Verslag

Titel: *Verslag 5*

Dit verslag werd opgesteld door:

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

Aantal man-uren besteed: *13 uur*
Moeilijkheidsgraad: *6* /10 (1 is heel makkelijk, 10 is heel moeilijk)

Inhoud van de oplossing

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

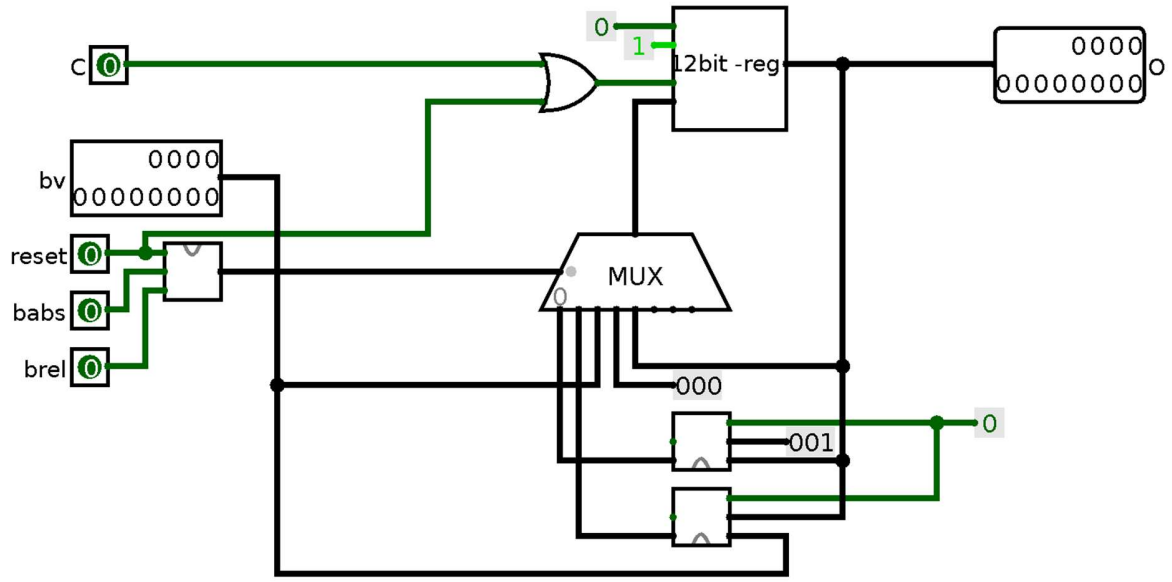
- bestand1.ext: toelichting van bestand1.ext
- bestand2.ext: toelichting van bestand2.ext
- ...

Verslag

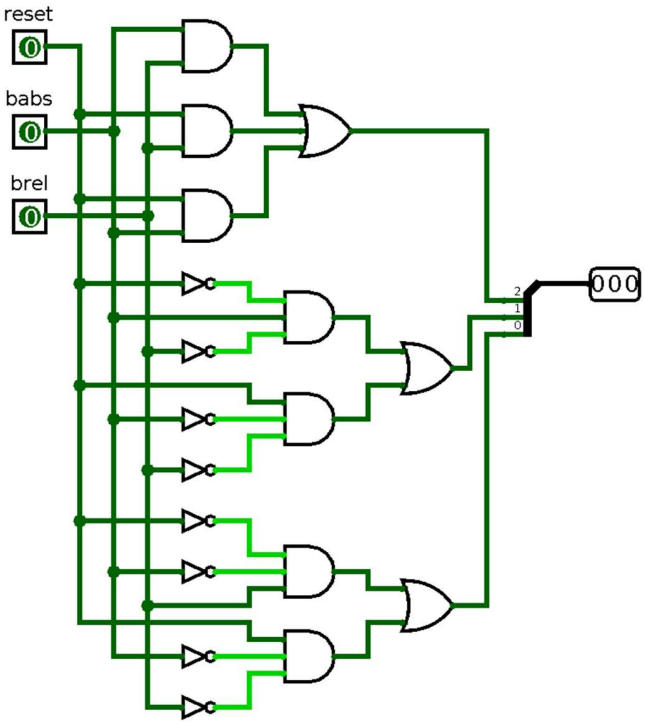
I dit verslag zullen we meer uitleg geven over onze keuzes en implementaties van bepaalde “features”. We zullen beetje bij beetje opbouwen, startende met onze program counter en we eindigen met hoe dit alles samenhangt en gecontroleerd wordt door het hoofd control circuit.

1. The program counter

De program counter start uiteraard met de clock, wanneer het geen speciale input krijgt, dan telt deze één op bij het huidig bewaarde getal. Dit getal wordt opgeslagen in een register dat altijd zal schrijven. Er zijn echter vier andere inputs die het gedrag van de program counter kunnen aanpassen. We hebben een reset, branch relative, branch absolute en een branch data. Omdat de program counter op basis van deze inputs een andere waarden zal aannemen, leek het ons een goed idee om deze verschillende gevallen te multiplexen. We stonden hiermee echter voor één probleem, we hebben drie aparte inputs, maar de multiplexer verwacht één getal. Om dit op



te lossen hebben we een encoder gemaakt die deze drie waarden omzet naar één unieke waarde.

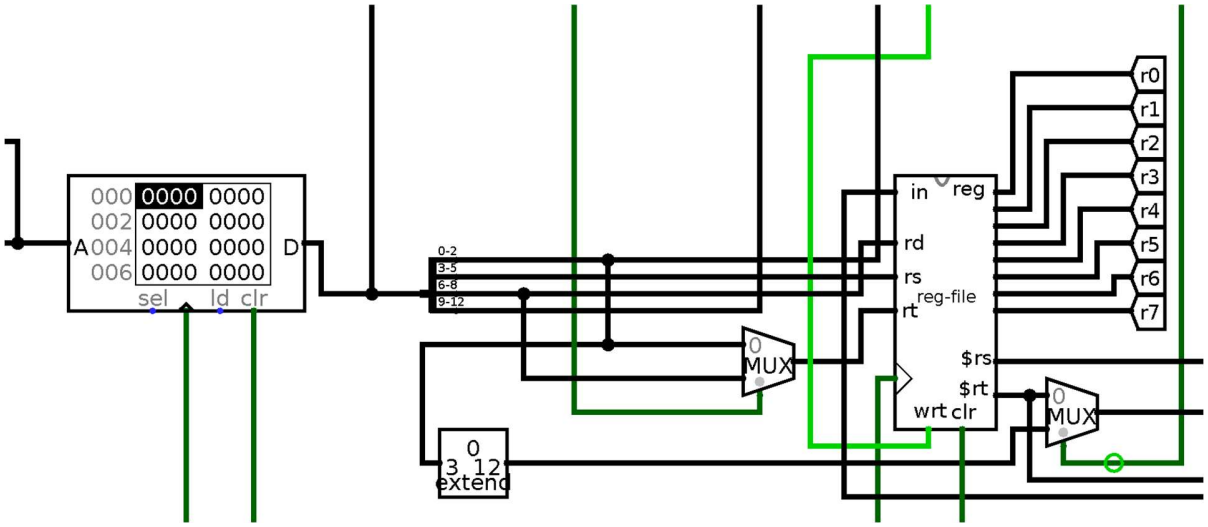


Deze encoder doet het volgende, wanneer enkel de reset aanstaat, dan geeft hij de output 011. Wanneer enkel de branch absolute aanstaat, dan geeft hij 010. Enkel branch relative geeft 001. Zoals u ziet zijn dit maar vier gevallen (000 staat voor geen inputs en dus het standaard geval) en was dit dus perfect te coderen in twee bits. We hebben echter gekozen om een extra output toe te voegen, het geval voor een illegale input (100). Wanneer meerdere inputs tegelijk aanstaan, dan is de verwachte output niet meer éénduidig en dus hebben we een vijfde output die staat voor doe niets en slaag de huidige waarde terug op. De reset input doet het volgende: de encode zet de multiplexor op waarde 011 en dus wordt het twaalf bit getal 0 doorgelaten. Omdat het register schrijft op “falling edge”, hebben we de reset verbonden aan de clock zodat we de clock kunnen negeren in het geval van een reset. Wanneer branch absolute aanstaat,

dan schrijven we wat er in branch value staat direct naar het register ongeacht wat de huidige waarde is. Wanneer branch relative aanstaat, dan tellen we de branch value (geïnterpreteerd als two's complement) op bij de huidige waarde.

De program counter stelt een adres voor in binair. Dit is een adres in het instruction memory.

2. Instruction memory and register-file



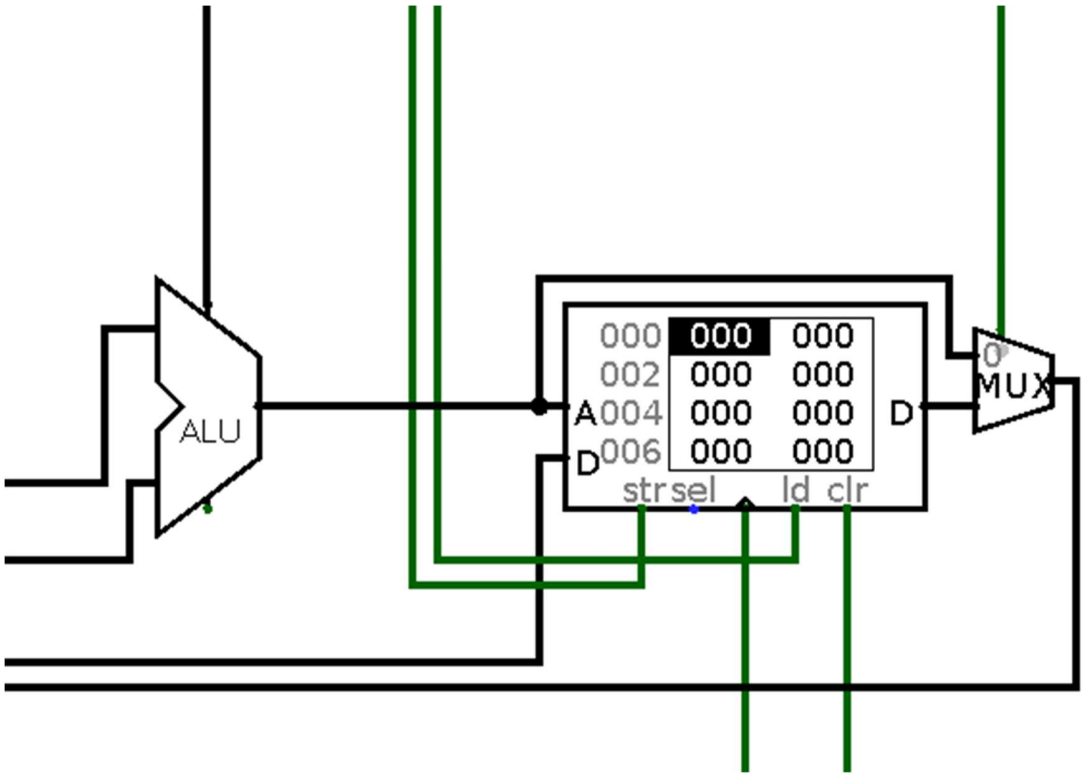
Het instruction memory bevat de huidig uit te voeren instructie (13 bits) volgens de program counter. De instructie is gecodeerd als volgt: de eerste vier bits bevatten om wat voor soort instructie het gaat: unary- (0000), binary- (0001-1000), memory-instruction (1001, 1010).

Een unary instruction bevat een register om het resultaat in op te slaan, een register om een operatie op uit te voeren en een opcode. Omdat we maar werken met 13 bits is deze opcode drie bits lang, onze ALU verwacht echter vier bits als opcode (zie project ALU). We hebben de opcodes van onze ALU zodanig gekozen dat wanneer het gaat om een unary operation, we enkel een "1" moeten toevoegen vooraan de opcode (behalve de zero operatie: zie ALU-control logic). De register file bevat acht registers (zie project memory) en is dus te adresseren met een drie bit getal.

In het geval van een binary operation, bevat de instructie ook een adres om het resultaat in op te slaan en twee registers om de operatie op uit voeren. Deze instructie type bits, dienen eveneens als instructies voor de ALU.
Ten slotte hebben we de twee geheugen instructies. De eerste ("load word") bevat een register om het resultaat in op te slagen, een register dat een adres in het data geheugen moet bevatten en een immediate. Wanneer we ons programma starten, hebben we geen manier om arbitraire getallen en adressen in onze register file op te slaan. Hierdoor kan er enkel een adres aanwezig zijn door unary of binary operaties uit te voeren op het getal 0 en verder. Echter verwacht deze instructie dat er een adres is opgeslagen in deze register file, dus maken we gebruik van een immediate als offset. Wanneer de register file leeg is, kunnen we toch negen verschillende acht

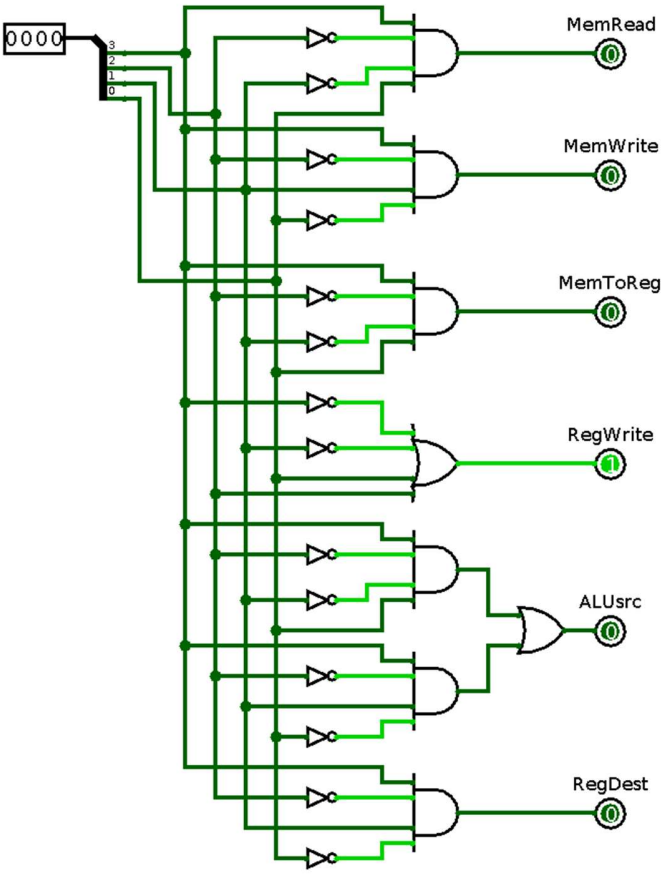
verschillende adressen aanspreken in het datageheugen, door een offset van drie bits op te tellen bij het adres nul.
De tweede instructie, de “store word” instructie is een beetje anders, normaal gezien staat de tweede sectie bits voor het register om het resultaat in op te slaan. In dit geval staat voor een register om een getal uit te lezen. We moeten dit dus i.p.v. ‘rd’, met ‘rs’ of ‘rt’ verbinden. Omdat ‘rs’ al bezet in deze gecodeerde instructie, kiezen we dus voor ‘rt’. ‘Rt’ werd echter in de andere instructies gebruikt als register om een operatie op uit te voeren. We zouden kunnen kiezen om de ALU op “no operation” te zetten, maar omdat we hier een immediate bij het adres moeten optellen, is de ALU dus al bezet, we kiezen ervoor om deze ‘rt’ output voor de multiplexer te “hijacken” en deze rechtstreeks met het data geheugen te verbinden.

3. Data memory



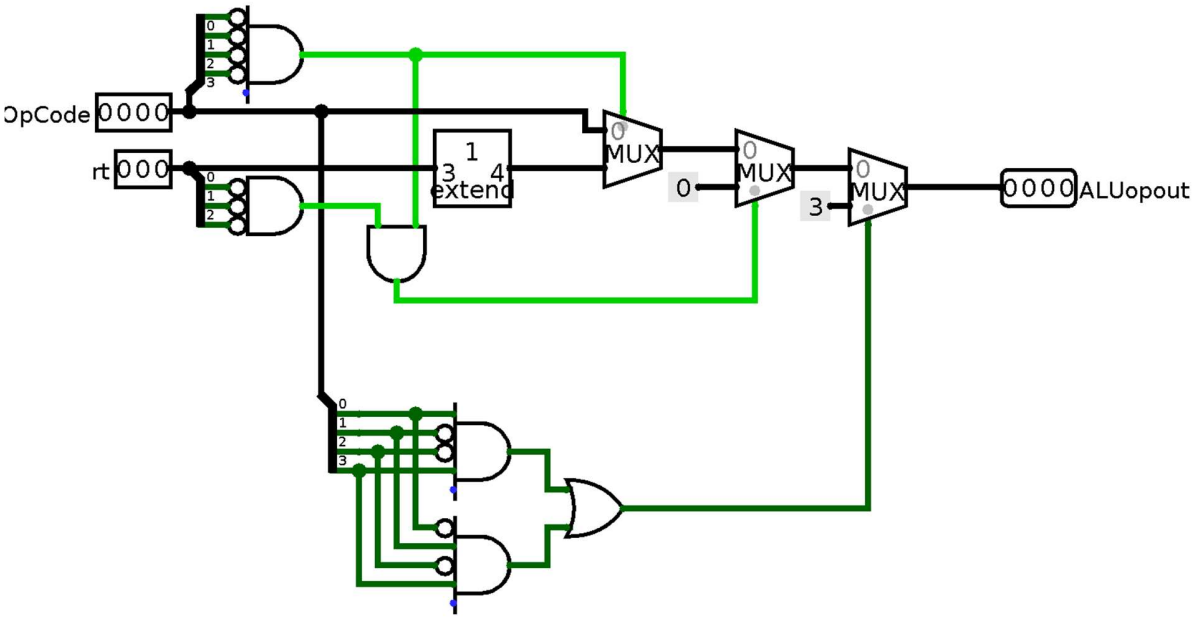
Nadat de nodige operaties zijn uitgevoerd zijn uitgevoerd, moet deze data ofwel een adres in het datageheugen voorstellen (alleen bij de geheugen instructies), ofwel een waarde die terug naar de register file moet geschreven worden. De “load word” operatie kan echter ook naar naar de register file schrijven, daarom multiplexen we de output van de ALU met de eventuele output van het datageheugen.

4. Main control logic



De hoofd controlelogica, komt neer op één grote decoder. Deze bepaald waar er welke input moet doorgelaten worden in welke multiplexer in welk geval. De “memory read” zal enkel aan staan voor de “load word” operatie, namelijk wanneer er data wordt gelezen uit het geheugen. De “memory write” staat aan voor de “store word” operatie, wanneer er dus naar het geheugen geschreven wordt. “Memory to register” staat aan voor “load word”, wanneer er gelezen wordt uit het geheugen en de output gemultiplext wordt met de output van de ALU (die in dit geval een adres voorstelt). ALU-source bepaald of er met een immediate wordt gerekend of een output uit een register (alleen bij de geheugen operaties). Ten slotte hebben we “register destination” en dit is de multiplexer die zegt of we naar de eerste sectie van de instructie moeten kijken voor het register adres ‘rt’ of de laatste drie.

5. ALU control logic



De ALU control logica bevat vier gevallen, bepaald door de eerste sectie bits uit de instructie en de laatste sectie bits. Het eerste geval is het geval “zero”, we schrijven nul naar het gekozen register. We zetten de ALU opcode dus op 0000 ofwel zero operatie.

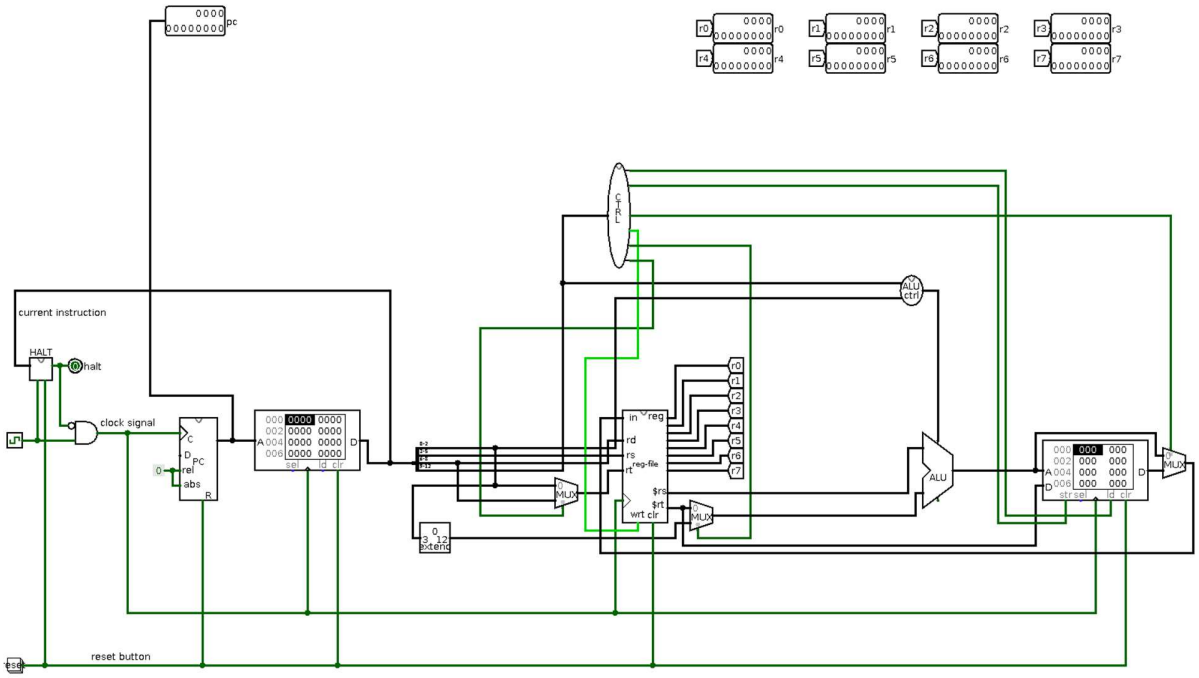
Het tweede geval is de unary instruction, namelijk wanneer de instruction bits allemaal nul zijn. In dit geval kijken we voor de ALU opcode naar de laatste sectie bits van de instructie. Omdat deze maar drie bits lang is en we onze ALU opcode zodanig hebben gekozen, moeten we enkel een “1” vooraan toevoegen om de juiste ALU opcode te genereren.

Het derde geval is de binary instructie. We laten de instructie bits dienen als ALU opcode.

Ten slotte hebben we de geheugen instructie, deze encodeerd geen ALU opcode, maar deze heeft dan ook maar één ALU opcode nodig, namelijk 0011 (add operatie) we tellen in alle gevallen een immediate op bij een adres in één van de registers.

We beseffen dat dit ook in één multiplexer kan (omdat er geen voorrangsregels zijn in dit geval), maar we kiezen voor deze oplossen omdat een extra encode zou resulteren in meer eps en dit zou ons meer tijd kosten dan nodig.

6. Grand picture



Als we alles combineren, bekomen we dit resultaat. Een datapath dat instructies kan tellen en kan laden uit een op voorhand te definiëren geheugen. Een register file waarop we operaties kunnen uitvoeren en terug kunnen opslagen in een te kiezen register. Een data geheugen dat het toelaat om te gebruiken data op voorhand te definiëren en berekende data terug op slaan. Dit alles wordt dan in goede banen geleid door een logica controller en een ALU opcode controller, die hun output afleiden uit de instructie in het instructie geheugen.

7. Conclusion

Het merendeel van de moeilijkheid zat het hem in het maken van ALU. Het datapath zelf was vooral rekening houden met de verschillende soorten instructies en correct handelen naargelang het type. Al bij al vonden we dit geen moeilijk project, maar hadden we wel een grote hulp aan het boek (voor de layout) en de slides (voor het op een rijtje krijgen van alle instructies).