Processoren voor

Programmeerbare embedded cpu-cores bepalen in belangrijke mate de programmeerbaarheid van systemen. Met name waar het gaat om 'controle-achtige' delen van de applicatie, zoals user interfaces en dergelijke, waar nauwelijks sprake is van parallellisme. Het is een type processor dat goed met tools wordt ondersteund en waarop de eindgebruiker eigen applicatiesoftware kan ontwikkelen.

GER SCHOEBER, High Tech Automation, Son

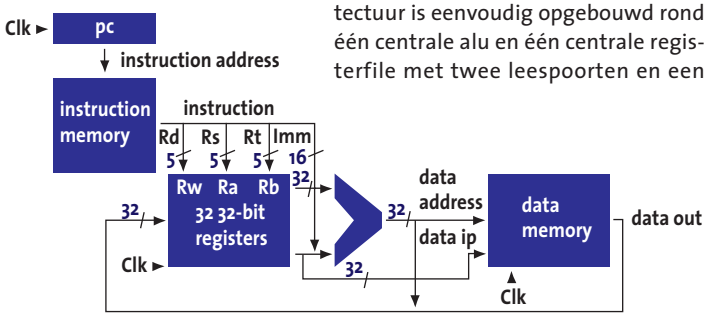
De programmeerbare cpu is zodanig ontworpen dat die voor zoveel mogelijk applicaties kan worden ingezet. Dit houdt in dat het ontwerp is geoptimaliseerd naar de *clock rate*. In het eindstadium van het ontwerp gebeurt dit zelfs vaak nog met de hand. Met name dit handwerk wordt steeds meer als een groter nadeel ervaren bij de overgang naar steeds kleinere procestechnologie. In tegenstelling tot standalone cores die alleen op clock-performance zijn geoptimaliseerd is een embedded core een compromis waarbij ook oppervlakte en vermogensdissipatie een belangrijke rol spelen.
Voorbeelden van leveranciers van programmeerbare embedded cpu-cores zijn ARM (telecom, laag vermogen, compacte code), Mips (zeer populair) en Sparc. Afgeleiden van algemeen toepasbare cpu's voor embedded applicaties komen verder onder meer van Intel, NEC, Hitachi, National en Motorola.
Programmeerbare cpu-cores beschikken over een zogenoemde ISA (instructieset-architectuur). De ISA bepaalt de instructies die zichtbaar zijn voor de programmeur en is dus het interface tussen hardware en software. Wat ISA's betreft kunnen we het volgende onderscheid maken:
- *implicit operands*, denk aan accumulator-machines en stack-machines, waarbij het voordeel bij de laatste duidelijk de compacte code is, bijvoorbeeld de ST-20 processor;
- *explicit operands*, general purpose registers, denk aan register/memory en register/register.
Impliciet en expliciet geven aan of de operanden impliciet of expliciet aanwezig zijn. Om de verschillen te verduidelijken is éénzelfde bewerking, namelijk C = A + B uitgewerkt in figuur 1.
Als voorbeeld van een register/register-processorarchitectuur is in figuur 2 (weliswaar in een vereenvoudigde weergave) de architectuuur van de MIPS 3930 weergegeven. Deze architectuur is eenvoudig opgebouwd rond één centrale alu en één centrale registerfile met twee leespoorten en een schrijfpoort. Het aantal registers is 32. Zie figuur 3 voor de instructieset-formaten zoals de cpu die kent. Verder kent de cpu een drietal groepen van instructies (zie figuur 4).
Een nadeel van deze architectuur is dat de cyclustijd van de processor wordt bepaald door de traagste instructie, in casu de load-instructie. Om dit op te lossen wordt *pipelining* toegepast, waarbij elke instructie wordt opgedeeld in vijf basisstappen:

- *Ifetch*: leesactie op het instructiegeheugen;
- *RF-read*: het lezen van de operanden uit de registerfile;
- *ALU*: ALU-operatie voor de berekening van het adres;
- *Dmem*: leesactie op het datageheugen;
- *RF write*: schrijfactie in de registerfile.

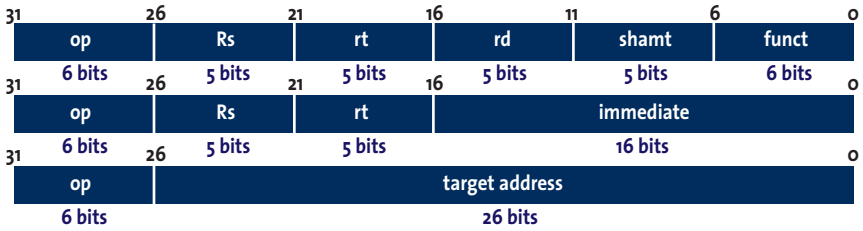
Het voordeel is dat tijdens elke nieuwe klokcyclus een nieuwe instructie kan worden opgestart. We zeggen dat de CPI (*cycles per instruction*) dan gelijk is aan 1. Dit kan echter tot een probleem leiden wanneer we een *LW*- en een *ADD*-instructie na elkaar gebruiken daar een *ADD*-instructie slechts vier *pipeline stages* nodig heeft (geen *Dmem*) en er een resource-conflict kan optreden op de schrijfpoort van de registerfile. Dit is op te lossen door een dummy *Dmem*-stap toe te voegen aan instructies van het register-type.
Een ander probleem kan optreden wanneer een instructie informatie nodig heeft die wordt geleverd door een instructie daarvoor. Deze data-afhankelijkheden worden ook wel *data hazards* genoemd. De oplossing hiervoor bestaat uit het toepassen van *bypasses* in

Stack	Accumulator	Register - memory	Register - register
Push A	Load A	Load R1, A	Load R1, A
Push B	Add B	Add R1, B	Load R2, B
Add	Store C	Store C, R1	Add R3, R1, R2
Pop C			

1. Om de verschillen te verduidelijken tussen implicit operands en explicit operands is de bewerking C = A + B uitgewerkt.



2. De instructieset-formaten zoals de MIPS 3930 die kent.



3. De architectuur van de weergegeven als voorbeeld van een register/register-processorarchitectuur.

de core (figuur 6). Dit is hardware die op run-time de data-afhankelijkheden detecteert en oplost. Deze additionele routing-hardware vereist meer multiplexing en kost wel wat extra tijd. Ook de vergelijkingen, die zijn uitgevoerd in hardware om te detecteren of het resultaat van een vorige instructie nodig is voor een volgende, kosten extra rekenvermogen.

Pipeline interlock

Nog lastiger kan het worden wanneer we een *LW*-instructie beschouwen. Een volgende instructie heeft dan data nodig die door een vorige instructie vanuit het datageheugen naar de registerfile wordt geladen. Dit is niet enkel met een bypass op te lossen. Er zal dan een extra *pipeline interlock* moeten worden toegevoegd (figuur 5). Ook dit zal de processor zelf moeten kunnen herkennen. Al dit soort 'extra's' maakt dat general purpose processoren veel meer energie verbruiken dan dedicated processoren. Compilers kunnen hier ook nog wel het een en ander in doen, maar vaak wordt het toch door de programmeur zelf opgelost.

Pipeline-interlock wordt ook gebruikt bij een vermenigvuldiging. Sommige versies van de MIPS-processoren hebben een vermenigvuldiger die echter twee klokcycli nodig heeft om een resultaat te produceren. Ook dit probleem wordt opgelost via (weliswaar een extra) pipeline-interlock. Dan zijn er nog versies met een deler die 35 cycli nodig hebben. We hebben dan een pipeline interlock van 35 cycli!

Naast de zojuist geschetste data hazards kun je je voorstellen dat er ook *control hazards* zijn. Control hazards treden op bij branch-instructies. Zo'n branch-delay-slot kan worden opgevuld met een *nop*-instructie hetgeen vaak door een compiler wordt opgelost. Voor embedded programmeerbare cpu-cores is het mogelijk goede en robuuste compilers te ontwikkelen. Dit is een gevolg van onder meer de centrale registerfile en het feit dat alle instructies sequentieel worden uitgevoerd. De enige vorm van parallellisme is pipeline-parallellisme.

Een tweede tool dat belangrijk is voor de architect is de *instructieset-simulator* (ISS). Van de architectuur van de processor wordt een simulatiemodel gemaakt, bijvoorbeeld in C of in VHDL, waarop de gegenereerde assemblercode kan worden geëxecuteerd. Dit is vooral belangrijk als er ook nog andere componenten in het systeem aanwezig zijn die gebruik maken van gemeenschappelijke resources. Een belangrijk nadeel is echter de simulatiesnelheid. Om hieraan tegemoet te komen wordt er onderzoek verricht naar modellen van cores die werken op een hoger niveau van abstractie en toch een nauwkeurige schatting toelaten.

Achtergrond

Auteur Ger Schoeber is consultant software- en systeemarchitectuur en gestart met de opleiding tot systeemarchitect bij het Eindhoven Embedded Systems Institute (EESI) Met enige regelmaat zal hij met een redactionele bijdrage deze kennis ook voor de lezers van PT Embedded Systems beschikbaar maken.

Rode draad in de opleiding tot 'systeemarchitect embedded systems' zijn de trefwoorden: *silicon, software, system* en *stakeholder* (de zogenoemde 4 s-en). De opleiding richt zich met name op de hiaten tussen deze disciplines, die niet als zelfstandige onderdelen worden gezien maar als één geheel.

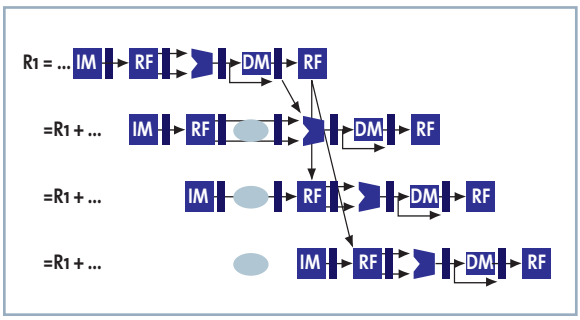
Het eerste artikel komt vanuit de *silicon* en betreft de architectuur van een viertal type processoren: embedded cpu cores (dit artikel), embedded dsp-cores, applicatie-domeinspecifieke processoren en applicatiespecifieke processoren (in volgende edities). ◻

Operand alu instructies
ADD rd, rs, rt R[rd] = R[rs] + R[rt]

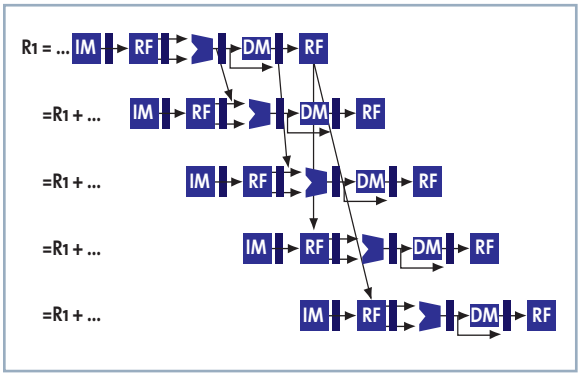
Load en store instructies
LW rt, rs, imm16 R[rt] = Mem R[[rs] + signext(imm16)]

Jump en branche instructies
BEQ rs, rt, imm16 if (R[rs] - R[rt] eq 0)
then PC = PC +4 + signext9imm16)
else PC = PC + 4

4. Verder kent de cpu een drietal groepen van instructies.



5. Soms is het nodig een extra pipeline interlock toe te voegen.



6. De oplossing tegen data hazards bestaat uit het toepassen van bypasses in de core