En tirade om prosessorer, språk og utviklingsmetoder

Min første datamaskin var en *ZX81*, utgitt i 1981. Den hadde en *Zilog Z80* CPU og kunne programmeres i *Basic*. Treg som en isbre, men den hadde en såkalt rask modus: Skjermen ble drevet av CPU-en, så å slå den av økte hastigheten mye, relativt sett. Den var et flott stykke ingeniørkunst: Det var totalt tre brikker på hovedkortet inkludert CPU, foruten RAM som var en eller to ekstra brikker. Andre hjemmedatamaskiner på den tiden var bygget opp av omtrent 20 brikker, så den var billig å produsere og halvmorsom. Her avbildet ved siden av en *Weighted Companion Cube* som størrelsereferanse:

> Utdrag fra boken *Ekte Programmering* (2021) av Corneliusen & Julin.
> Mer informasjon: www.ignorantus.com/ekte/

Deretter gikk jeg videre til *Commodore 64*. Det kan ha vært en *VIC-20* i veien før det: Broren min hadde en, og vi hadde det gøy med den, men 3,5 KB RAM får deg ikke så veldig langt. Commodore 64 hadde hele 64 KB RAM og ganske bra spill, men den var fremdeles 8-bit. Den brukte en *MOS Technology 6502*-variant kalt *6510*. Programmering av den var litt enklere enn Z80, men bare litt. Basic kjørte i vei litt raskere enn en isbre. Den hadde noen spesialtilpassede video- og lydbrikker i seg, og de virkelig kule laget, og lager fremdeles, flotte greier for den. Jeg ble grundig imponert over hva folk klarte å få til med den da. Selv kom jeg aldri veldig langt med den: Skrev noe maskinkode i et forferdelig verktøy, men det var ikke morsomt.

En CPU utfører instruksjoner, én etter én. Dette grunnleggende prinsippet har ikke endret seg mye på 50 år. Riktignok har det skjedd noen fremskritt i senere år, men det er kjernen i det. Selvsagt kan en CPU ha mer enn én utførelsesenhet, og de kan ha pipeliner og alt det der, men faktum gjenstår: Organiseringen av instruksjonene i minnet forblir konstant, og CPU-en må dekode dem i rekkefølge. Den kan selvfølgelig endre på rekkefølgen, men det finnes en optimal rekkefølge å plassere dem i. Og egentlig, hvor mye omstokking kan gjøres ved høye klokkehastigheter? Ikke mye. Det vil alltid være en fornuftig rekkefølge: Start innlasting av data så tidlig som mulig og bruk dem så sent som mulig - og bruk helst alle registrene. Det er ikke rakettvitenskap. Eller, det er det faktisk innimellom, både for de som skriver kode for rakettstyring, og de som prøver å skvise siste livsrest ut av en billig mikrokontroller.

CPU-er arbeider effektivt med grunnleggende datatyper. De har aldri hatt, og vil sannsynligvis aldri ha, noe begrep om funksjoner, metoder, klasser, typeløse data eller andre abstraksjoner. Det er ett unntak: Intel ga ut en slik CPU i 1981: *Intel iAPX 432*. Den mislyktes fullstendig, hovedsakelig fordi den var fryktelig langsom og fryktelig dyr, og kompilatorene var fryktelig dårlige. Ikke noen vinnerkombinasjon da, og heller ikke en vinnerkombinasjon i dag.

Så hvorfor konvergerer programmeringsspråk mot slike konsepter? For en programmerer som liker å skrive hurtig kode er det forvirrende. Etter min mening bør rask eksekvering alltid vies oppmerksomhet, og spesielt nå som både folk og firmaer kjører kode på leide servere i skyen og betaler per CPU-sekund brukt. Det kan være en konspirasjon av de som eier maskinene, som tvinger brukerne til å programmere i trege språk for å bruke mer CPU-tid enn nødvendig. Men det er litt vel søkt. Antakelig.

Jeg antar at det har rot i oppfinnelsen av objektorienterte språk på 60-tallet. De fikk ikke utbredt popularitet før midten av 90-tallet, da raske CPU-er begynte å bli tilgjengelig for de fleste. Internett tok av, flere mennesker fikk datamaskiner, og ikke bare hackerne ønsket å skrive programmer. Det var alltid et håp om at den neste generasjonen av CPU-er ville kjøre koden bedre, og gjøre skikkelige språk som krever en viss innsikt, som *C*, unødvendige.

I min lett fargelagte oppfatning har det skjedd to store revolusjoner i CPU-verdenen de siste 40 år: Lanseringen av *Motorola 68000* og, mye senere, *Tilera TILE-Gx*. De gjorde stort sett det samme som alle andre CPU-er, bare bedre. Mye bedre. De utførte fortsatt instruksjoner én om gangen og hadde ingen begrep om de nevnte abstraksjonene som har vært så populære siden midten av 90-tallet.

Et viktig problem på 70-tallet var at CPU-ene ikke var brede nok. Eller raske nok. Eller hadde nok minne. Eller hadde raskt nok minne. Poenget burde være klart som kildevann. Z80- og 6502-varianter var enormt vellykkede i hjemmedatamarkedet, men de opererte på 8-bits datatyper. Hjul måtte finnes opp for å addere eller multiplisere 32-bits tall. La oss ikke nevne divisjon. CPU-ene hadde ikke de grunnleggende aritmetiske operasjonene for å skrive hurtig kode på en sivilisert måte. Til tross for vid utbredelse av hjemmemaskiner ble de sjelden brukt til noe nyttig av andre enn datidens hackere. Jeg husker et program for Commodore 64 som kunne holde rede på innholdet i fryseboksen. Flott idé, bortsett fra at det tok et par minutter å laste inn fra kassett. Morsomme tider!

Som alle de kule ungene husker lanserte Prince albumet *Prince* i 1979, og Motorola lanserte 68000-serien. La oss fokusere på sistnevnte. Det bør nok også nevnes at Pink Floyd ga ut albumet *The Wall* det året. 1979 var et flott år, både innen musikk og CPU-er.

Uansett var 68000 et stort fremskritt for hackerne, og senere resten av verden. Den kan betraktes som den første 32-bits CPU-en. Det var den egentlig ikke, da den bare hadde en 16-bits databuss og 16-bits ALU, men fra et programmeringsperspektiv var den det. Den hadde et enormt antall registre: Åtte 32-bit dataregistre og like mange 32-bit adresseregistre. Minnetilgangen brukte 32-bits pekere, uhørt i bransjen så langt. Ja, det var bare 24 bits som ble brukt, men registrene var 32 bit. Multiplikasjon var 16-bit ganger 16-bit og ga et 32-bits svar. Det var også en divisjonsinstruksjon som ga både kvotienten og resten som resultat. Divisjon var fryktelig tregt, men med verktøy som dette var det mulig å skrive ordentlige programmer.

Og den var *big-endian*. Nå var *little-endian* ganske smart på 8-bits CPU-er siden det ble enklere å addere brede tall rett fra minne: Bredere ord er byte-reversert, så de åtte lave bitene kommer først. På en 32-bits CPU betyr det ikke noe, big-endian er den logiske måten å jobbe med faktiske bits inne i et ord bredere enn 8-bit: En big-endian minnedump gir mye mer mening enn en little-endian, ihvertfall for folk som vet hva en minnedump er. Jeg aner ikke hvorfor nyere CPU-er bruker little-endian. Gammel vane er vond å byte-reversere.

Flere populære datamaskinplattformer brukte 68000 CPU-en. Den største var selvfølgelig Commodore Amiga, men jeg er sikker på at det vil møte mange protester fra Atari ST- og Apple Macintosh-fans. Amigaen hadde de mest avanserte brikkene for grafikk, lyd og I/O den gangen, langt foran konkurrentene.

Nyere revisjoner av 68000 kom med ekte 32-bit adressering, ekte 32-bit ALU og andre moderne funksjoner. Det var en fantastisk brikke å lære ekte programmering på.

Og hva skrev vi kode i? Lenge i 68000 assembler. Så begynte delvis anstendige C-kompilatorer å dukke opp til Amiga, så da lærte vi C. C er et av få språk som legger så lite tull som mulig mellom koden som skrives og maskinkoden som genereres via assembler. Språket har noen særegenheter, men så lenge du ikke skriver søppel får du ikke søppel. Det gir deg stort sett frie hender. Når ting ikke er raskt nok er det enkelt å legge inn litt assembler-kode i programmet.

Selvfølgelig varte ikke 68000-linjen evig. Intel klarte omsider å lage en brukbar CPU som var 32-bit: *Intel 80386*, og Microsoft laget et operativsystem som støttet den. Det tok lang tid: Mens '386 ble lansert så tidlig som 1985, fikk ikke Microsoft et skikkelig operativsystem i butikken før 1993, kalt Windows NT. For massemarkedet drøyde det helt til 1995, med Windows 95.

Som nevnt, objektorientert programmering, et konsept kokt sammen på 60-tallet, tok ikke skikkelig av før på midten av 90-tallet da CPU-er begynte å bli mye raskere og mer effektive. Det var fullt av forskjellige CPU-er på markedet: *Digital Alpha* var ganske flott, *Suns SPARC*-serie var kul, *MIPS IV* og *PowerPC* hadde dukket opp og det var mange, mange flere, i tillegg til Motorola og Intel sine greier, selvfølgelig. Det er morsomt hvordan det gikk til slutt (dvs. nå i 2020): Hovedsakelig to arkitekturer er i bruk, ARM og x86. To. Så når du vil skrive rask kode er dette antall arkitekturer du må tenke på. Ikke mye.

Da CPU-ene ble raskere og raskere, ble språk som gjorde ting helt annerledes enn C på en eller annen måte nødvendig. Koden ble konvertert til et mellomspråk som kunne emuleres på mål-CPU-en. Da man senere oppdaget at emulert kode var veldig treg, forsøkte man å lage maskinkode rett fra mellomkoden. Den trenden fortsetter til i dag. Det sugde da og det suger nå. Det vil fortsette å suge i overskuelig fremtid.

Dessverre fanget disse språkene vinden. De blåste inn galskap som dynamisk minneallokering som krevde såkalt søppelhåndtering, abstraherte bort vanlige datatyper og skrotet alle konsepter om effektiv programmering. Håpet var at den neste CPU-en som kom i butikken ville kjøre den såkalte koden raskt.

Noen CPU-produsenter trodde det ville bli bedre å integrere kjøring av noe mellomkode i CPU-en. ARM var en av dem, med en løsning som kunne utføre noen av instruksjonene, kjent som *bytecode*, direkte i maskinvare. Dessverre var teknologien belemret med hemmelighetskremmeri, så *open source*-samfunnet tok avstand fra den. Den er nå død i vannet. De fulgte opp med mindre lukkede versjoner, men jeg har virkelig ikke hørt særlig mer om den siden begynnelsen av 2000-tallet.

Og dessverre kjørte Intel også på grunn med sin *Netburst*-arkitektur fra slutten av 2000 og *Pentium 4*-CPU-ene. De prøvde å bruke ekstremt dype pipeliner for å skvise ut noen ekstra MHz. Det går bra til det er en forgrening i programmet: Å rydde pipelinen tok en evighet når dette skjedde. De holdt fast på dette til 2006, selv om den underpresterte kraftig. *AMD*, som lager Intel-kompatible CPU-er, klarte å holde skipet flytende i denne perioden. Intel fant til slutt ut at de måtte gå tilbake en revisjon og senke klokkehastigheten, og de vant ytelseskronen igjen med *Core*-arkitekturen fra 2006.

I all denne galskapen blomstret de rare språkene. Du måtte ikke lenger være en ekte programmerer eller forstå pekere for å skrive programmer! Noen av disse språkene hadde ikke engang typer. Et vendepunkt for meg var dagen jeg forsto hvorfor slike språk trenger den tredobbelte likhetsoperatoren: Siden ingenting har en type, hvordan kan det avgjøres om to objekter er eksplisitt det samme, ikke bare evaluerer til det samme? Det førte til en ildstorm av interessante feil. Jeg følte et øyeblikks illuminasjon, etterfulgt av tristhet. Så bestemte jeg meg for å blåse i denne språkgruppen, som enhver sunn person burde gjort.

Jeg hadde hatt den samme følelsen noen år tidligere da jeg forstod konseptet kjent som funksjonell programmering: Alt blir evaluert som matematiske funksjoner som returnerer verdier. Jeg har nyheter for de som fant det opp: CPU-er fungerer ikke på den måten. Moderne metoder har gjort denne klassen av språk mer effektiv, men jeg vil si de begynte i feil ende. Dette rådet gjelder alle som designer språk: Hva med å se på hvordan CPU-en fungerer først, og så snekre sammen et språk etterpå? Hint: Alle CPU-er fungerer ganske likt.

La oss heretter referere til moderne programmeringsspråk, hvor søppelbiler kjører rundt og/eller typene er uklare, som lekespråk, med *Phyreztorm* som navnet på et slikt språk. Det er fiktivt, selvsagt.

Jeg kan høre dem rope: "Bare skriv det i Phyreztorm, så blir det raskt! Veldig snart." Jeg telte omtrent to nye programmeringsspråk på nettstedet *Slashdot* hver uke i årene før dot.com-boblen sprakk i år 2000. De pleide å løse ett problem veldig bra. Hver. Og det startet en trend som fortsetter til i dag: Hver gang nye versjoner av lekespråkene blir publisert må alt skrives om, siden de åpenbare feilene fra forrige versjon må rettes. Flotte greier!

Men her er kjernen: Var og er disse språkene egentlig forskjellige? Innfører de noen revolusjonerende nye konsepter som vil kjøre fort? Det ville være den ultimate testen, og det eneste svaret jeg har sett hittil er "nei". Nyvinningen er praktisk talt null. I stedet for å konvergere mot det man forventer, rask kode, konvergerer de mot det esoteriske og teoretiske. Et godt eksempel er Microsofts *Bosque* som eliminerer "unødvendige" ting som løkker. Virkelig. De fleste av disse språkene er egentlig bare oppgulp av andre slike språk. Jeg kommer tilbake til effektivitetsspørsmålet senere.

Uansett, i moderne tider, hvor selskaper leier CPU-tid fra store serverfarmer, må spørsmålet stilles: Hvorfor vil de ikke bruke mer effektiv kode som koster mindre? For dem som bryr seg om miljøet bør svaret være åpenbart. Her er et tilfelle der effektivitet og miljøvern faktisk stemmer overens. Det skjer ikke veldig ofte, siden det er to helt ubeslektede ting. Det er morsomt, og også litt trist.

Ifølge den samme trenden, hvorfor er ikke smarttelefonselskapene interessert i effektiv kode? Hvis det meste av det som kjører på smarttelefoner, er treg, emulert kode, vil det ikke tappe batteriet raskere? Hvorfor vil de ikke forlenge batteriets levetid? Jeg husker jeg kjørte på hyttetur med noen venner for et par år siden. Tre timer etter avgang begynte de febrilsk å lete etter steder å koble til laderne sine. Noen hadde til og med ekstra klumpete batteripakker med seg. Et enkelt svar er at lading reduserer batteriets levetid, så mer lading fører til økt telefonsalg. I det minste økt batterisalg, hvis det er mulig å erstatte det. Men jeg prøver å unngå konspirasjonsteoriene, selv om det ikke er lett.

Som nevnt klarte AMD å finne på noe nytt i denne perioden, rundt 2003: 64-bits utvidelser for x86-instruksjonssettet, kalt *x64*. (Det finnes et forvirrende utvalg av navn på det. La oss holde oss til x64.) Vi nærmet oss 64-bits-alderen! Ikke mer av den gamle 32-bit dritten. Om det finnes én sannhet i databehandling, er det at alt blir gammelt og slitt etter noen år. Men i motsetning til lekespråk, lever de gode CPU-konseptene videre: Kunne du håndtere 32-bit adressering greit ville 64-bit være veldig enkelt.

Men du vet hvordan historien går. Hvis ikke bør du finne noe annet å lese. Jeg vil at registrene skal være minst 64-bit, og jeg vil ha mange av dem. Og jeg vil jobbe med mindre størrelser parallelt. *SIMD* (Single Instruction Multiple Data) løste flere av disse problemene. Det dukket opp diverse Intel/AMD SIMD-utvidelser: *MMX* kom i 1997 og ble etterfulgt av en rekke spin-offs i årene etter. Noen av de mer populære ble kalt *SSE, SSE2, SSSE3, AVX, AVX2* og *AVX-512*. MMX var 64-bit, SSE-varianter 128-bit, AVX/AVX2 256-bit og AVX-512 512-bit. Ja, det er noe forenklet, og jeg kommer tilbake til det senere.

Dessverre visste de ikke når de skulle stoppe. Å skrive effektiv kode med 512-bits registre blir fort rotete når det bare er 32 av dem. Og samsvaret mellom instruksjonssettene forble mangelfullt. Toppen var 128-bit. Jeg sier det igjen: Toppen var 128-bit. I stedet for å lage kjempestore 512-bits enheter, burde det vært flere 128-bit enheter. Eller enda bedre: Flere SIMD-registre. I 128-bits SSE2 modus er det bare 16. Gi meg 32 så blir jeg glad. La oss kalle det *Mega-SSE2*.

Nylig, i sommeren 2020, har Intel annonsert en ny arkitektur som heter *Lakefield*. Mens detaljene er få, ser det ut til å være et skritt i riktig retning. Så vidt jeg skjønner er de bredeste SIMD-enhetene (AVX, AVX2, AVX-512) blitt pensjonert. Det vil bli interessant å se hvor dette ender når faktiske brikker blir tilgjengelige. Siste nytt på den fronten er at de sier at de ikke skal fjerne dem, men det blir vel litt som å spørre finansministeren om valutaen skal devalueres dagen før det skjer.

VLIW (Very Long Instruction Word) var ganske kult da det kom. Det var vanligvis DSPer som brukte det, dvs. brikker uten alle sikkerhetsfunksjonene i CPU-er fra Intel og lignende, men de fortjener hederlig omtale for å være banebrytende. Den grunnleggende ideen var at flere instruksjoner ble utstedt og sendt til forskjellige utførelsesenheter samtidig, som spesifisert av programmereren.

Teknologien hadde vært i bakgrunnen lenge, men i 1996 ble *Philips TriMedia TM1000* lansert, den første brikken av den klassen jeg syntes var faktisk nyttig. Den hadde fem utførelses-spor og kodet fem instruksjoner som én (jeg forenkler mye her); hvis ikke alle spor kunne fylles, uflaks. Det var rikelig med 32-bits registre: 128 av dem. Hver operasjon var skjermet, så det var mulig å slå sammen *if()*-grener, om det var den raskeste måten. Den hadde forskjellige operasjoner per spor: Alle fem hadde en ALU, alle fem hadde skyve-operasjoner, to hadde *DSPMUL* dvs. de kule SIMD-greiene osv. Alt var pipelinet også, bortsett fra flyttallskvadratrot og -deling, men de var der om nødvendig.

Kompilatoren hadde litt problemer med å generere nyttige fem-spors bunter av instruksjoner og utnytte de to SIMD-banene; men den hadde iboende funksjonsstøtte, også kjent som *intrinsics*: Skriv assembler-instruksjoner direkte i programmet som funksjonskall. Et av de beste nye programmeringskonseptene noensinne. Kompilatoren håndterte registertildelingen og instruksjonsrekkefølgen, og den gjorde vanligvis en utmerket jobb. Bortsett fra noen få tilfeller da den like gode assembleren kunne brukes. Den hadde begrenset suksess, men bleknet og endte sitt liv som integrert kontroller i fjernsynsapparater. En litt skuffende slutt for en banebrytende arkitektur.

Texas Instruments prøvde å lansere en åtte-veis VLIW-brikke tilbake i 2001, kalt *TMS320C6x*-familien, men de klønet det til ved å teipe samme to fireveis VLIW-kjerner. La oss ignorere det og holde hodet høyt. Den var en suksess, men den var vanskelig å programmere. Siden det var to fireveis kjerner, var det også to registerbanker, og dataene som trengtes var alltid i feil registerbank. Jeg likte VLIW til jeg prøvde å skrive kode for TI-brikken. Ga den opp.

Jeg brukte flere år på å lage artige ting på TriMedia, men *embedded*-verdenen konvergerte mot større SoC-er: System on Chip. En SoC er en CPU med all nødvendig I/O, og innimellom minne, i en enkel pakke. Noen ganger betyr det at det er en ekstra CPU inni der som håndterer I/O, og den CPU-en kjører programvare. Og programvare har feil. Sjokkerende! Husker at jeg laget noen greier på en *AMD Elan SC400* SoC tilbake i 1998, fikk noen problemer med DMA-kontrolleren og konsulterte den lokale guruen. Han ga meg en løst sammensatt bok på 600 sider som dokumenterte alle feil i brikken. Problemet mitt var selvsagt ikke oppført. Brukte en uke på å finne ut av det og sendte inn en feilrapport. Det ble lagt til i neste utgivelse av feildokumentasjonen, nå på 601 sider. Det var sånn da, og det er sånn nå. Det vil fortsette å være sånn i fremtiden.

En annen CPU som er verdt å nevne er den som ble brukt i *PlayStation 3*: *Cell*. Den ble lansert med brask og bram i 2006. Den hadde en *PowerPC*-kjerne for normal prosessering og åtte regnekjerner. De hadde alle lange, kompliserte navn. Ikke viktig. Hver regnekjerne hadde egen RAM og registre, og kunne gjøre småjobber lynraskt. De var koblet til en ringbuss, så den raskt kunne sende datablokker til naboene. En god idé som aldri ble utnyttet til sitt fulle potensiale. Den ble noe brukt i superdatamaskiner på den tiden, men senere glemt. Vel, de solgte nesten 90 millioner enheter av PS3, så den var vellykket, antar jeg. Jeg likte ideen, og de var morsomme å skrive kode for. Jeg ville bare ha flere av disse kjernene. Mange flere.

Og gjett hva? Ideen har sett en gjenoppblomstring i det siste i mega-brikker laget av firmaet *Graphcore*. I 2018 lanserte de sin første brikke, treffende kalt *Colossus*. Den har et stort, unnskyld, gigantisk, antall Cell-lignende kjerner per brikke som kan kobles til andre slike brikker via et høyhastighetsnettverk. De prøver å selge dem som en løsning for moderne AI- og maskinlæringsproblemer. Jeg vil gjerne ta den teknologien og prøve noe annet, men velkjent: Å forbedre normale problemer. Som Cell kan den muligens flerre gjennom store datasett på nye og effektive måter. Dessverre er antallet Graphcore-brikker i hulen min null. En morsom greie er at jeg søkte jobb der en eller to ganger, avhengig av hvem som teller. Det ble ikke noe av. Jeg klandrer dem ikke. Men det ville sikkert være gøy å ha en slik brikke å prøve ting på. Og jeg mener virkelig gøy. Noen ganger gir slik moro faktiske nye resultater. Og noen ganger gjør det det ikke, noe som er veldig vanskelig å selge til moderne programvareledere. Jeg kommer tilbake til det senere.

Uansett, dataverdenen står sjelden stille, og det gjorde den heller ikke i denne perioden: Rundt 2009 gikk GPU-gutta på slankekur, noe som gjorde brikkene deres tynnere og raskere, og de fikk flere kjerner. CPU-gutta ble enda fetere med mega-brede SIMD-varianter, den feteste var Intels *Xeon Phi* lansert i 2013. Den hadde 57 eller flere kjerner og AVX-512: 512-bit utførelsesenheter og 32 registre. Hver kjerne hadde en *Intel Atom*-basert kontrollkjerne som ikke klarte å holde tritt med dekoding, så flere tråder var nødvendig, via *Hyper-threading*, bare for å holde mega-SIMD-enhetene opptatt. Seriøst?

Hyper-threading ble lansert tilbake i 2002, og ideen var ganske god: Bytt raskt til en annen tråd hvis CPU-en stopper opp, som ved en minneaksess som ikke kan fullføres. Ikke ved at dekodingen stopper opp. Jeg vil påstå at det er bevis på dårlig design. Men jeg husker da Hyper-threading kom: Først tenkte jeg "åh flott, det vil øke gjennomstrømmingen" og deretter "men fillern da, hva med sikkerheten?" Vel, det har fått noe oppmerksomhet i det siste og er godt dekket i teknologipressen, så det trengs ikke å gå i detalj om det.

ARM prøvde sitt beste med *Neon* SIMD-utvidelser fra rundt 2005. Dessverre, i likhet med SSE2, led de av ikke nok registre: I den gamle (ikke den helt første, altså) Neon er det 32 64-bit registre som kan kalles D-registre, eller de kan pares som seksten 128-bit Q-registre, så lenge de er ved siden av hverandre og den første har et partallsnummer. Jaha. Heldigvis ryddet de opp i registerrotet i *AArch64* Neon, som ble lansert i 2011 med 32 128-bit registre. Takk, ARM! Endelig kom Mega-SSE2, men på en helt annen arkitektur. Og de ryddet opp i mange særheter i instruksjonssettet også. Neon føltes ikke like påklistret lenger. Spørsmål til Intel: Kan jeg få SSE2 med 32 registre? Men, det som kjører fort på *NVidia Tegra X2* sin Neon-enhet, kjører kanskje ikke like raskt på noen andres. Og som kjent kjører ingenting fort på de beryktede NVidia *Denver*-kjernene som bruker noen rare dynamiske rekompileringsgreier. Det er omtrent så ille som det høres ut. Jeg skal begrave det liket senere.

Det som trengtes var en revolusjonerende 64-bits CPU, muligens med færre sikkerhetsproblemer. Som nevnt, i 1979 revolusjonerte Prince og Pink Floyd musikkverdenen (vel, stort sett Pink Floyd) og Motorola databehandlingen. I 2010 var det største som skjedde på mediefronten at filmen *Kick-Ass* kom ut, så det var et ganske tregt år. Og ja, det er et veldig godt poeng at de filmene, musikken og spillene jeg liker sjelden matcher med resten av verden. Men på prosessorfronten skjedde det store ting: Tileras *TILE-Gx* kom på markedet. Det var det første pustet av frisk 64-bits luft på lenge.

Tilera var kjent for å lage brikker med mange kjerner: De hadde lansert *TILE64* 64-kjerneprosessoren i 2007, og *TilePro* i 2008. Jeg lekte litt med de CPU-ene rundt 2009, men følte at noe manglet. Så dukket TILE-Gx opp, og den hadde alt. Nesten. Det var 64 64-bits registre, tre utførelsesspor, de fleste SIMD-operasjonene du kan tenke deg, og en superkort to-trinns pipeline. Og endelig mange kjerner: 36 i den første versjonen, 72 i en senere. Alt dette klokket på komfortabelt lave 1.2 GHz, derav den korte pipelinen. Korte pipeliner er bra. Tro meg på det.

Grenforutsigelsen ble gjort på riktig måte: Programmereren bestemte. Hver gren skulle enten forgrene seg eller ikke, og det ble hardkodet i instruksjonen. Var det feil kostet det bare 2 klokkesykluser, takket være den korte pipelinen. Kompilatoren *GCC* gjorde det stort sett riktig, men kunne overstyres med et innebygd kall.

Og Tilera gjorde statusregistret på riktig måte: Intet statusregister i det hele tatt. Det var sammenligningsinstruksjoner som lagret resultatet i et register som 0 eller 1 eksplisitt. Og greninstruksjonene forgrenet seg på et register og en gitt betingelse. Enkelt.

Alle disse kule greiene gjorde den til en lek å programmere. Gitt at det var så mange registre, ble risikoen for at kompilatoren skulle bli forvirret og søle registre utover stacken redusert betydelig. Å bruke intrinsics for å utnytte SIMD-ting var enkelt. Andre produsenter har fomlet her; jeg kommer tilbake til det.

Men uansett, et register var til slutt 64-bit. Virkelig 64-bit. De kunne brukes som 64x1, 32x2, 16x4 eller 8x8. Og det stoppet ikke der. Den hadde den beste instruksjonen noensinne: *v1ddotpus*, som beregner to fireveis 8-bit prikkprodukter på en gang, og gir to 32-bit svar på to klokkesykluser. Som med de fleste ting i TILE-Gx var alt pipelinet, slik at det kunne startes og avsluttes en per syklus.

Og så var det instruksjonen kalt *shufflebytes* som stokker bytes, åpenbart, fra to kilder til en. Tenk Intels *mm_shuffle()*, bare nyttig. Og det var *dblalign* som justerer to 64-bit kilder over en 128-bit grense. Kjekt!

Den hadde ikke mye flyttallsstøtte. Det som var der virket påklistret: Fire instruksjoner, og sju-åtte sykluser var nødvendig per operasjon. Og det var bare *mul* og *add/sub*. Det var også *double*-støtte, men siden det ikke var noe sted å lagre (dessverre nødvendige) statusbiter, var det ekstra instruksjoner for det. For *float* ble statusbitene lagret i de øvre 32 bitene i registret.

Dessverre fikk du lite av denne storheten ved å skrive standard C-kode. Auto-vektorisering hadde ikke tatt av ennå (*drive-by-optimalisering* liker jeg å kalle det). Jeg kommer tilbake til det senere. Men denne CPU-en fløy virkelig når

det ble gjort riktig. Og den kom med alle sporingsverktøy og simulatorer og slikt som trengs for å skrive genuint god kode.

Men den absolutt største nyvinningen i denne CPU-en var det integrerte maskenettverket. Det var fire av dem, og to var tilgjengelig for brukeren. Hvert var 64 bits bred og klokket på samme 1.2 GHz. Å sende data fra en kjerne til en annen var lynraskt: Den hadde en veksler på hver kjerne, så antall sykluser det tok å transportere 64-bits data til en annen kjerne var antall trinn langs den ene aksen pluss den andre. Andre produsenter på den tiden brukte mer og mer komplekse ringbussløsninger. Jeg aner ikke hvorfor. Jeg antar at Tilera hadde alle patentene på denne type nettverk og ingen ønsket å betale dem godtgjørelse. Men jeg bare gjetter. Uansett har noen nyere Intel-prosessorer begynt å bruke slike nettverk. Bra for dem.

Men dessverre, som med Betamax, kvalitet vinner ikke alltid. Tilera ble kjøpt av EZChip i 2014 for en latterlig lav pris: 130 millioner dollar i kontanter. Og så ble EZChip kjøpt av Mellanox i 2016. Og Mellanox ble kjøpt av Nvidia i 2019. Mellanox hadde fortsatt TILE-Gx brikkene på nettsiden sin før siste overtakelse, men det har ikke vært noen ny utvikling siden 2014. Jeg hadde noe innsideinformasjon om neste generasjons arkitektur, men jeg antar at den er like død som disko. Noen av hackergutta og jeg hadde bedt dem om å lage noen spesielle instruksjoner som ville gjøre videobehandling enklere. Antar at det ikke kommer til å skje.

Så TILE-Gx var gøy, og endel av tingene jeg laget presenteres senere i boken. I mellomtiden ga AMD ut sin underpresterende arkitektur kalt *Bulldozer* i 2011. Ideen var ikke så dum, og her forenkles det grovt for å gjøre det overkommelig: I stedet for Hyper-threading kombinerte den beregningsressursene til to kjerner, og begge kjernene kunne velge det de trengte fra et større reservoar. En kjerne kunne suse av gårde med alle enhetene om nødvendig. Flott idé, bortsett fra at dekoderen ikke hang med. Og den hadde bare en flyttallsenhet per to kjerner, som gjorde den ubrukelig for vitenskapelige applikasjoner. Eller beregninger generelt. Klønete.

Dessverre medførte dette at utviklingen hos Intel stoppet mer eller mindre opp. CPU-verdenen ble sittende fast med fire kjerner i lang tid. Prisen per kjerne for mer enn fire kjerner var latterlig høy i mange år. Jeg så ingen stor utvikling noe sted på lang tid. En venn av meg holdt sitt *Intel i7-2600K "Sandy Bridge"*-baserte system fra 2011 flytende i mange år på grunn av dette. Det føltes som i 2007, da Nokia hadde dominert mobiltelefonmarkedet siden starten: De var verdens konger, så det var ingen grunn til å innovere lenger. Det endret seg litt det året, ved lanseringen av iPhone.

Heldigvis, i 2017 kom AMD tilbake fra de døde. De klarte faktisk å lage en helt ny arkitektur, kalt *Zen*, bygget på et helt annet konsept: I stedet for en kjempestor brikke brukte de flere mindre brikker som hadde maksimalt åtte kjerner og en stor I/O-brikke i midten. Billig å produsere, og rask. Og så mange kjerner! Jeg betalte med alle kontantene mine pluss alle fremtidige Lotto-gevinster for den nåværende åttekjerners Intel-prosessoren jeg har. Nå, i 2020, er prisen for en 32-kjerners AMD *ThreadRipper* CPU omtrent den samme som den åttekjernede fra Intel. Og et godt navn er viktig! ThreadRipper høres så mye kulere ut enn de innsjøbaserte navnene til Intel. Når kommer *Crystal Lake*[2]? Jeg lurer virkelig på hva Intel kommer til å gjøre. Det blir spennende. Intel har alltid sprettet tilbake før, så jeg håper på enda mer spenning i CPU-markedet de kommende årene.

Hvis vi ser fremover, er det en interessant arkitektur i horisonten: *The Mill*. Den har vært under utvikling siden 2003, og representerer en revolusjon i hvordan minnet benyttes, og hvordan data blir manipulert, og hvordan grener håndteres, listen er lang. Alt er forbedret. Og det er ingen registre, bare et belte der ting faller utenfor kanten, så pipelinen er åpen og eksponert. Det er et spennende konsept, og hovedmannen bak det, Ivan Godard, har laget flere videoer om detaljene på YouTube. Dessverre, 17 år senere, er det fremdeles ingen brikker på markedet. Jeg anbefaler allikevel sterkt alle om å se disse videoene. De får deg til å tenke over mange ting på en ny måte.

Og likevel er grunnkonseptet det samme som før: Den utfører instruksjoner etter hverandre, og det er fortsatt ingen rare datatyper. Eller mangel på datatyper. Og ingen søppelbiler som kjører rundt og håndterer dynamisk allokert søppel.

[2] Antageligvis aldri. Crystal Lake er kjent fra Fredag den 13.-filmene. Og American Horror Story: 1984, selv om det aldri blir nevnt. Kjennere vet at det er der den finner sted.

I løpet av hele denne tiden har C stort sett bestått. I dag er standardisering av språket mer eller mindre under kontroll. C99-standarden er ganske nyttig og gir det en moderne stil. Men C er fortsatt det enkle språket det alltid har vært. Det du skriver oversettes stort sett til fornuftig kode. Den mest avanserte abstraksjonen er fremdeles *struct*, som lar deg gruppere data sammen.

Men, og det er alltid et men, det krever en programmerer som vet hva han eller hun gjør. Det krever stor interesse for programmering. Det krever at du faktisk skriver kode. På jobb og hjemme. Hvis du ikke har en datamaskin med tastatur hjemme, er du ikke en C-programmerer. Du skriver sannsynligvis noe som ligner på kode i Phyreztorm og tror det er programmering. Gjett hva som skjer når Phyreztorm erstattes av *Phyrekloud* neste år? Du er foreldet. Med mindre du lærer deg Phyrekloud, selvfølgelig. Som du vil, siden det er nesten det samme som det forrige, bare med litt mindre fryktelige feil. Og noen nye, som vil bli løst i *Phyreball*. Og hvis du synes navneskjemaet mitt er barnslig; et lekespråk (jeg vet egentlig ikke hva det er) som heter *Node.js* fra 2009, fikk en avlegger som heter *Deno* i 2018. Virkelig. Når vil *Done* dukke opp på markedet? Jeg gjetter på 2027, etter at de alle de nye fatale feilene i Deno krever et helt nytt språk. Igjen.

Tilbake til poenget: En god programmerer vil prøve ut ting bare for moro skyld. Jeg hadde en gang en dårlig sjef som hevdet at mye av programmering var kjedelig. Jeg forlot det firmaet kort tid etter. Det ble kjedelig.

Apropos moro: Programvareledere har en tendens til å unngå moro, siden det ikke alltid gir resultater og aldri passer inn i femårsplanen. Jeg overdriver kanskje litt med femårsplanen, siden store selskaper aldri planlegger lenger enn tre måneder, men poenget er fortsatt: Hvordan i all verden får de plass til en boks som heter "oppdage revolusjonerende ny metode for å beregne vertikale prikkprodukter på vår nåværende arkitektur" i sin PowerPoint-plan for neste periode? De har ingen anelse om hva det betyr, og de setter foten ned og krever mer gjennomsnittlig kode per time i stedet. Å få dem til å forstå at det er nødvendig å ta sjanser for å lage banebrytende produkter er en tapt sak. Jeg vil kalle dette

> *Corneliusens to lover om programvareledelse (den første og den siste): Hvis alt som brukes er gjennomsnittlige standardløsninger, blir produktet garantert gjennomsnittlig.*

Men la oss komme tilbake til å diskutere C og dets storhet: C lar deg skrive kode som er tvilsom. Det finnes en god løsning på det problemet: Ikke gjør det! Et tegn på en dårlig C-programmerer er at de har en flyktig forståelse av minne, pekere og allokering og ikke forstår et viktig konsept: Å telle fra 0 - null. Det er så enkelt, og så lett å gjøre feil.

I mai 2020 leste jeg en artikkel kalt *Memory Safety*[3] om, åpenbart, minnesikkerhet i Googles *Chromium*-prosjekt, brukt i Googles nettleser, og Microsofts tredje forsøk, eller kanskje det er deres fjerde. Mistet tellingen. Uansett, la meg sitere fra artikkelen:

> *Around 70% of our high severity security bugs are memory unsafety problems (that is, mistakes with C/C++ pointers). Half of those are use-after-free bugs.*

Ting som det skjer alltid når du ikke har kontroll. Kanskje det er for mange dårlige, eller gjennomsnittlige, programmerere som jobber med det? For det første er det helt feil å beskrive det som C/C++. Det er enten C, eller det er C++. Ikke begge. Og minnebehandlingen i C er ekstremt enkel. C++ kan gjøres så komplisert du vil: Vanligvis blir det mer komplisert enn nødvendig, og rare feil begynner å dukke opp, som de nevnte use-after-free bugs.

I en seksjon kalt *What we're trying* sier de:

> *In particular, we feel it may be necessary to ban raw pointers from C++.*

En greie som var populær for noen år siden var å kalle programmering et håndverk. Jeg kommer ikke til å gå inn i en diskusjon om det, men å prøve å forklare utsagnet over i håndverkstermer kan kanskje gi litt mening for de som tror på det:

Tenk deg at det blir oppdaget at de fleste hodeskader på en byggeplass er forårsaket av murstein som faller ned fra stillasene. Den åpenbare løsningen ville være å forby murstein. Og som en følge av det vil alle bygninger heretter bygges av tre. Jeg er ikke sikker på at høyhus bygget i tre vil være særlig populære eller trygge. Neste år vil de foreslå at alle verktøyene fjernes, siden de gir mange håndrelaterte skader.

Å fjerne de verktøy og byggematerialer som er nødvendige for å lage en skikkelig bygning, gjør ikke bygget bedre. Jeg tror det er bedre å erstatte de dårlige entreprenørene, men som jeg allerede har prøvd å forklare, er det i dag som om alle sikter seg inn mot gjennomsnittet ved å formalisere programmering med såkalte "best practices" og "programming patterns": De dårlige programmererne blir gjennomsnittlige, og de gode også. Kanskje de finner ut av det, eller kanskje ikke. I mellomtiden ser jeg på andre nettlesere. Vi hadde det gøy en stund, Chrome! Kanskje jeg prøver Microsoft Edge... å, vent. Ja, det skipet har glidd av kartet.

Noe som bringer meg til slutten, og til et veldig dominerende problem i programmeringsverdenen i dag: Rekruttering. Jeg er ganske sikker på at det nå er flere rekrutteringsselskaper enn faktiske programmerere som søker jobb. Rekrutterere er i utgangspunktet uvitende om programmering, så de får en spesifikasjon fra en uvitende programvareleder og følger den eksakt. De vil bare greie å ansette gode programmerere ved et uhell. I fotballterminologi, og det er en sport jeg bryr meg lite om, rekrutterer de en endeløs rekke av tredje divisjons midtbanespillere. Det eneste som kan sies om dem er at de er helt gjennomsnittlige på alle måter, og at de aldri vil prestere over evne. Alt du får er karriereguttа, som tror de kan lære programmering som om det er et håndverk og bare gjør programmering på jobben, og blir dårlige ledere når de finner ut at de suger. Rekrutteringssystemet i dag samsvarer fint med målet om at alt skal være gjennomsnittlig.

Og mens vi er i gang; de fleste slike rekrutterere hevder at kundene betaler konkurransedyktige lønninger. Det er i beste fall misvisende, siden alle betaler det samme. Jeg vil kalle det gjennomsnittlig, men det høres antageligvis ikke like bra ut i en jobbannonse. Jeg kan faktisk dokumentere det, av personlig erfaring: Jeg har avverget et stort antall slike rekrutterere (rundt 40 hittil, antallet øker hver uke) ved å spørre om hva lønnen er, og statistikken er klar: 30% svarte ikke og gikk videre til lettere bytte, 25% prøvde å unngå spørsmålet, og de resterende 45% oppga helt gjennomsnittlige tall. Det skal bemerkes at det i Norge er trivielt å finne gjennomsnittslønnen: De to største interesseorganisasjonene for programmerere (blant andre ting), NITO og Tekna, har utmerket statistikk over dette som utgis på deres nettsider. Noen bør informere rekrutterere, som ofte opererer fra fjerne land.

Alt innen programvareutvikling konvergerer mot gjennomsnittet. La oss fortsette med noe gøy isteden.

[3] https://www.chromium.org/Home/chromium-security/memory-safety