

CE/CZ3001-Advanced Computer Architecture

Project Report

By Group 19

*Chandra Bhavya U1322721J*

*Jain Noopur U1323005A*

*Bansal Ankur U1322989K*

*Seshadri Madhavan U1322790J*

PART 1

# TestBench Screenshots for BEQ, LW, SW, R and I type instuctions for a 5-staged pipeline

# Explaination of the working of LW,SW and BEQ

**16-BIT INSTRUCTION FORMAT:**

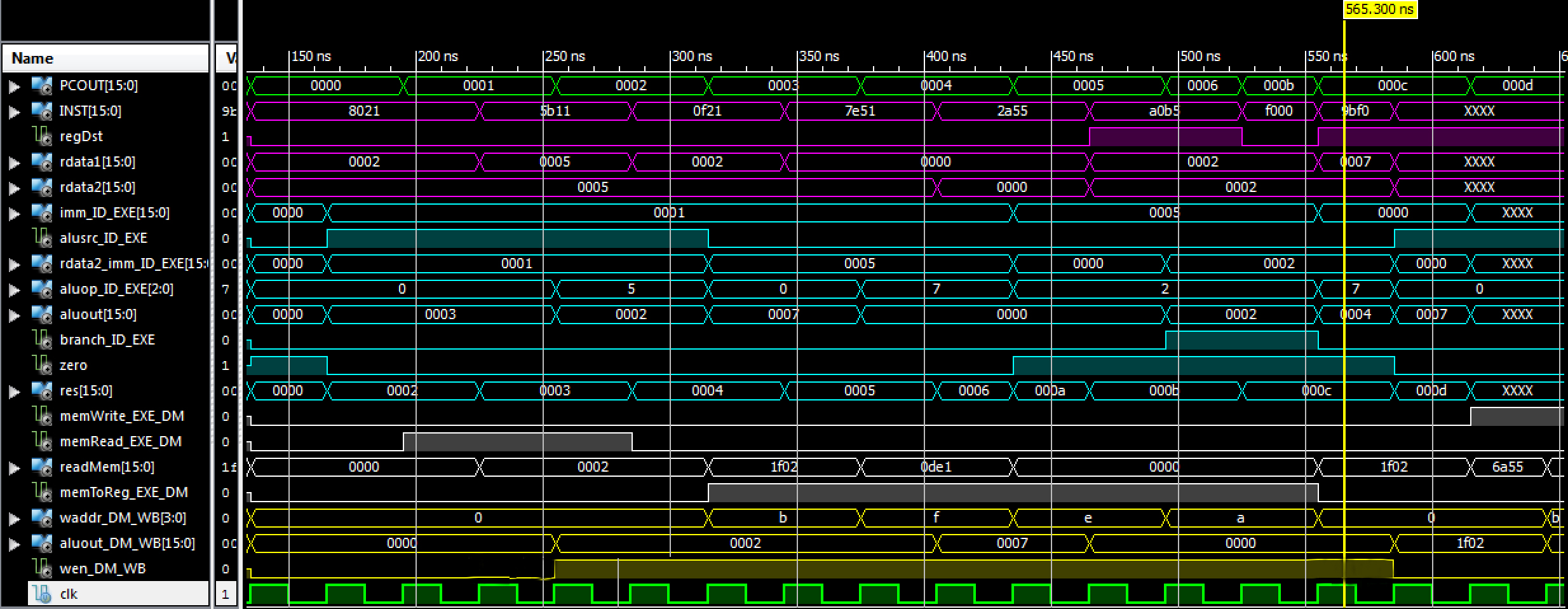
|  |  |  |  |
| --- | --- | --- | --- |
| Op-code | Rd | Rs | Rt/Immediate |
| 4-bit | 4-bit | 4-bit | 4-bit |

A 5-stage pipeline was implemented, and the test bench was analyzed as shown in Figure 1.1, Figure 1.2 and Figure 1.3 below containing LW, SW , R , I and BEQ instructions.

Table 1.1 below shows the assembly code and its MIPS translation for the same.

|  |  |  |
| --- | --- | --- |
| Instruction test file | MIPs | Data memory test file |
| 0000 8021  0001 5b11  0002 0f21  0003 7e51  0004 2a55  0005 a0b5  0006 f000    0007 f000  0008 f000  0009 9bf0  000a 0de2  000b 9bf0 | lw $0,1($2)  srl $b,$1,1  add $F,$2,$1  mul $E,$5,$1  and $A,$5,$5  beq $0,$B,5  nop  sw $b,0($f)  (lw, sw, srl, add, BEQ-all 5 kinds of instructions have been used in the code.) | 0000 0000  0001 0531  0002 1f02  0003 0002  0004 6a55  0005 26a2  0006 3fa5  0007 0de1  0008 5bd1  0009 3b13 |

Figure 1.1 explains the working of Load word; Figure 1.2 explains the working of BEQ and Figure 1.3 explains the working of Store word. The test bench screenshots also incorporate other R & I type instructions such as srl $b, $1, 1 and add $F, $2, $1.



*Figure 1.1 LW instructions*

*Working:* The 5-stage pipeline flow for a load word instruction is highlighted in the test bench above. Lw $rd, imm($rs) means rd<-Mem[imm+$rs].

This is the highlighted instruction 0x8021 (as LW is defined by the opcode 8) meaning reg0=[reg2+imm] value.

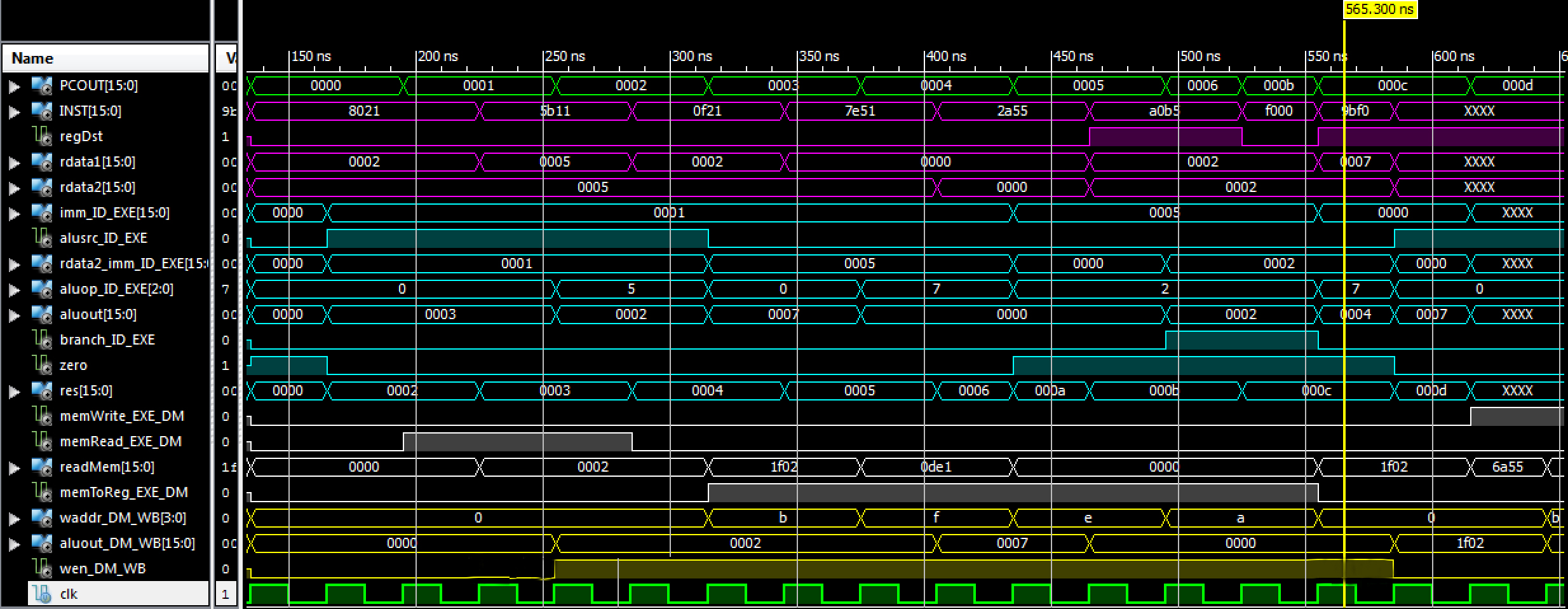
The **Fetch(F)** stage brings the instruction from the memory location 0x0000

**Decode(D)** stage determines the control signals such as *regDst, memRead, memWrite, aluSRC, Wen, AluOp* (which maybe passed further in the pipeline). It retrieves data from $rs i.e $2 and stores it in rdata1=0x0002. We do not care about Rdata2 as *aluSRC* will select the immediate value.

**Execute(E)** stage: The immediate value=0x1. Control Signal aluSRC=1, therefore the alu receives Rdata1 and the immediate value as the input along with control signal aluOp=0 for addition. The aluout=Rdata1+imm value->0x0002+0x1=0x0003. This is pipelined to the DataMemory Stage with the remaining control signals.

**Data Memory(DM) Stage**: The control signal for memRead is high, the address fed into data memory(=aluout)=0x0003. From table 1.1 we can see that the value 0x0002 is fetched. Since we need to store this value in $0, control signal memToReg is set to 0.

**Write Back(WB) stage:** Finally the control signal Wen was set to 1, the address is set to $0 and wdata is 0002. This is stored in reg0.



*Figure 1.2 BEQ instructions*

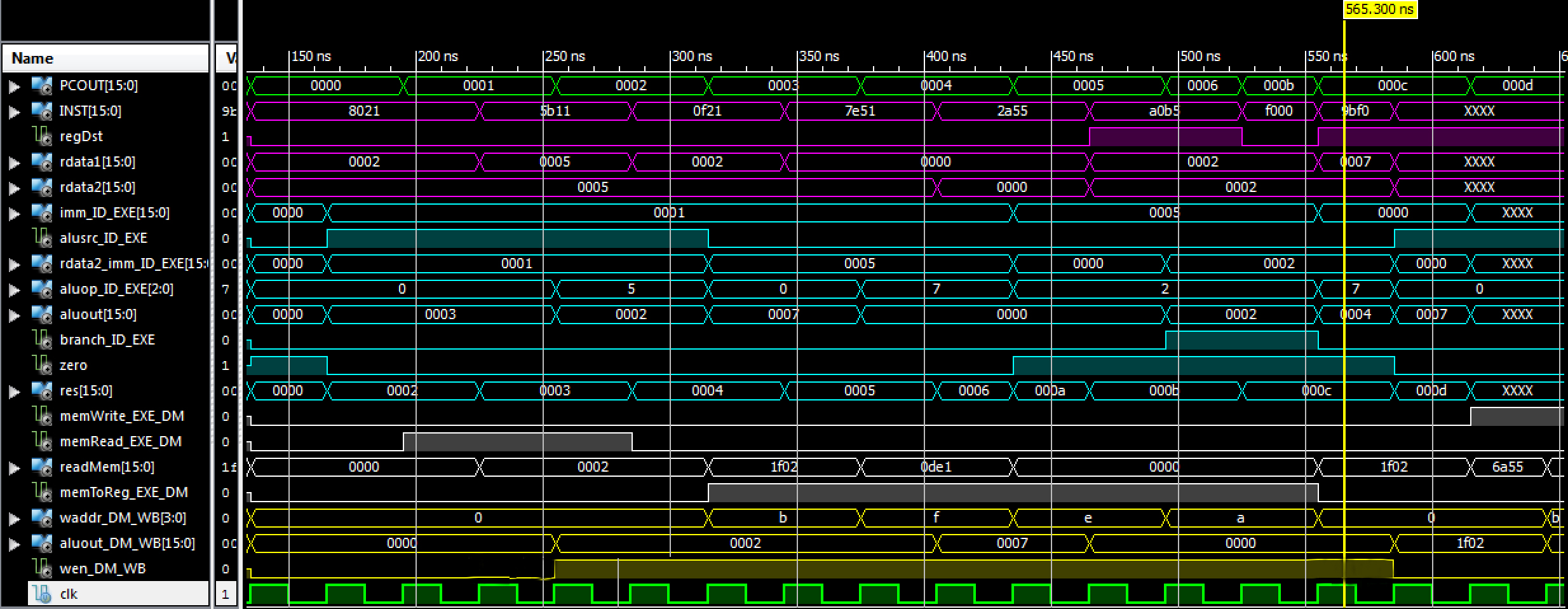
*Working:* The 5-stage pipeline flow for a Branch if equal instruction is highlighted in the test bench above. beq $rs,$rd, imm means PC<-nPC+imm if $rs=$rd. This is the highlighted instruction 0xa0b5 (as BEQ is defined by the opcode A) meaning if reg b=reg 0 then PC=PC+5.

The **Fetch(F)** stage brings the instruction from the memory location 0x0005. The next PC is incremented to 0x0006.

**Decode(D)** stage determines the control signals such as *regDst, memRead, memWrite, aluSRC, Wen, AluOp* (which maybe passed further in the pipeline). Rdata1=$b 0x0002 in this case, and control signal regDest is high so Rdata2=$0 i.e. 0x0002 . The immediate value is 0x5.

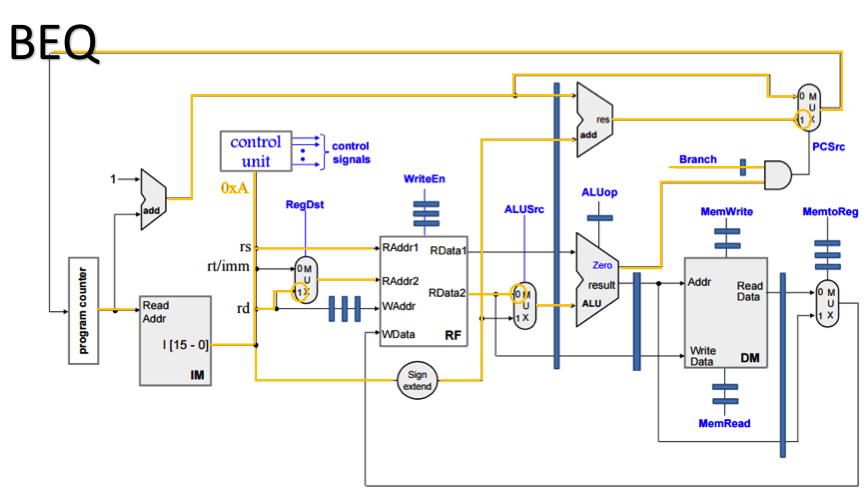
**Execute(E)** stage: . Since Rdata1==Rdata2, the value of signal zero is HIGH. The control signal BRANCH is also HIGH. The immediate value is added to the next Pc value i.e. 0x6+0x5 =0xB. This is stored in res. Since both Branch & Zero are HIGH, the next value of PC is updated to res instead of PC+1.

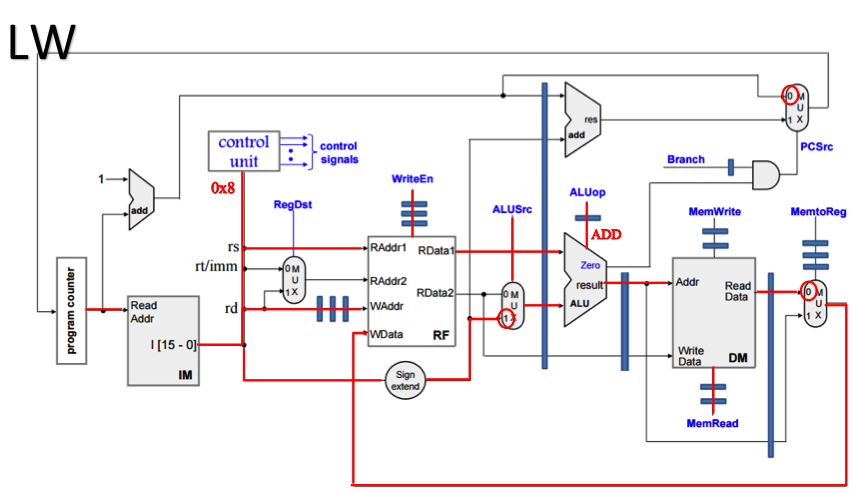
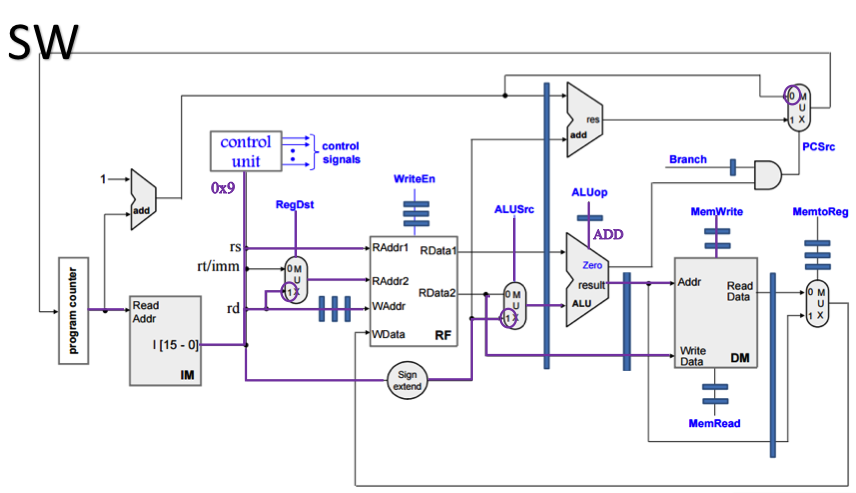
**This takes 3 stages only.** The pipeline needs to be stalled by a 1 NOP, because the next instruction is fetched while the BEQ instruction is being decoded. **Not executed:** Data Memory (DM) Stage, Write Back(WB) stage *(Wen and other control signal are 0)*



*Figure 1.3 SW instructions*

|  |  |
| --- | --- |
| *Working:* The 5-stage pipeline flow for a store word instruction is highlighted in the test bench above. sw $rd, imm($rs) means Mem[imm+$rs]<-$rd. | *Figure 1.3.a Memory location 7 is changed to 2* |
| This is the highlighted instruction 0x9bf0 (as SW is defined by the opcode 9) meaning Mem[$f+0]=$B. |
| The **Fetch(F)** stage brings the instruction from the memory location 0x000b.  **Decode(D)** stage determines the control signals such as *regDst, memRead, memWrite, aluSRC, Wen, AluOp* (which maybe passed further in the pipeline). It retrieves data from $rs i.e $f and stores it in rdata1=0x0007. regDest is high so Rdata2=$0 i.e. 0x0002 . The immediate value is 0x0. |
| **Execute(E)** stage: The immediate value=0x0. Control Signal aluSRC=1, therefore the alu receives Rdata1 and the immediate value as the input along with control signal aluOp=0 for addition. The aluout=Rdata1+imm value->0x0007+0x0=0x0007. This is pipelined to the DataMemory Stage with the remaining control signals. Further Rdata2 i.e. 0x0002 is also pipelines as this is the data needed to be stored in the DM.  **Data Memory(DM) Stage**: The control signal for memRead is LOW and memWrite is HIGH, the address fed into data memory(=aluout)=0x0007. The Write data is 0x0002(From Rdata2) and this value is finally stored in memory location 0x0007.  **Write Back(WB) stage:** This stage is not required for SW. | |

The following figures 1.4a, 1.4b and 1.4c provide a summary of the path taken by LW, SW and BEQ.



*Additional note:*

A dummy register was added to pipeline AluOut , Wen, MemToReg and few other signals in the DataMemory stage since the data memory contains internal registers and the values were required to be synced.

# Synthesis Report for Part-1.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| CPU | BITWIDTH | No of LUT slices | No of registers | minimum period in ns |
| Five-stage Pipeline | 16 | 435 | 784 | 6.404ns |

PART 2

# Test-Bench Screenshots for JUMP, JR, JAL, R and I type instructions for a 5-staged pipeline

# Explanation of the working of JUMP, JR, JAL

|  |  |  |
| --- | --- | --- |
| Instruction test file | MIPs | Data memory test file |
| 0000 0012  0001 5f11  0002 c009  0003 f000  0004 0000  0005 d00b  0006 f000  0007 d00b  0008 f000  0009 b010  000a f000  000b 0012 | add $0,$1,$2  srl $f,$1,$1  jump 9  nop  add $0,$0,$0  jal b  nop  jal b  nop  jr $1  nop  add $0,$1,$2  (jal,jump,jr, add, srl-all 5 kinds of instructions have been used in the code.) | 0000 0000  0001 0531  0002 1f02  0003 0002  0004 6a55  0005 26a2  0006 3fa5  0007 0de1  0008 5bd1  0009 3b13 |

Figure 2.1 explains the working of JUMP instruction; Figure 2.2 explains the working of JAL instruction and Figure 2.3 explains the working of JR. The test bench screenshots also incorporate other R & I type instructions such as srl $f, $1, $1 and add $0, $0, $0.

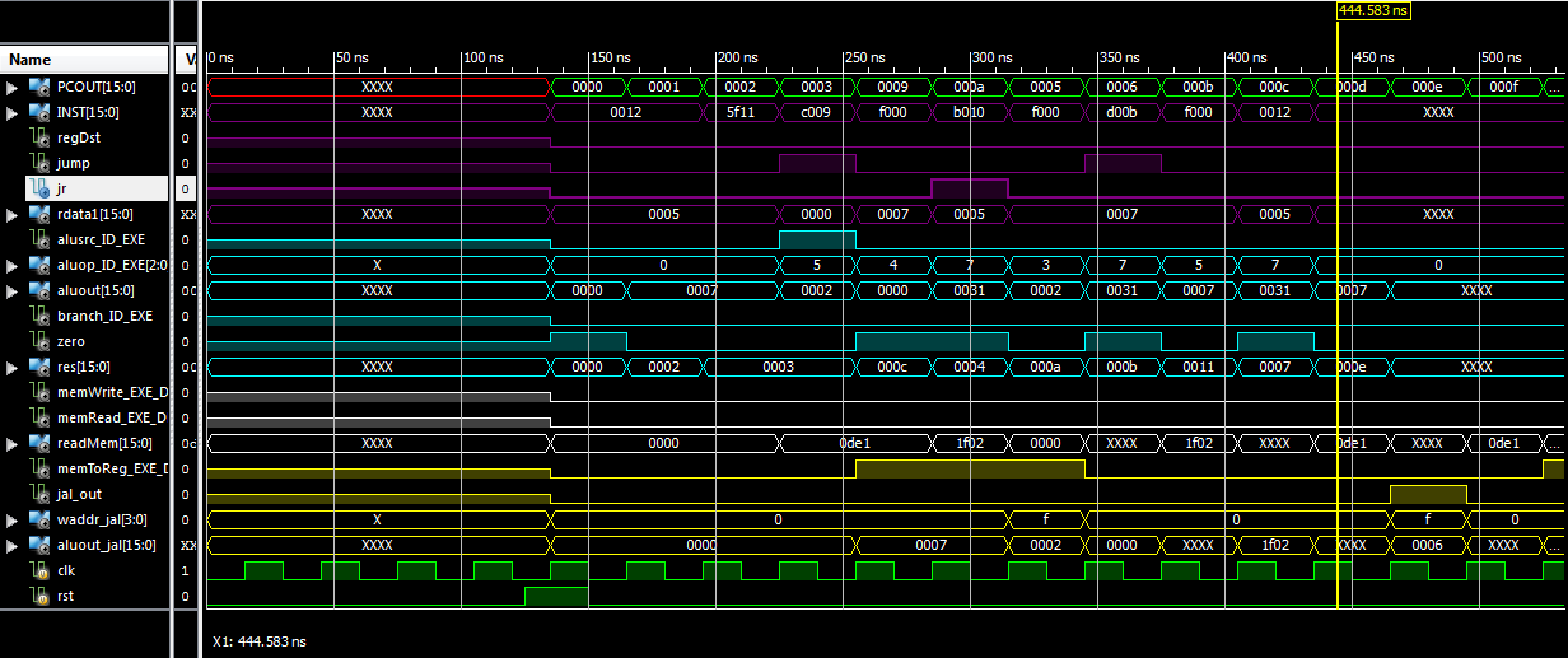


Figure 2.1: JUMP instructions

## Working:

The 5-stage pipeline flow for a jump instruction is highlighted in the test bench above. JUMP offset means PC<-IMem[offset].

This is the highlighted instruction 0xc009 (as JUMP is defined by the opcode c) meaning Jump to instruction 0x009.

The **Fetch(F)** stage brings the instruction from the memory location 0x0002

**Decode(D)** stage determines the control signals such as *regDst, memRead, memWrite, aluSRC, Wen, AluOp, jump, jr, jal* (which maybe passed further in the pipeline). The Jump signal is enabled and it concatenates the next instruction as follows PC[15:12]+I[11:0]. After concatenation the value is passed to the multiplexer and as the Jump signal is enabled, it is passed on to the Program Counter as the next instruction to be executed. I[11:0] denotes the address of the instruction to be jumped to. PC[15:12] is concatenated to keep it within the address space.

**The JUMP instruction does not make use of the Execute(E), Data Memory(DM)**  **and Write Back(WB) stages.**

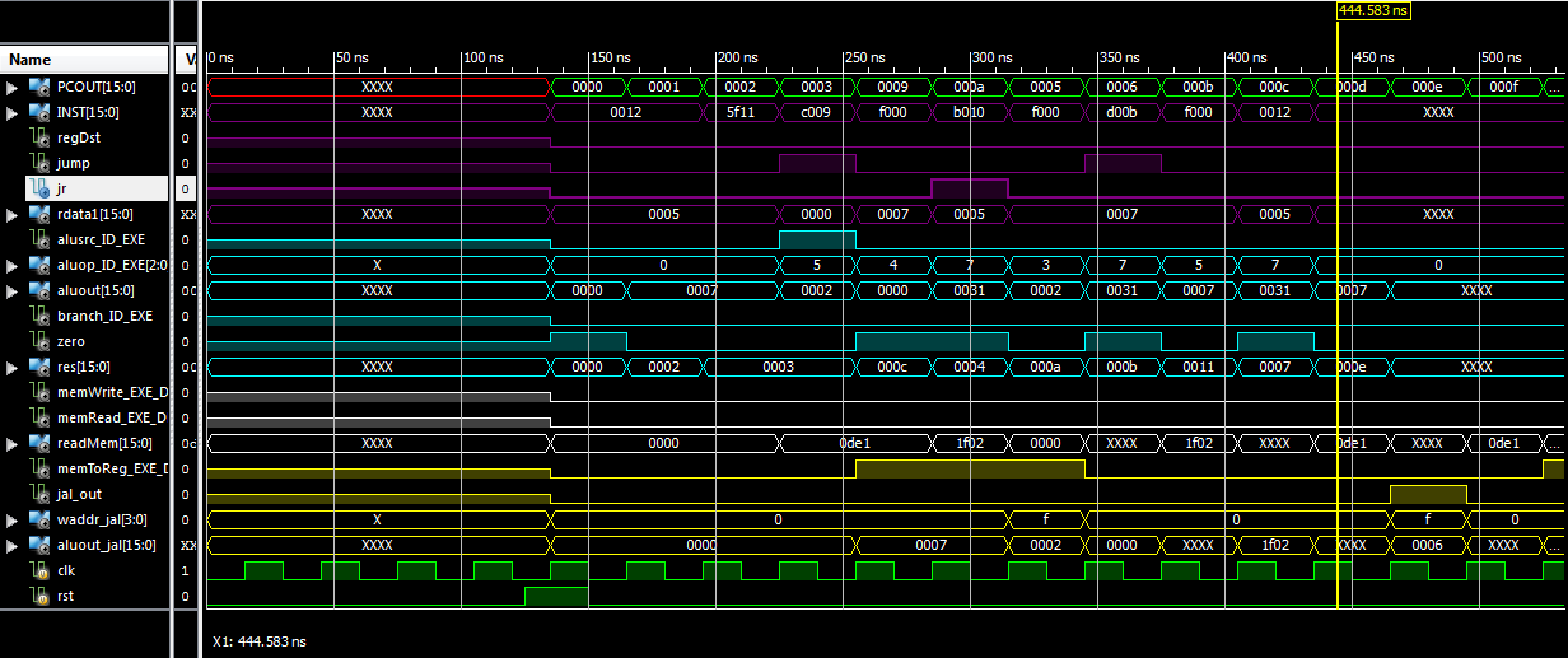
****

Figure 2.2: JAL instructions

|  |  |
| --- | --- |
| The 5-stage pipeline flow for a JAL instruction is highlighted in the test bench above. JAL offset means that the PC<-IMem[offset] is to be updated and the value of the previous PC+1 is to be stored in $rf of the Register File. | *Figure 2.2a Reg15 stores 6* |
| The **Fetch(F)** stage brings the instruction from the memory location 0x0005 |
| **Decode(D)** stage determines the control signals such as *Wen, AluOp, jump, jr, jal*. For the JAL instruction when passed to control unit both the JAL and JUMP signals are high. In the decode stage, Address is concatenated and the next instruction is PC[15:12]+I[11:0]. |
| After concatenating the value is passed to the multiplexer and as the Jump signal is enabled, it is passed on to the Program Counter as the next instruction to be executed. I[11:0] denotes the address of the instruction to be jumped to specified by the offset of the instruction.  **The JAL instruction does not make use of the Execute(E) and Data Memory(DM) stages.**  **Write Back(WB) stage:** In this stage, JAL is HIGH as a result the PC+1 is the Wdata instead of aluout, and the it is written in $rf of the Register File. The Wen is also HIGH in this stage. Figure 2.2a shows how the register files were updated after the execution of JAL in Figure 2.2 | |

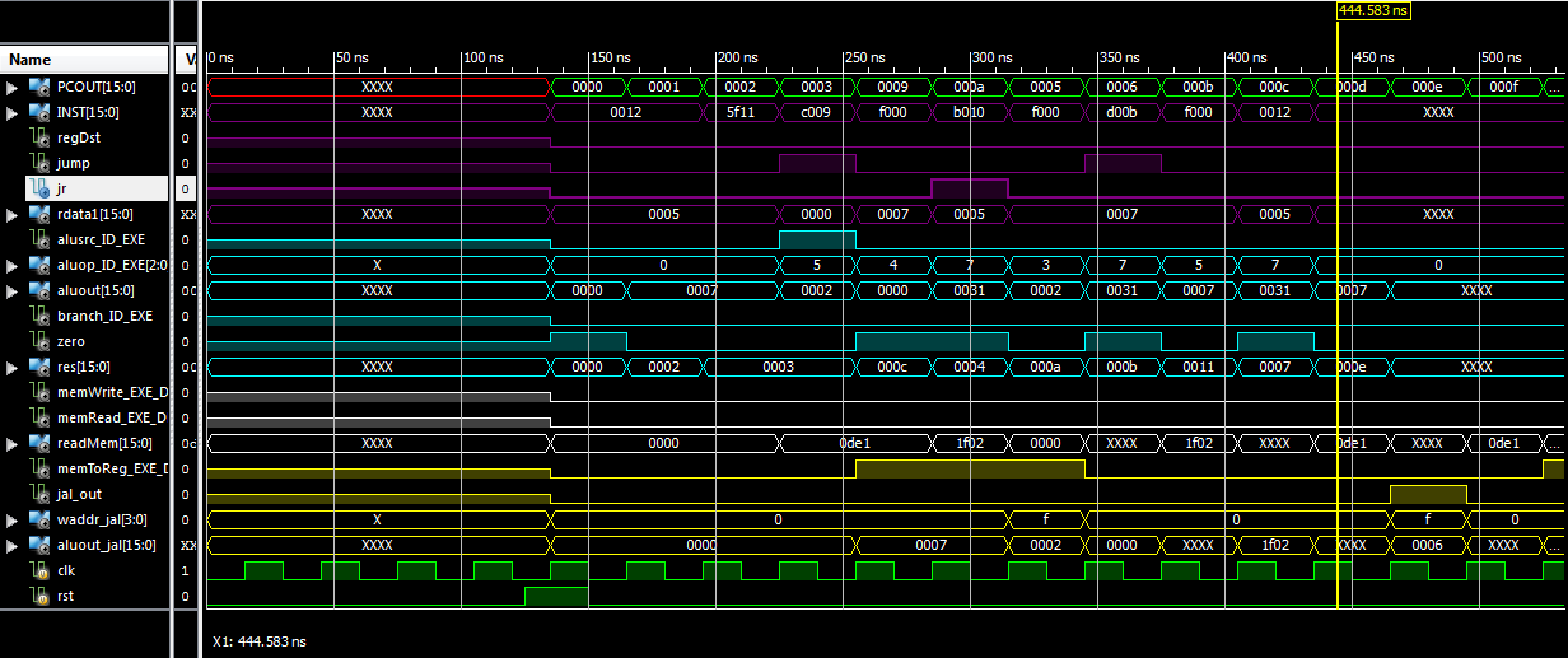


Figure 2.3: JR instructions

## Working:

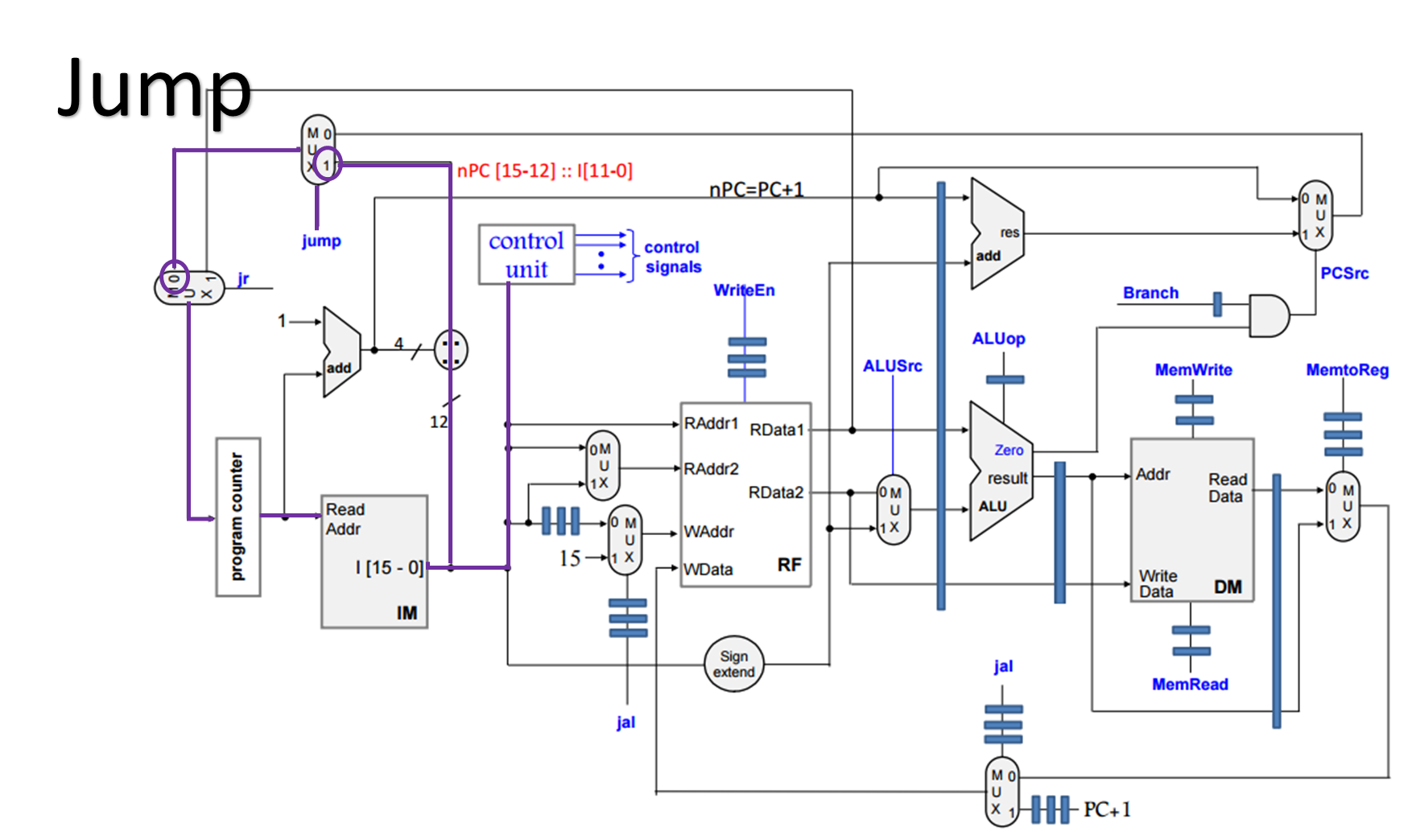
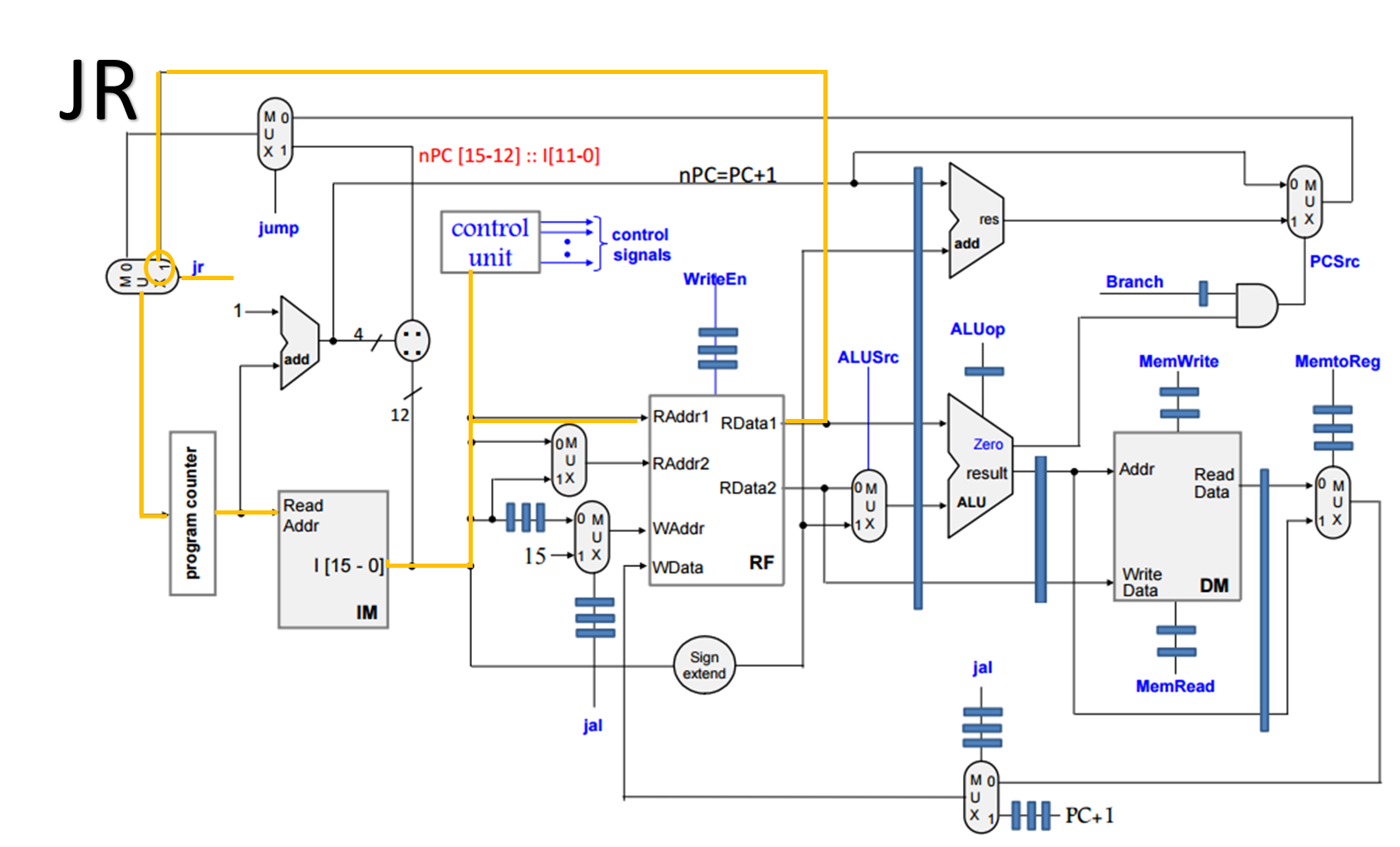
The 5-stage pipeline flow for a jr instruction is highlighted in the test bench above. Jr $rs means that the PC<-$rs is to be updated.

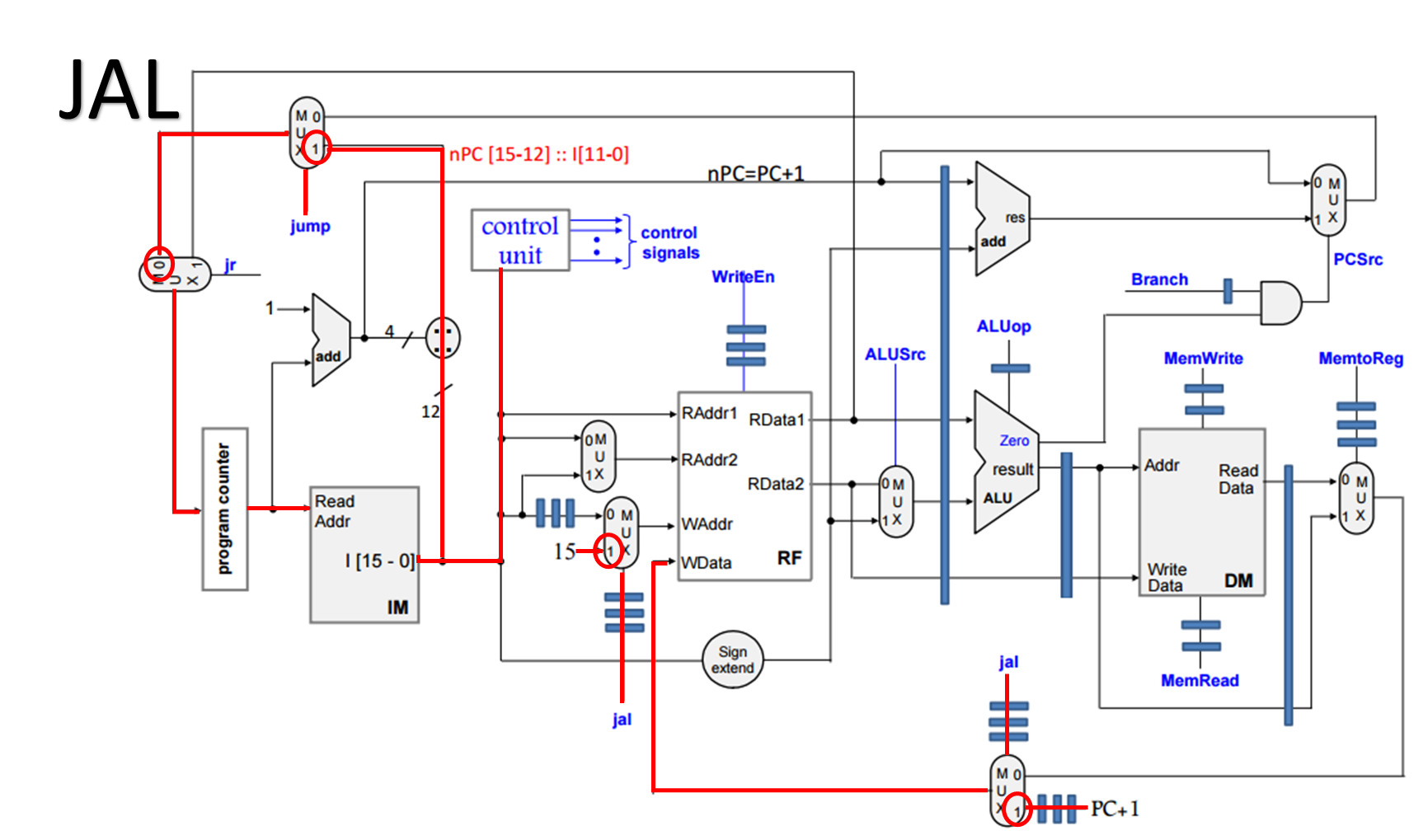
The **Fetch(F)** stage brings the instruction from the memory location 0x0009

**Decode(D)** stage determines the control signals such as  *jr.* The value $rs e.g. Reg1 in Figure 2.3 is passed to the RAddr1 of the Register File. The corresponding data is read out of the Register File and its value is passed into a multiplexer, which is enabled by a jr signal from the Control Unit. The corresponding address read is then updated into the Program Counter.

**The JUMP instruction does not make use of the Execute(E), Data Memory(DM)**  **and Write Back(WB) stages.**

The following figures 2.4a, 2.4b and 2.4c provide a summary of the path taken by JUMP, JR and JAL.





# Synthesis Report for Part-2.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| CPU | BITWIDTH | No of LUT slices | No of registers | minimum period in ns |
| Five-stage Pipeline | 16 | 827 | 518 | 6.471ns |

PART 3

# Reduction in CPI

Our interpretation of the given code

|  |  |  |
| --- | --- | --- |
| for (i=6; i<=65; i++){  a[i]=b[i]+a[i-6];  } |  | for(i=0; i<=59; i++){  a[i+6]=b[i+6]+a[i];  } |

## Without unrolling

**(i)** **MIPS-like assembly code**

|  |  |  |
| --- | --- | --- |
| //The following register assignments are made  //The first 4 elements of data memory contain 4, 1, 76 and 64 respectively to initialize registers  //R0 = 4 : address of first element of array a  //R1 = 1 : to increment counters  //R2 = 76 : address of the 7th element of array b whose first element is at position 66  //R3 = 64 : To check for end of loop | | |
|  | LW R0, 0(R7) | //all registers initially store 0 so any register can be used |
|  | LW R1, 1(R7) |  |
|  | LW R2, 2(R7) |  |
|  | LW R3, 3(R7) |  |
| LOOP: | LW R4, 0(R0) | //load a[i] into R4 where i varies from 0 to 60 |
|  | LW R5, 0(R2) | // load a[i+6] into R5 |
|  | NOP | // no operations added due to RAW dependency in R4 and R5 |
|  | NOP |  |
|  | NOP |  |
|  | ADD R4, R4, R5 | //storing addition in temporary register |
|  | NOP |  |
|  | NOP |  |
|  | NOP | // no operations added to handle RAW dependency of R6 |
|  | SW R4, 6(R0) | //storing the result of addition in a[i+6] |
|  | ADD R0, R0, R1 | //incrementing counter for array a |
|  | ADD R2, R2, R1 | //incrementing counter for array b |
|  | NOP |  |
|  | NOP |  |
|  | NOP |  |
|  | BEQ R0, R3, EXIT | //checking if counter stored in R0 is equal to 64 |
|  | NOP |  |
|  | NOP | //handling control hazards |
|  | J LOOP | //jump to beginning of loop |
|  | NOP | //handling control hazards |
| EXIT: |  |  |

**(ii) Machine Language**

|  |  |  |
| --- | --- | --- |
| **Instr No.** | **Instr** | **Comments** |
| //Initialising registers with constant values | | |
| 0000 | 8043 | //Reg[0]=4 (4 stored in mem location 0x0003)start address of a[i] |
| 0001 | 8241 | //Reg[2]=76 (76 stored in mem location 0x0001)address of 7th element of b[i] |
| 0002 | 8342 | //Reg[3]=64 (64 stored in mem location 0x0002) |
| 0003 | 8140 | //Reg[1]=1 (1 stored in mem location 0x0000) |
| //Loop starts here | | |
| 0004 | 8400 | //load address of a[i] to R4 |
| 0005 | 8520 | // load address of b[i+6] to R5 |
| 0006 | F000 | nop |
| 0007 | F000 | nop |
| 0008 | F000 | nop |
| 0009 | 0445 | // reg4=a[i]+b[i+6] |
| 000A | F000 | nop |
| 000B | F000 | nop |
| 000C | F000 | nop |
| 000D | 9406 | // a[i+6]=reg4 |
| 000E | 0001 | //Increment R0 i++ |
| 000F | 0221 | //Increment R2 |
| 0010 | f000 | nop |
| 0011 | f000 | nop |
| 0012 | f000 | nop |
| 0013 | A033 | // branch to 3 lines after if R0=R3 |
| 0014 | F000 | nop |
| 0015 | F000 | nop |
| 0016 | C004 | //Jump to 0x0004 |
| 0017 | F000 | //End of loop |

**(iii) Number of clock cycles and execution time**

|  |  |
| --- | --- |
| Time period of 1 clock | = 30ns |
| Total Execution Time | =36,374.400ns |
| Reset time | =135ns |
| **Actual total execution time** | =36,374.400-135  =36,239.400 ns |
|  |  |
|  |  |
|  |  |
| Steady state CPI |  |

## Maximal loop unrolling

1. **Unrolling by a factor of 6 and reordering, MIPS-like assembly code**

|  |  |  |
| --- | --- | --- |
| //The following register assignments are made  //The first 4 elements of data memory contain 4, 1, 76 and 64 respectively to initialize registers  //R0 = 4 : address of first element of array a  **//R1 = 6** : to increment counters  //R2 = 76 : address of the 7th element of array b whose first element is at position 66  //R3 = 64 : To check for end of loop | | |
|  | LW R0, 0(R7) | //all registers initially store 0 so any register can be used |
|  | LW R1, 1(R7) |  |
|  | LW R2, 2(R7) |  |
|  | LW R3, 3(R7) |  |
| LOOP: | LW R4, 0(R0) | // load a[i] into R4 where i varies from 0 to 60 |
|  | LW R5, 0(R2) | // load a[i+6] into R5 |
|  | LW R6, 1(R0) | // load address of a[i+1] to R6 |
|  | LW R7, 1(R2) | // load address of b[i+7] to R7 |
|  | LW R8, 2(R0) | // load address of a[i+2] to R8 |
|  | LW R9, 2(R2) | // load address of b[i+8] to R9 |
|  | LW R10, 3(R0) | // load address of a[i+3] to R10 |
|  | LW R11, 3(R2) | // load address of b[i+9] to R11 |
|  | LW R12, 4(R0) | // load address of a[i+4] to R12 |
|  | LW R13, 4(R2) | // load address of b[i+10] to R13 |
|  | LW R14, 5(R0) | // load address of a[i+5] to R14 |
|  | LW R15, 5(R2) | // load address of b[i+11] to R15 |
|  | ADD R4, R4, R5 |  |
|  | ADD R6, R6, R7  ADD R8, R8, R9  ADD R10,R10,R11  ADD R12,R12,R13 |  |
|  | ADD R14,R14,R15 | //storing addition in temporary register |
|  | SW R4, 6(R0) |  |
|  | SW R6, 7(R0) |  |
|  | SW R8, 8(R0) |  |
|  | SW R10, 9(R0) |  |
|  | SW R12, 10(R0) |  |
|  | SW R14, 11(R0) | // storing result of addition to memory |
|  | ADD R0, R0, R1 | //incrementing counter for array a |
|  | ADD R2, R2, R1 | //incrementing counter for array b |
|  | NOP |  |
|  | NOP | //Stall-RAW dependency between r0 |
|  | NOP |  |
|  | BEQ R0,R3,EXIT | //checking if counter stored in R0 is equal to 64 |
|  | NOP |  |
|  | NOP | //handling control hazards |
|  | J LOOP | //jump to beginning of loop |
|  | NOP | //handling control hazards |
| EXIT: |  |  |

1. **Assembly code**

|  |  |  |
| --- | --- | --- |
| **Instr No.** | **Instr** | **Comments** |
| //Initialising registers with constant values | | |
| 0000 | 8043 | //Reg[0]=4 (4 stored in mem location 0x0003)start address of a[i] |
| 0001 | 8241 | //Reg[2]=76 (76 stored in mem location 0x0001)address of 7th element of b[i] |
| 0002 | 8342 | //Reg[3]=64 (64 stored in mem location 0x0002) |
| 0003 | 8140 | //Reg[1]=1 (1 stored in mem location 0x0000) |
| //Loop starts here | | |
| 0004 | 8400 | // load addresses of 6 elements of a and b in R4 to R15 |
| 0005 | 8520 |  |
| 0006 | 8601 |  |
| 0007 | 8721 |  |
| 0008 | 8802 |  |
| 0009 | 8922 |  |
| 000A | 8a03 |  |
| 000B | 8b23 |  |
| 000C | 8c04 |  |
| 000D | 8d24 |  |
| 000E | 8e05 |  |
| 000F | 8f25 |  |
| 0010 | 0445 | // adding the a[i] and b[i+6] for 6 elements |
| 0011 | 0667 |  |
| 0012 | 0889 |  |
| 0013 | 0aab |  |
| 0014 | 0ccd |  |
| 0015 | 0eef |  |
| 0016 | 9406 | // storing result of addition to a[i+6] |
| 0017 | 9607 |  |
| 0018 | 9808 |  |
| 0019 | 9a09 |  |
| 001A | 9c0a |  |
| 001B | 9e0b |  |
| 001C | 0001 | //Increment R0 i++ |
| 001D | 0221 | //Increment R2 |
| 001E | f000 | nop |
| 001F | f000 | nop |
| 0020 | f000 | nop |
| 0021 | A033 | // branch to 3 lines after if R0=R3 |
| 0022 | f000 | nop |
| 0023 | f000 | nop |
| 0024 | c004 | //Jump to 0x0004 |
| 0025 | f000 | //End of loop |

1. **No. of clock cycles and CPI**

|  |  |
| --- | --- |
| Time period of 1 clock | = 30ns |
| Total Execution Time | =10,514.800ns |
| Reset time | =135ns |
| **Actual total execution time** | =10,514.80-135 ns  =10,441.800ns |
|  |  |
|  |  |
|  |  |
| Steady state CPI |  |

## Explanation

By utilizing more resources, unrolling the loop to add 6 values in one iteration rather than 1 is possible which removes the RAW dependency of the registers as the instructions can be reordered. This decrease the number of ‘no-operations’ required per loop thus decreasing the steady state CPI from 2 to 1.2

Thus, Speed Up achieved due to loop unrolling

ns