Verslag

Titel: *Datapath*

Dit verslag werd opgesteld door:

- **Naam:** *Caluwé Jonas*
  Studentennummer: *s0210051*
  Email adres: *jonas.caluwe@student.uantwerpen.be*
-
- **Naam:** *Van pellicom Gilles*
  Studentennummer: s0211449
  Email adres: *gilles.vanpelicom@student.uantwerpen.be*
-

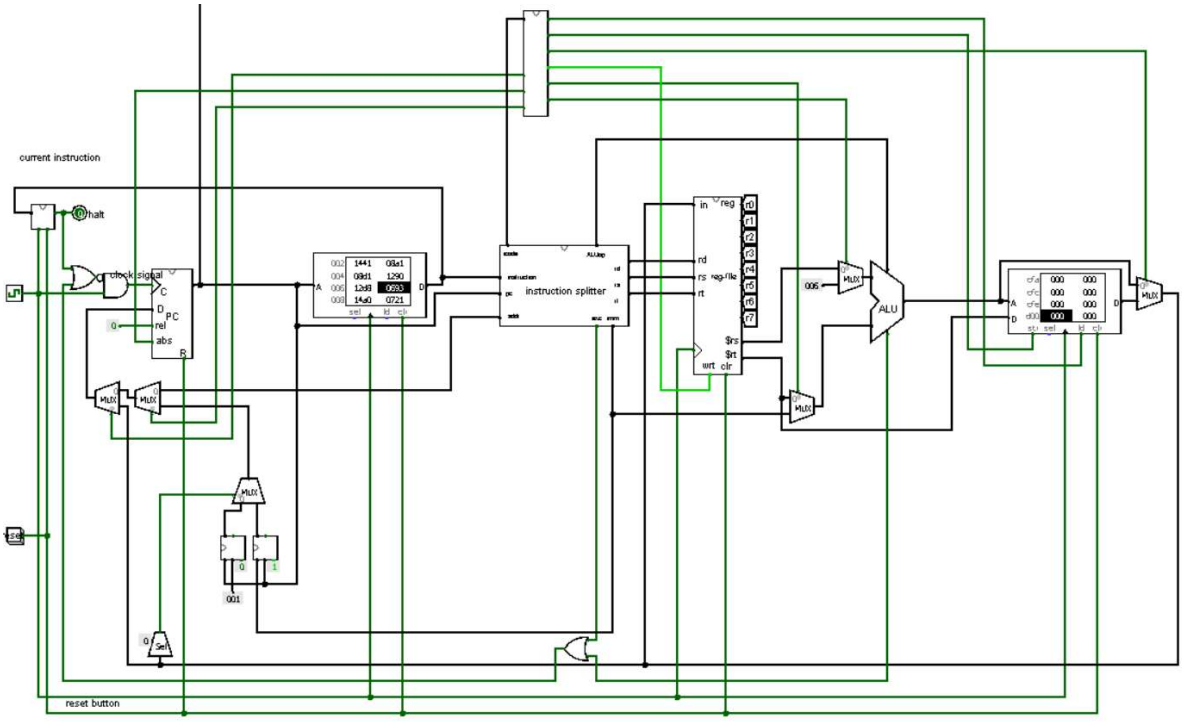
Aantal man-uren besteed: *40 uur*
Moeilijkheidsgraad: *7* /10 (1 is heel makkelijk, 10 is heel moeilijk)

Inhoud van de oplossing

De oplossing bestaat uit de volgende bestanden (geef alle bestanden op):

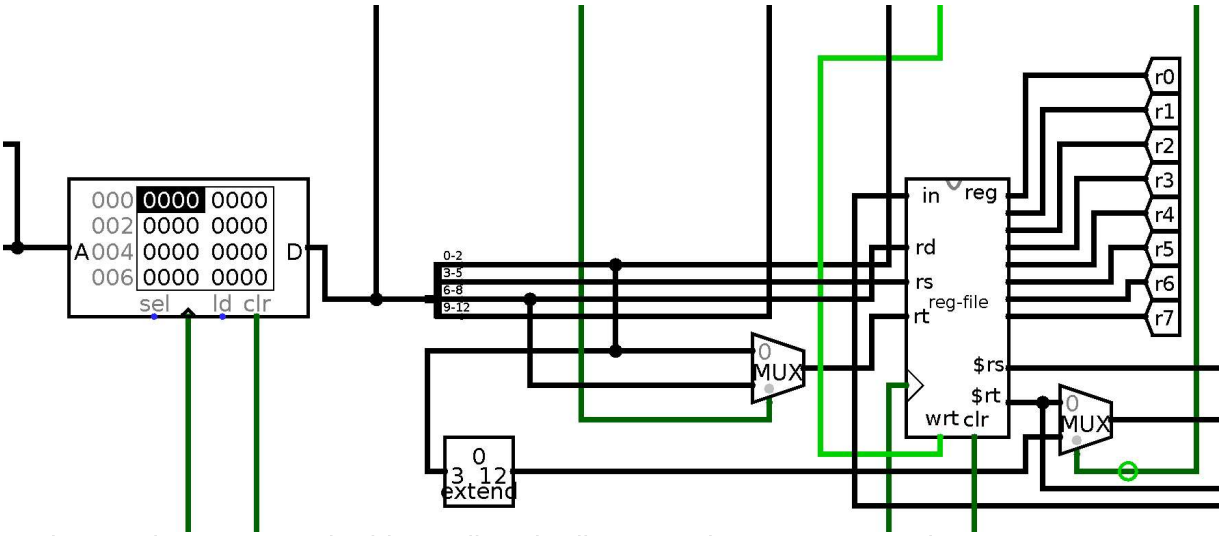
- bestand1.ext: toelichting van bestand1.ext
- bestand2.ext: toelichting van bestand2.ext
- ...

Verslag



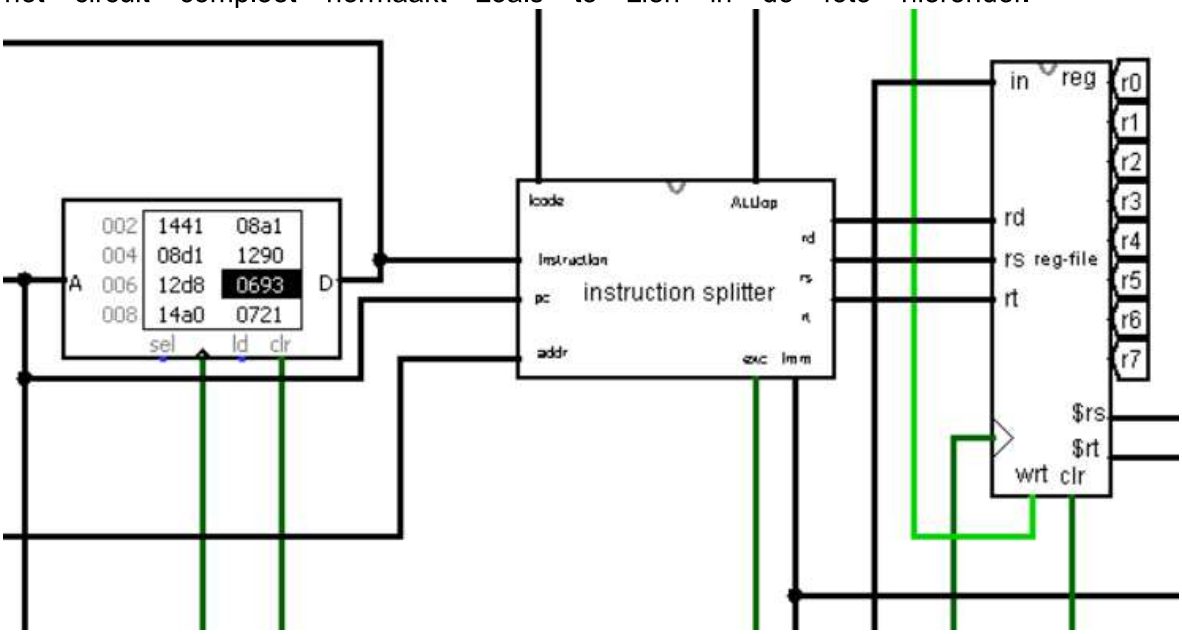
In dit verslag zullen wij de verbeteringen tegenover de vorige versie uitleggen. Ook gaan we de nieuwe instructies (ORI, LUI, BRNZ, JR, J en JAL) uitleggen. Dit zal niet de meest samenhangende uitleg zijn aangezien deze functionaliteit verdeeld is over meerdere plaatsen in het circuit.

1. The instruction splitter

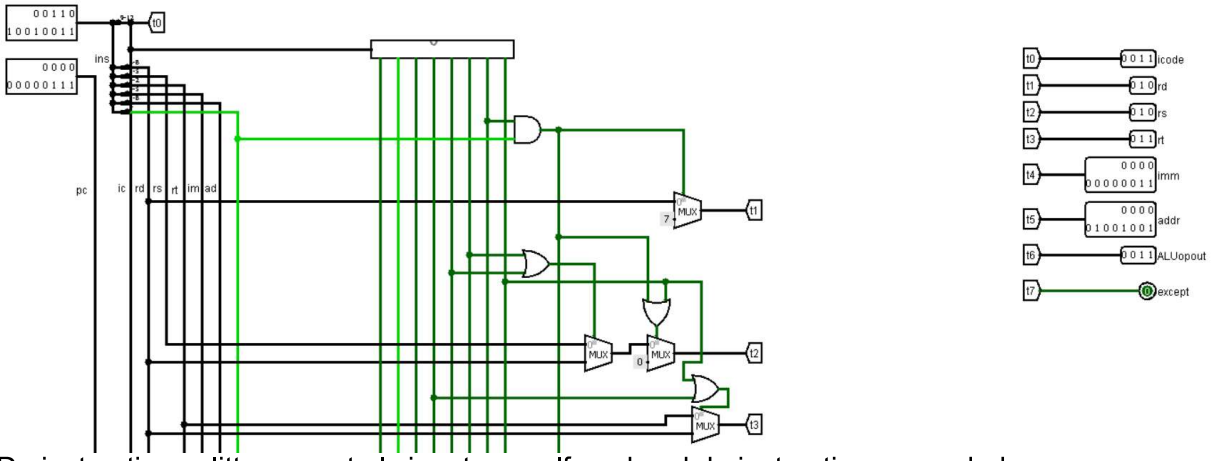


In het vorige verslag hadden wij gebruik gemaakt van een splitter en multiplexers om de juiste input voor de juiste instructie te voorzien. Deze aanpak was niet erg elegant en totaal niet scalable dus wij hebben diit deel van

het circuit compleet hermaakt zoals te zien in de foto hieronder.

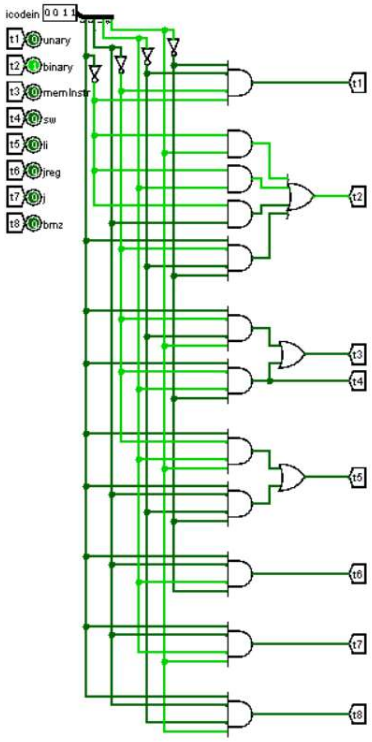


Zoals u kan zien is dit een veel propere en praktische oplossing. Op de volgende foto ziet u een deel van de werking van deze module.

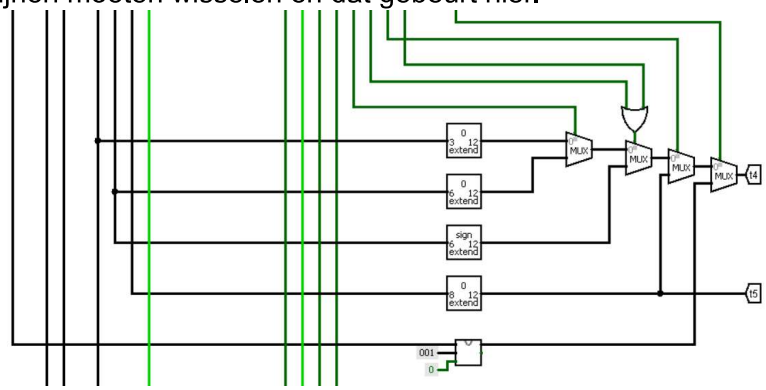


De instruction splitter neemt als input vanzelfsprekend de instructie maar ook de program counter. De instructie wordt verdeeld in alle nodige delen voor elke instructie beschreven in de instructieset. Dit is de bus van lijnen aan de linkerkant van de foto. Natuurlijk moeten we weten welke lijnen te gebruiken bij welke instructie. Dit is de job van de instructie decoder (de module te zien verbonden aan de groene lijnen en in detail op de foto links van deze tekst) Deze decoder verdeelt de gelijke instructies zodat deze op dezelfde manier worden behandeld. Zo zullen de input lijnen van een ADD en SUB niet gelijk zijn aan Jump of JAL aangezien de verdeling bij ADD en sub 4/3/3/3 is en 4/7/1 bij Jump en JAL. Op de bovenste foto kan u in het midden de logica voor rd, rs en

rt terugvinden. Soms kan het dat voor een bepaalde instructie twee van deze

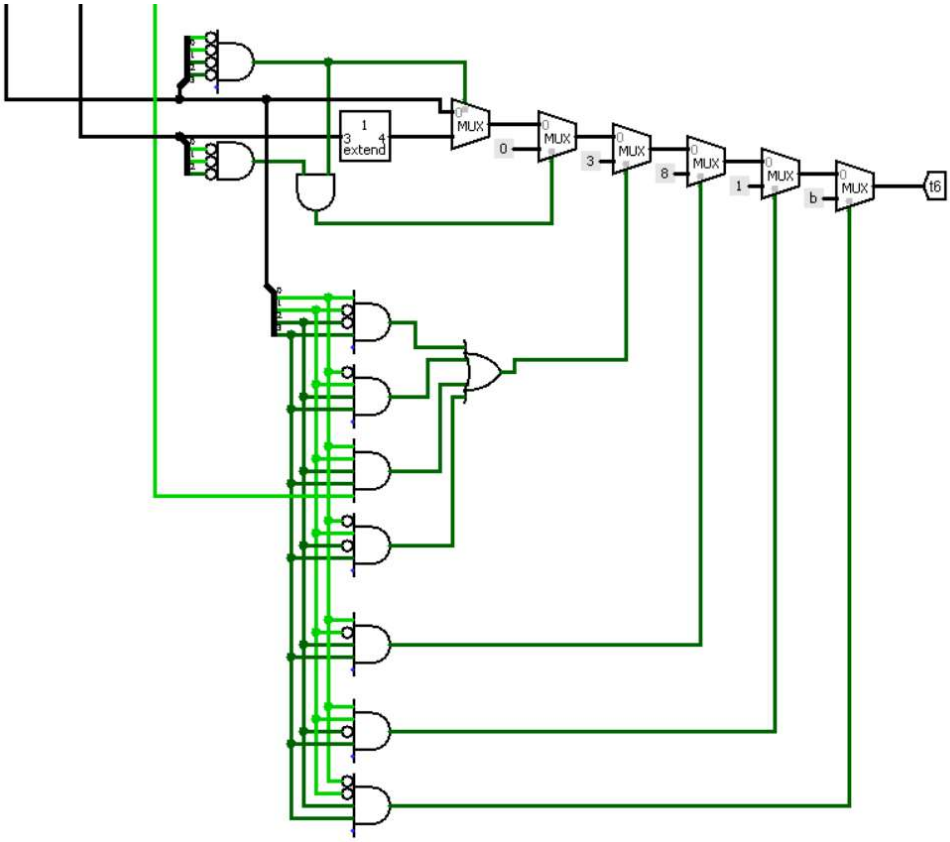


lijnen moeten wisselen en dat gebeurt hier.



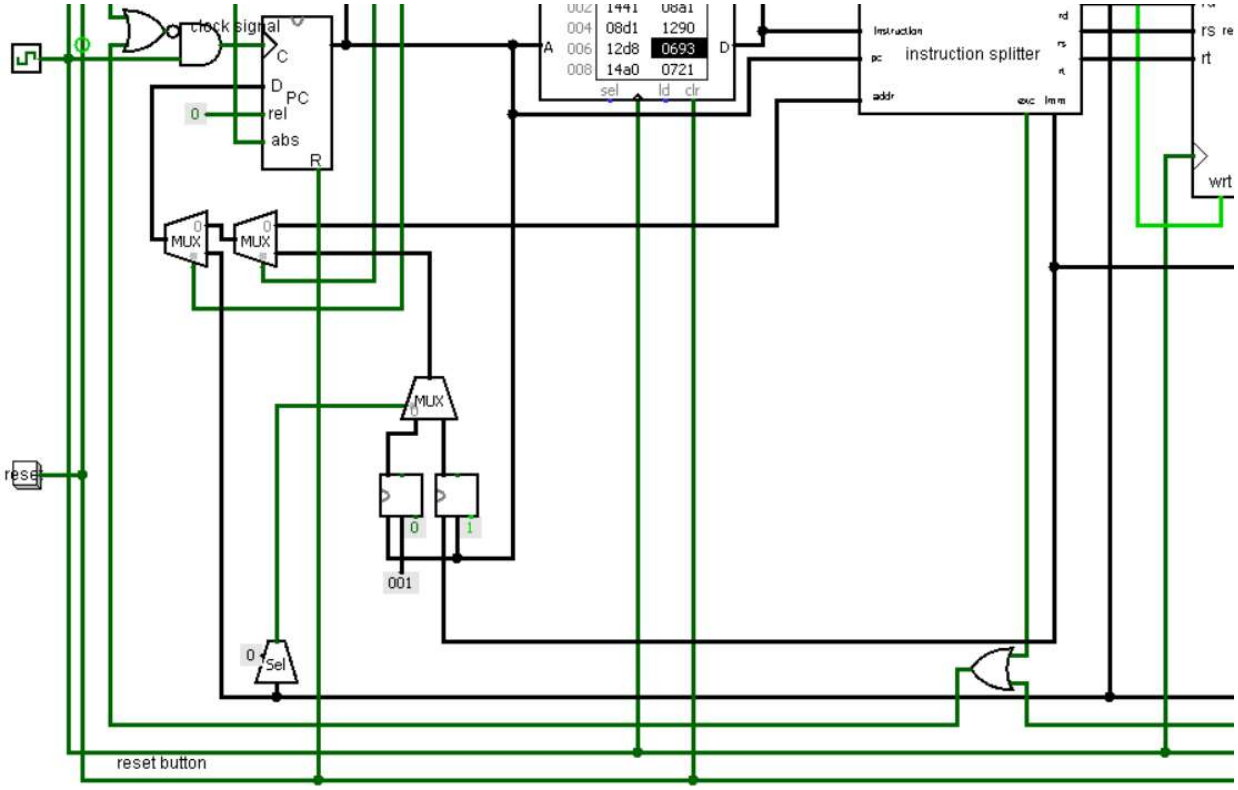
Rechts hierboven het tweede deel van de splitter. Dit deel is verantwoordelijk voor de immediate en het adres. Hier worden de lijnen ge-extend of ge-signextend naargelangen de instructie. De immediate kan simpelweg de immediate in de instructie zijn of zoals u kan zien ook rt, program counter of de adres lijn. De adres output is altijd de extended versie van het adres in de instructie.

2. ALU decoder



De ALU-decoder (Zoals te zien hierboven) zit in dezelfde module als de instruction splitter, aangezien we hier al gesplitst hebben in al de nodige buslijnen. De decoder kan je bekijken als een switch case. De default output is NOOP en zal dus als output 1111 hebben als geen van de condities voldaan zijn. De condities bestaan uit bubbled and gates om een bepaalde opcode te herkennen. Als een opcode herkend wordt gaat de overeenkomende multiplexer aan en zal de juiste opcode worden doorgegeven aan de ALU. Wat hier veranderd is tegenover de vorige iteratie, is dat er nu ook rekening zal worden gehouden met JAL. J en JAL hebben beide 1111 als opcode dus we hebben ook de LSB nodig om deze te kunnen onderscheiden. Dit gebeurt in de AND-gate met 5 poorten.

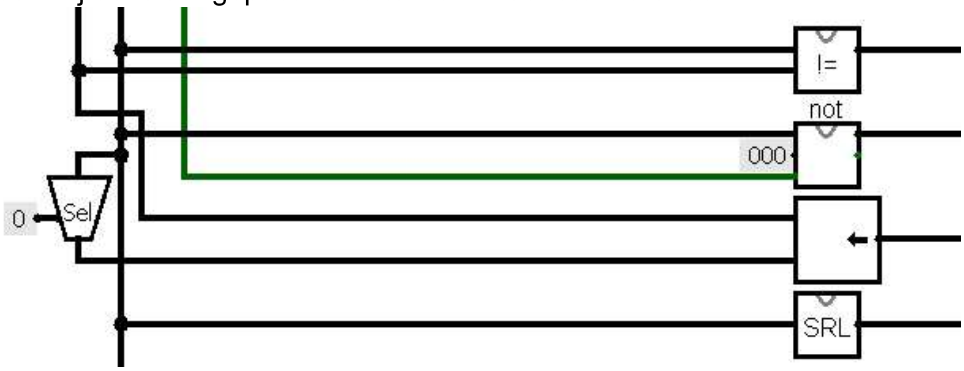
3. Jumps



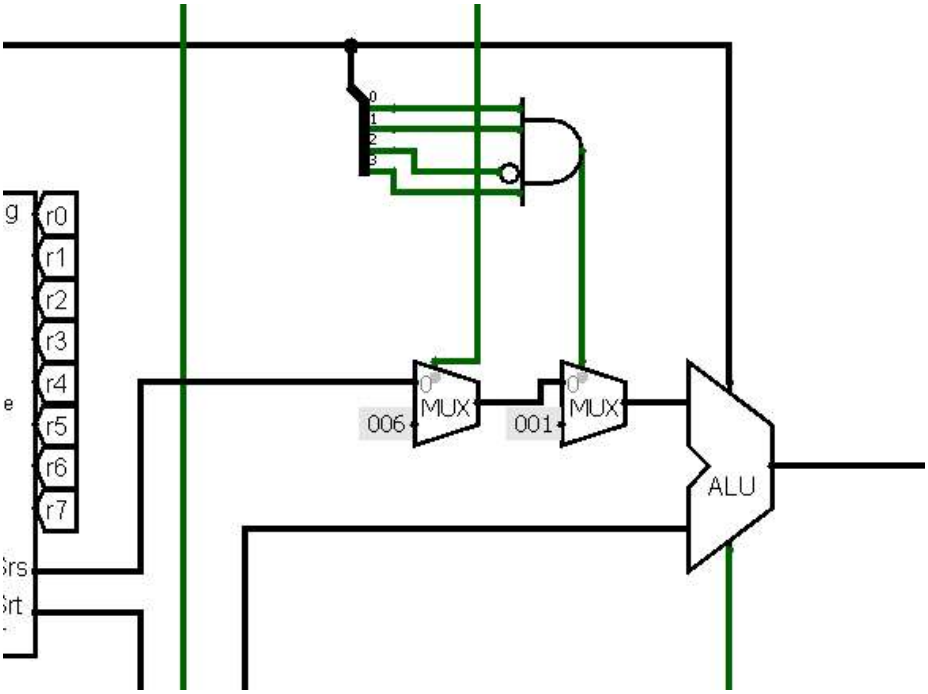
Zoals u op de éérste foto in dit verslag kon zien is onze datastructuur chronologisch opgesteld. Het begint bij de clock, daarna de program counter, program memory, register, alu en dan het memory. Natuurlijk is het de bedoeling om met berekende resultaten opnieuw te werken dus we de structuur zou meer op een cirkel moeten lijken. Wat er berekend is uit de alu of wat er opgevraagd is uit het geheugen komt terug naar voor om dan naar de program counter te gaan als bij een jump instructie,of naar het register voor andere instructies dat opslaan in het register. Voor de jumps hebben we een multiplexer verbonden aan een bitselector dat de laatste bit selecteert om te differentieren tussen een J en een JAL. Ook is er een multiplexer voor een immediate adres of niet.

4. Load upper immediate

Deze functie gebruiken we om gebruik te maken van de volledige grote in een register. Aangezien een instructie maar 13 bits lang is is er maar 6-bit plaats per instructie voor een waarde. Om dit op te lossen gaan we de waarde 6 plaatsen naar links shiften en dan de instructie ORI gebruiken om de nu vrijgekomen eerste 6 bits te vullen. Aangezien onze alu maximaal 1 plaats per keer kon shiften hebben wij deze aangepast zoals te zien in de foto hieronder.



SLL is vanaf nu geen unary operation maar een binary operation waarbij input A de shift hoeveelheid is en B het getal dat moet worden geshift.



Natuurlijk moeten wij nu ook een getal als input kunnen geven. Dit doen wij door middle van twee multiplexers waarbij er zes zal worden doorgegeven als er een LUI instructie aankomt en één bij een SLL instructie.

5. Or immediate

Dit is veruit de meest simple instructie in dit verslag. We zetten voor deze opcode de immediate multiplexer aan in de controller en opcode 0001 (OR) in de alu controller. De infrastructuur om de uitkomst op te slaan in het register bestaat al.

6. Branch not zero

Net zoals de vorige instructie is dit een simpele kwestie van controllers aan te passen. We laten de alu de NEQ (1000) operatie gebruiken en vergelijken input A, de waarde, met B, nul uit het nulregister.