more accurate prediction mechanism. Only if two incorrect guesses are made will the guess be changed, resulting in fewer incorrect guesses.

2.1 Memory Structure

The Servo Drive memory structure will have a three-level cache. The level 1 cache is part of the CPU chip and is not shared between cores. Cache levels 2 and 3 are shared between CPU cores. A three-level cache is necessary because there needs to be a 30KB buffer shared between cores. Each axis in a servo drive is controlled by a separate CPU core. Each will share target position and velocity data stored in this buffer, located in the level 3 cache.

Instruction and data cache will be separate for all three levels and encoded in little-endian format. Separate instruction and data cache is more advantageous in terms of pipelining as the fetch and decode stages can be performed in parallel. The split design also allows for reduced latency as the instruction cache can be placed close to the instruction fetch unit and the data cache can be placed close to the memory unit.

The instructions that access memory are the load word (lw) and store word (sw) commands.

The size of the first level cache is 16,384 bytes (16KB). Each block contains $64\ bytes = 2^6\ bytes$. Therefore, the offset needs to be 6 bits long to select a byte inside of a block. The number of blocks in the cache is $\frac{16{,}384\ bytes}{\frac{64\ bytes}{block}} = 256\ blocks$. Since the system is 4-way associative, there are 4 blocks per set. Therefore, the number of sets in the cache is $\frac{256\ blocks}{\frac{4\ blocks}{set}} = 64\ sets$. $2^6 = 64$, so 6 bits are needed in the index field. The rest of the bits are used as the tag.

Bits	12-31	6-11	0-5
Meaning	Tag	Index	Offset

The size of the second level cache is 32,768 bytes (32KB). Following the same steps in the previous calculation, the below bit mapping is produced. Notice that one extra bit is needed in the index field.

Bits	13-31	6-12	0-5
Meaning	Tag	Index	Offset

The size of the third level cache is 65,536 bytes (65KB). Following the same steps in the previous calculation, the below bit mapping is produced. Notice that one extra bit is needed in the index field.

Bits	14-31	6-13	0-5
Meaning	Tag	Index	Offset

The replacement architecture is Least Recently Used (LRU). In the event of a cache miss, a block will need to be replaced to make room for a new block. The LRU replacement strategy replaces the least recently used block with the new block. The LRU algorithm is expensive to maintain as age bits are assigned to each block in cache. The age bits represent when the block was last accessed. LRU works to keep the blocks in cache that are frequently used, which decreases the number of cache misses. The reason LRU was chosen is to mitigate the number of cache misses, which are very costly in a real-time system.

The Servo Drive Broadcaster: A Memory Sharing Protocol

The Servo Drive Broadcaster is a memory sharing protocol that is similar to the bus snooping protocol (snoopy). When data is shared across different caches, there needs to be a protocol that maintains the validity of the data. There are three different states that a block of data can have: Invalid (I), Shared-ok (S), and Modified (M). The 11 cases of the servo drive broadcasting protocol are below.

1. If the state of a block is shared (S) between processor 1 and processor 2, and processor 1 performs a local read, the block will remain shared (S).
2. If the state of a block in processor 1’s cache is modified (M), and processor 1 performs a local read on that block, the block remains modified (M).
3. If the state of a block in processor 1’s cache is modified (M), and processor 1 performs a local write, the block remains modified (M).
4. If the state of a block in processor 1’s cache is invalid (I), and processor 1 performs a local write on that block, the block becomes modified (M).
5. If the state of a block in processor 1’s cache is modified (M) and processor 2 modifies that block, the block is invalid (I).
6. If the state of a block in processor 1’s cache is modified (M) and processor 2 reads that block, the state of the block will be shared (S).
7. If the state of a block in processor 1’s cache is invalid (I) and processor 1 reads that block (local read), the block will become shared-ok (S). The block is shared or ok to be shared.
8. Normally in the snoopy protocol, if the state of a block in processor 1’s cache is shared-ok (S) and processor 2 modifies that block (write request), the state of the block will be invalid (I).

However, the servo drive broadcaster will perform a “write update” in this event, which will modify all copies of the data. Therefore, processor 1’s cache will be automatically updated by processor 2 and will remain shared-ok (S).

9. If the state of a block in processor 1’s cache is shared-ok (S) and processor 2 reads that block (read request), the state of the block will remain shared-ok (S).
10. Normally in the snoopy protocol, if the state of a block in processor 1’s cache is shared-ok (S) and processor 1 performs a local write, the state of the block will be modified (M). However, the servo drive broadcaster will perform a “write update” in this event, which will modify all copies of the data. Therefore, processor 1’s cache will remain shared-ok (S).
11. If the state of a block in processor 1’s cache is Invalid (I) and processor 2 performed a read or a write (read or write request), the state of the block remains invalid (I).

As seen in cases 8 and 10, the servo drive broadcaster protocol uses the “write update” strategy, which means in the event of a write, all other copies of shared data are updated as well. Because of its longer message size, it requires more bandwidth than the “write invalidate” method. However, the servo drive system has plenty of bandwidth and a limited number of parameters to update (embedded application). Also, by updating the shared data, there will be fewer cache misses, which is essential in a real-time application.